Oracle® NoSQL Database

Administrator's Guide

Release 23.1
E85373-35
May 2023

Oracle NoSQL Database Administrator's Guide, Release 23.1

E85373-35

Copyright © 2011, 2023, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Conventions Used in This Book xiv
Diversity and Inclusion xiv

1 Introduction

Introduction to Oracle NoSQL Database 1-1

2 Install and Upgrade

Installing Oracle NoSQL Database 2-1
- Installation Prerequisites 2-1
- Installation 2-2

Upgrading an Existing Oracle NoSQL Database Deployment 2-3
- Preparing to Upgrade 2-4
- General Upgrade Notes 2-4
- Steps to Upgrade 2-6
- Upgrading JDK on your Oracle NoSQL Database deployment 2-10

3 Configure

Configuration Basics 3-1
- Installation Configuration Parameters 3-1
- Configuring the Firewall 3-4

Configuring security in a data store 3-5
- Basics of data store security 3-5
- Configuring security using securityconfig tool 3-5
- Create users and configure security with remote access 3-6

Configure a single node KVLite 3-7
Configuring a single region data store 3-8
- Configuring your data store installation 3-8
- Using Plans 3-11
  - Tracking Plan Progress 3-11

Plan States 3-12
Reviewing Plans 3-12
Plan Ownership 3-13
Pruning Plans 3-13
Start the Administration CLI 3-14
Name your data store 3-15
Create a Zone 3-15
Create an Administration Process on a Specific Storage Node 3-17
Create a Storage Node Pool 3-18
Create the Remainder of your Storage Nodes 3-19
Create and Deploy Replication Nodes 3-20
Smoke Testing the System 3-21
Create a script to configure the data store 3-23
Troubleshooting 3-24
Where to Find Error Information 3-25
Service States 3-25
Useful Commands 3-26
Configure data store - Advanced scenarios 3-27
Create Additional Admin Processes 3-27
Configuring with Multiple Zones 3-28
Adding Secondary Zone to the Existing Topology 3-34
Oracle NoSQL Database Proxy 3-39
About the Oracle NoSQL Database Proxy 3-39
Configuring the Proxy 3-40
Using the Proxy in a non-secure data store 3-41
Using the Proxy in a secure data store 3-46
Oracle NoSQL Database Drivers 3-54
About Oracle NoSQL Database SDK drivers 3-55
Obtaining a NoSQL handle 3-57
Creating Regions 3-67
Creating Tables and Indexes 3-67
Adding Data 3-78
Reading Data 3-88
Using Queries 3-93
Deleting Data 3-101
Modifying Tables 3-106
Drop Regions 3-108
Drop Tables and Indexes 3-109
Handling Errors 3-112
Configuring Multi-Region Data Stores 3-117
Use Case 1: Set up Multi-Region Environment 3-118

Deploy the data store 3-118
Set Local Region Name 3-120
Configure XRegion Service 3-121
Start XRegion Service 3-125
Create Remote Regions 3-126
Create Multi-Region Tables 3-127
Access and Manipulate Multi-Region Tables 3-132
Stop XRegion Service 3-133
Use Case 2: Expand a Multi-Region Table 3-133
Prerequisites 3-133
Create MR Table in New Region 3-137
Add New Region to Existing Regions 3-138
Access MR Table in New and Existing Regions 3-141
Use Case 3: Contract a Multi-Region Table 3-142
Alter MR Table to Drop Regions 3-142
Use Case 4: Drop a Region 3-143
Prerequisites 3-143
Isolate the Region 3-144
Drop MR Tables in the Isolated Region 3-145
Drop the Isolated Region 3-145
Use Case 5: Backup and Restore a Multi-Region Table 3-147
Troubleshooting multi-region data store setup 3-150

4 Administer

Changing the Store's Topology 4-1
Determining Your Store's Configuration 4-1
Steps for Changing the Store's Topology 4-2
Make the Topology Candidate 4-3
Transforming the Topology Candidate 4-4
View the Topology Candidate 4-8
Validate the Topology Candidate 4-8
Preview the Topology Candidate 4-9
Deploy the Topology Candidate 4-9
Verify the Store's Current Topology 4-11
Deploying an Arbiter Node Enabled Topology 4-13
Backup and Recovery 4-18
Backing Up the Store 4-18
Taking a Snapshot 4-19
Copying a Snapshot 4-21
Deleting a Snapshot 4-21

Managing Snapshots 4-21
Recovering the Store 4-23
Using the Load Program 4-24
Restoring Directly from a Snapshot 4-28
Recovering from Data Corruption 4-28
Detecting Data Corruption 4-29
Data Corruption Recovery Procedure 4-29
Replacing a Failed Disk 4-31
Replacing a Failed Storage Node 4-33
Using a New Storage Node 4-33
Task for an Identical Node 4-35
Repairing a Failed Zone by Replacing Hardware 4-37
Managing your kvstore 4-38
Increasing Storage Node Capacity 4-38
Managing Storage Directory Sizes 4-41
Managing Disk Thresholds 4-42
Specifying Storage Directory Sizes 4-43
Specifying Differing Disk Capacities 4-44
Monitoring Disk Usage 4-45
Handling Disk Limit Exception 4-47
Managing Admin Directory Size 4-55
Admin is Working 4-56
Admin is not Working 4-56
Disabling Storage Node Agent Hosted Services 4-57
Verifying the Store 4-57
Erasing Data 4-62
Setting Store Parameters 4-62
Changing Parameters 4-63
Setting Store Wide Policy Parameters 4-64
Admin Parameters 4-64
Changing Admin JVM Memory Parameters 4-65
Storage Node Parameters 4-67
Replication Node Parameters 4-70
Arbiter Node Parameters 4-72
Global Parameters 4-72
Security Parameters 4-73
Admin Restart 4-74
Replication Node Restart 4-75
Removing an Oracle NoSQL Database Deployment 4-76
Modifying Storage Node HA Port Ranges 4-76
Modifying Storage Node Service Port Ranges 4-78

Storage Node Not Deployed 4-78
Storage Node Deployed 4-79
Availability, Failover and Switchover 4-80
Availability and Failover 4-80
Replication Overview 4-81
Loss of a Read-Only Replica Node 4-81
Loss of a Read/Write Master 4-82
Unplanned Network Partitions 4-83
Master is in the Majority Node Partition 4-83
Master is in the Minority Node Partition 4-84
No Majority Node Partition 4-85
Failover and Switchover Operations 4-85
Repairing a Failed Zone 4-86
Performing a Failover 4-86
Performing a Switchover 4-93
Zone Failover 4-99
Durability Summary 4-99
Consistency Summary 4-100

5 Tools and Utilities

Using Oracle NoSQL Database Migrator 5-1
Overview 5-1
Terminology used with Oracle NoSQL Database Migrator 5-2
Workflow for Oracle NoSQL Database Migrator 5-4
Sources and Sinks 5-12
Supported Sources and Sinks 5-12
Source and Sink Security 5-13
Source Configuration Templates 5-14
Sink Configuration Templates 5-49
Transformation Configuration Templates 5-81
Mapping of DynamoDB table to Oracle NoSQL table 5-85
Oracle NoSQL to Parquet Data Type Mapping 5-86
Mapping of DynamoDB types to Oracle NoSQL types 5-87
Use Case Demonstrations 5-88
Migrate from Oracle NoSQL Database Cloud Service to a JSON file 5-88
Migrate from Oracle NoSQL Database On-Premise to Oracle NoSQL Database Cloud Service 5-93
Migrate from JSON file source to Oracle NoSQL Database Cloud Service 5-95
Migrate from MongoDB JSON file to an Oracle NoSQL Database Cloud Service 5-99
Migrate from DynamoDB JSON file to Oracle NoSQL Database 5-101

Migrate from DynamoDB JSON file in AWS S3 to an Oracle NoSQL Database Cloud Service 5-105
Migrate from CSV file to Oracle NoSQL Database 5-110
Troubleshooting the Oracle NoSQL Database Migrator 5-114
Using Plugins and Extensions for Development 5-117
About Oracle Enterprise Manager (OEM) Plugin 5-117
Importing and Deploying the EM Plug-in 5-118
Deploying Agent 5-119
Adding NoSQL Database Targets 5-121
Components of a NoSQL Store 5-126
Store Targets 5-126
About IntelliJ Plugin 5-130
Setting Up IntelliJ Plug-in 5-131
Creating a NoSQL Project in IntelliJ 5-131
Connecting to Oracle NoSQL Database from IntelliJ 5-132
Managing Tables Using the IntelliJ Plugin 5-133
About Eclipse plugin 5-137
About Oracle NoSQL Database Visual Studio Code Extension 5-138
Installing Oracle NoSQL Database Visual Studio Code Extension 5-138
Connecting to Oracle NoSQL Database from Visual Studio Code 5-140
Managing Tables Using Visual Studio Code Extension 5-145
Removing a Connection 5-151

6 Monitor

Monitoring the Store 6-1
Events 6-2
Log File Compression 6-3
Software Monitoring 6-4
System Log File Monitoring 6-5
Monitoring for Storage Nodes (SN) 6-6
Monitoring for Replication Nodes (RN) 6-23
Monitoring for Arbiter Nodes 6-27
Monitoring for Administration (Admin) Nodes 6-29
Hardware Monitoring 6-31
Monitoring for Hardware Faults 6-31
Servers 6-44
Standardized Monitoring Interfaces 6-71
Java Management Extensions (JMX) 6-71
Displaying the Oracle NoSQL Database MBeans 6-72
Enabling JMX Monitoring 6-72
Using ELK to Monitor Oracle NoSQL Database 6-73

Enabling the Collector Service 6-74
Setting Up Elasticsearch 6-75
Setting Up Kibana 6-75
Setting Up Logstash 6-75
Setting Up Filebeat on Each Storage Node 6-76
Configure security for the Elastic Stack 6-76
Using Kibana for Analyzing Oracle NoSQL Database 6-77
Creating Index Patterns 6-79
Analyzing the Data 6-79

7 Reference

Terminologies used in Oracle NoSQL Database 7-1
Admin CLI Reference 7-2
aggregate 7-4
aggregate table 7-4
await-consistent 7-5
change-policy 7-6
configure 7-6
connect 7-7
connect admin 7-7
connect store 7-8
delete 7-8
delete kv 7-9
delete table 7-9
execute 7-10
exit 7-11
get 7-11
get kv 7-11
get table 7-14
help 7-15
hidden 7-15
history 7-16
load 7-16
logtail 7-18
namespace 7-18
page 7-19
ping 7-19
plan 7-26
plan add-index 7-28
plan add-table 7-28

plan cancel 7-30
plan change-parameters 7-31
plan change-storagedir 7-32
plan change-user 7-34
plan create-user 7-34
plan deploy-admin 7-35
plan deploy-datacenter 7-36
plan deploy-sn 7-36
plan deploy-topology 7-37
plan deploy-zone 7-39
plan deregister-es 7-41
plan drop-user 7-41
plan enable-requests 7-41
plan evolve-table 7-43
plan execute 7-44
plan failover 7-45
plan grant 7-46
plan interrupt 7-46
plan migrate-sn 7-46
plan network-restore 7-47
plan register-es 7-48
plan remove-admin 7-48
plan remove-datacenter 7-49
plan remove-index 7-49
plan remove-sn 7-49
plan remove-table 7-49
plan remove-zone 7-50
plan repair-topology 7-50
plan revoke 7-51
plan start-service 7-51
plan stop-service 7-54
plan verify-data 7-55
plan wait 7-57
pool 7-58
pool clone 7-58
pool create 7-58
pool join 7-59
pool leave 7-59
pool remove 7-59
put 7-60
put kv 7-60

put table 7-61
repair-admin-quorum 7-62
show 7-62
show admins 7-63
show datacenters 7-64
show events 7-64
show faults 7-65
show indexes 7-66
show mrtable-agent-statistics 7-67
show parameters 7-73
show perf 7-74
show plans 7-74
show pools 7-75
show snapshots 7-75
show regions 7-75
show tables 7-75
show topology 7-75
show upgrade-order 7-76
show users 7-77
show versions 7-77
show zones 7-78
snapshot 7-78
snapshot create 7-79
snapshot remove 7-79
table 7-79
table-size 7-79
timer 7-83
topology 7-83
topology change-repfactor 7-84
topology change-zone-arbiters 7-84
topology change-zone-master-affinity 7-85
topology change-zone-type 7-85
topology clone 7-85
topology contract 7-85
topology create 7-86
topology delete 7-87
topology list 7-87
topology preview 7-87
topology rebalance 7-88
topology redistribute 7-88
topology validate 7-88

topology view 7-88
verbose 7-89
verify 7-89
verify configuration 7-89
verify prerequisite 7-90
verify upgrade 7-90
Admin Utility Command Reference 7-90
diagnostics 7-91
generateconfig 7-91
help 7-96
kvlite 7-96
load admin metadata 7-97
load store data 7-97
makebootconfig 7-98
ping 7-104
Ping Command Line Parameters 7-104
Ping Exit Codes 7-106
Ping Report Text Output 7-108
Ping Report JSON Output 7-109
restart 7-111
runadmin 7-112
securityconfig 7-113
start 7-114
status 7-114
stop 7-115
version 7-115
xrstart 7-115
xrstop 7-116
Initial Capacity Planning 7-116
Shard Capacity 7-117
Application Characteristics 7-118
Shard Storage and Throughput Capacities 7-119
Memory and Network Configuration 7-120
Machine Physical Memory 7-121
Sizing Advice 7-121
Determine JE Cache Size 7-122
Machine Network Throughput 7-123
Estimate total Shards and Machines 7-124
Number of Partitions 7-125
Tuning 7-125
Turn off the swap 7-126

Linux Page Cache Tuning 7-126
OS User Limits 7-127
File Descriptor Limits 7-127
Process and Thread Limits 7-128
Linux Network Configuration Settings 7-128
Server Socket Backlog 7-129
Isolating HA Network Traffic 7-129
Receive Packet Steering 7-129
MTU Size 7-130
Check AES Intrinsics Settings 7-131
Viewing Key Distribution Statistics 7-131
Examples: Key Distribution Statistics 7-136
Solid State Drives (SSDs) 7-137
Trim requirements 7-137
Enabling Trim 7-137
Diagnostics Utility 7-137
Setting up the tool 7-138
Packaging Information and Files 7-139
Verifying Storage Node configuration 7-141

Preface

This document describes how to install and configure Oracle NoSQL Database (Oracle NoSQL Database).

This book is aimed at the systems administrator responsible for managing an Oracle NoSQL Database installation.

Conventions Used in This Book

The following typographical conventions are used within this manual:

Information that you are to type literally is presented in `monospaced` font.

Variable or non-literal text is presented in *italics*. For example: "Go to your *KVHOME* directory."

> **Note:**
>
> Finally, notes of special interest are represented using a note block such as this.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1
Introduction

The article in this section gives an introduction to Oracle NoSQL Database.

Introduction to Oracle NoSQL Database

Oracle NoSQL Database is a distributed, shared-nothing, non-relational database that provides large-scale storage and access to key/value, JSON, and tabular data. It can deliver predictable, low latencies to simple queries at any scale and is designed from the ground up for high availability.

Oracle NoSQL Database's shared-nothing architecture allows it to scale horizontally to meet exceptionally high throughput demands while delivering predictable low latencies. Oracle NoSQL Database requires minimal administration and contains many ***self-healing*** features that enable it to remain ***always-on*** during failures- hardware failures, network partition failures, or even entire data center disasters.

Oracle NoSQL Database offers highly flexible deployment and various methods to access the data store from your application. For applications that require an embedded, ultra-low latency, zero administration database, it can be directly embedded into a Java application. In this deployment scenario, applications can start and stop the database using APIs. When using the Java Direct Driver , applications can read and write data to the database. In most scenarios, Oracle NoSQL is deployed on a cluster of commodity computers connected by a high-speed network. In this deployment scenario, applications must choose a programming language SDK to communicate with the Oracle NoSQL Database cluster. Oracle NoSQL Database offers two types of language SDKS:

1. **Direct driver:** This type of SDK will connect directly to every Oracle NoSQL node in the cluster using TPC/IP. Hence, care must be taken to ensure a network route between the application and every Oracle NoSQL node in the database cluster. Currently, the only supported programming language for direct drivers is Java .
2. **Standard:** This type of SDK will connect to the database using the HTTP protocol via the Oracle NoSQL HTTP proxy. Since standard SDKs use HTTP, you need only ensure a network route between your application code and the load balancer if using one, or between the application and the HTTP proxy if not using a load balancer.

Oracle NoSQL Database supports many of the most popular programming languages and frameworks with idiomatic language APIs and data structures, giving your application language native access to data stored in NoSQL Database. It currently supports the following programming languages and frameworks: Javascript (Node.js), Python, Java, Golang, C#/.NET, and Spring Data. You can also navigate the database as you develop your code with plugins for one of the following supported integrated development environments: Visual Studio Code, IntelliJ, or Eclipse.

2
Install and Upgrade

The articles in this section include steps to install a new Oracle NoSQL Database or upgrade the software of your Oracle NoSQL Database deployment.

Installing Oracle NoSQL Database

This article describes the installation process for Oracle NoSQL Database in a multi-host environment. For smaller systems, when you already know the number of Storage Nodes you will use, follow the subsequent instructions. If you are planning a large system for production use, in addition to the following instructions, please read Initial Capacity Planning to estimate the number of Storage Nodes you need to use to install the software. The Oracle NoSQL Database will make the best use of the Storage Nodes you provide.

- Installation Prerequisites
- Installation

Installation Prerequisites

Make sure that you have Java SE 11 or later installed on all of the Storage Nodes that you are going to use for the Oracle NoSQL Database installation. It is recommended to use Java SE 17. Use this command to verify the version:

```
java -version
```

> **Note:**
>
> Oracle NoSQL Database is compatible with Java SE 11 (64 bit) or later versions. It is tested and certified against Oracle Java SE 17 (64 bit). It is recommended that you upgrade your systems to the latest Java releases to take advantage of all bug fixes and performance improvements. See Release Notes - Overview for more details on Java requirements.

Be sure that the `jps` tool is working. Installing the JDK makes the `jps` tool available for use by the Storage Node Agent (SNA) to optimally manage Oracle NoSQL Database processes. The `jps` tool also allows you to diagnose any issues that may turn up. While Oracle NoSQL Database can operate without the `jps` tool, its absence diminishes the database's ability to manage its processes.

If the JDK and its tools are installed correctly, the output from invoking `jps` should list at least one Java process (the `jps` process itself). Use this command to verify successful installation:

```
% jps
16216 Jps
```

> **Note:**
>
> You must run the commands listed above as the same OS user who will run the Oracle NoSQL Database SNA processes.

Linux is the officially supported platform for Oracle NoSQL Database. Running the Oracle NoSQL Database requires a 64-bit JVM.

You do not necessarily need root access on each Storage Node for the installation process.

Finally, make sure that each of the target Storage Nodes is running some sort of reliable clock synchronization. This is to ensure that communication and resource sharing between the target Storage Nodes are as efficient as possible. Generally, a synchronization delta of less than half a second is required. Network Time Protocol `(ntp)` is sufficient for this purpose.

Installation

Follow this procedure to install the Oracle NoSQL Database.

1. Choose a directory where the Oracle NoSQL Database package files (libraries, Javadoc, scripts, and so forth) should reside. It is recommended that you use the same directory path on all Storage Nodes in the installation. Use different directories for the Oracle NoSQL Database package files (referred to as `$KVHOME` in this document) and the Oracle NoSQL Database data (referred to as `$KVROOT`). Both the `$KVHOME` and `$KVROOT` directories should be local to the Storage Node, and not on a Network File System.

   > **Note:**
   >
   > To make future software upgrades easier, adopt a convention for `$KVHOME` that includes the release number. Always use a `$KVHOME` location such as `/var/kv/kv-M.N.O`, where `M.N.O` represent the software *release.major.minor* numbers. You can achieve this easily by using unzip or untar on the distribution into a common directory (`/var/kv` in this example).

2. Extract the contents of the Oracle NoSQL Database package (`kv-M.N.O.zip` or `kv-M.N.O.tar.gz`) to create the `$KVHOME` directory. For example, `$KVHOME` is the `kv-M.N.O/` directory created by extracting the package). If `$KVHOME` resides on a shared network directory (which is not recommended) you need only unpack it on one Storage Node. If `$KVHOME` is local to each Storage Node, unpack the package on each Storage Node.

3. Verify the installation using the following command on one of the Storage Nodes:

   ```
   java -Xmx64m -Xms64m -jar $KVHOME/lib/kvclient.jar
   ```

You should see some output that looks like this:

```
23.1.21 2023-04-18 21:25:44 UTC  Build id: 477e7f102ab4 Edition: Client
```

where `23.1.21` is the database version number.

> **Note:**
>
> Oracle NoSQL Database is a distributed system and the runtime needs to be installed on every Storage Node in the data store. While the entire contents of the Oracle NoSQL Database package do not need to be installed on every Storage Node, the contents of the lib directory must be present.

Upgrading an Existing Oracle NoSQL Database Deployment

This article describes how to upgrade the software of your Oracle NoSQL Database deployment.

Upgrading a data store from an existing release to a new release can be accomplished one Storage Node at a time because Storage Nodes running a mix of two releases are permitted to run simultaneously in the same data store. This allows you to strategically upgrade Storage Nodes in the most efficient manner.

Installing new software requires that you restart each replication node (referred to as **node**). Upgrading a data store while the store remains online and available to clients is called *rolling upgrade*. A rolling upgrade is useful in most cases, since downtime is undesirable in any system. Depending on your data store's configuration, it is sometimes possible to perform a rolling upgrade. Upgrading a system after shutting down the data store and making it offline is called *offline upgrade*.

Even if your data store can support a rolling upgrade, you may sometimes want to perform an offline upgrade, which involves these steps:

1. Shutting down all nodes.
2. Installing new software on each Storage Node.
3. Restarting each node.

While an offline upgrade is a simpler process in some ways, your data store is unavailable for the duration of the upgrade.

You can perform a rolling upgrade if the data store's replication factor is greater than two. With a replication factor greater than two, shards can maintain their majorities and continue reading and writing data on behalf of clients. Meanwhile, you can restart and upgrade software on each Storage Node, one at a time.

If the replication factor is 2, the shards cannot maintain their majorities through a node restart. This means that each shard will be unavailable while you shutdown a node in the shard, upgrade the software, and restart the node. This is only true if there are no arbiter nodes configured. An arbiter node is capable of supporting write availability in one of these scenarios:

- When the primary replication factor is two and a single replication node becomes unavailable.

- When only two replication nodes are available and they are unable to communicate to determine which one of them is the master.

Steps to upgrade an existing database:

- Preparing to Upgrade
- General Upgrade Notes
- Steps to Upgrade
- Upgrading JDK on your Oracle NoSQL Database deployment

Preparing to Upgrade

Oracle NoSQL Database supports upgrades from releases for the current year and prior two calendar years. For example, to upgrade a store to the 23.x release, the store must be running release 21.x or later.

Before beginning the upgrade process, create a backup of the store by making a snapshot. See Taking a Snapshot.

In Oracle NoSQL Database, configuration changes and other administrative activities involve `plans`. For information about plans, see Using Plans.

> **Note:**
>
> During the upgrade process, you should not create any plans until all services in the data store have been upgraded.

As soon as possible after upgrading the service components, upgrade any application programs that use the kvstore client library. Also make sure to upgrade the Oracle NoSQL Database Proxy. This is necessary to compile the applications with the new software and to relink the application using the libraries of the new release of Oracle NoSQL Database.

General Upgrade Notes

This article contains upgrade information that is generally true for all versions of Oracle NoSQL Database.

When Oracle NoSQL Database is first installed, the software is placed in a `$KVHOME` directory. Such a directory can exist on each Storage Node, or be shared by multiple Storage Nodes (for example, using NFS). Here, you refer to this existing `$KVHOME` location, as `$OLD_KVHOME`.

> **Note:**
>
> It is recommended that installations adopt a naming convention for `$KVHOME` that includes the release number. If you always use a `$KVHOME` location such as `/var/kv/kv-M.N.O`, where `M.N.O` represents the release.major.minor numbers, the version is easily visible. You can achieve this naming by unzipping or untarring the distribution into a common directory, `/var/kv` in this example.

Installing new software requires that each replication node be restarted. Oracle NoSQL Database is a replicated system. To avoid excessive failover events, it is advised to restart any replication node that is running as a MASTER after all those marked as REPLICA. You can use the ping command to get the list of MASTER and REPLICA of Storage Nodes in your data store.

You can use the following command to understand the order of upgrade.

```
show upgrade-order
```

To make the upgrade process easy to debug when upgrading the software while the Storage Node is stopped, move the existing log files under `$KVROOT` and `$KVROOT/<storename>/log` to any other directory.

Use the host and registry port for any active replication node in the store. For example, if `rg1-rn1` and `rg2-rn1` are running as MASTER, restart those last.

When upgrading your data store, place the updated software in a new `$KVHOME` directory on a Storage Node running the admin service. Here you refer to the new `$KVHOME` directory as `$NEW_KVHOME`. If the `$KVHOME` and `$NEW_KVHOME` directories are shared by multiple Storage Nodes (for example, using NFS), maintain both directories while the upgrade is in progress. After the upgrade is complete, you no longer need the original `$KVHOME` directory. In this case, you must modify the start up procedure on each Storage node to refer to the `$NEW_KVHOME` directory so the Storage Node uses the new software.

> **Note:**
>
> In cases where each Storage Node has its own copy of the software installation, then it is possible to replace the installation in place and not modify the value of `$KVHOME`.

To add security after upgrading from a non-secure data store, see Adding Security to a New Installation in the *Security Guide*.

Upgrading the XRegion Service Agent

You should upgrade your data store first before upgrading the XRegion Service agent. If the agent is upgraded first before the data store is upgraded, the agent may get blocked when accessing the new system table and wait for the data store to be upgraded. To configure the XRegion Service agent See, Configure XRegion Service.

Steps to Upgrade

To upgrade your data store, first install the latest software to upgrade the Admin CLI. See Installation for more details. Before proceeding with the upgrade, see General Upgrade Notes.

Then, use the upgraded CLI to verify that all the Storage Nodes do not violate any basic requirements.

```
kv-> verify prerequisite
...
Verification complete, no violations.
```

Once all violations (if any) are corrected, install the latest software on a Storage Node that is running an admin service.

Do the following:

1. On a Storage Node running an admin service:
   a. Place the updated software in a new `$KVHOME` directory. If Storage Nodes share this directory using NFS, you only need to do this once for each shared directory.
   b. Stop the Storage Node using the CLI. Stopping the Storage Node shuts down the admin service on that Storage Node.

      If you have configured the Storage Node to automatically start the Storage Node Agent when the system reboots using `/etc/init.d`, or some other mechanism, then you need to modify your system startup script to point to `$NEW_KVHOME`. This is needed so that `$KVHOME` points to the correct directory after any system reboot.

      Once you have modified the system startup script, shutdown the Storage Node:

      ```
      java -Xmx64m -Xms64m \
      -jar $KVHOME/lib/kvstore.jar stop -root $kvroot
      ```

   c. Restart the Storage Node using the updated software release:

      ```
      nohup java -Xmx64m -Xms64m \
      -jar $NEW_KVHOME/lib/kvstore.jar start -root $kvroot &
      ```

      (If the system is configured to automatically restart the Storage Node Agent, this step may not be necessary.)
   d. Use the CLI to connect to the Storage Node which is now running the updated software release. See

      ```
      java -Xmx64m -Xms64m \
      -jar $NEW_KVHOME/lib/kvstore.jar runadmin -port 5100 -host node1
      \
      -security $KVROOT/security/root.login
      ```

e. Verify that all the Storage Nodes in the data store are running the proper software level required to upgrade to the new release.

```
kv-> verify prerequisite
Verify: starting verification of store mystore
based upon topology sequence #315
...
Verification complete, no violations.
```

The most likely reason for a violation is if you are attempting a release level downgrade. For example, it is illegal to downgrade from a higher minor release to a **lower minor release**.

> **Note:**
>
> It is legal to downgrade from a higher major release to a **lower major release**. For example downgrading from 22.3.15 to 22.3.14 would be legal, while downgrading from 22.3.15 to 22.2.15 would not be legal.

In any case, if the `verify prerequisite` command shows violations, resolve the situation before you attempt to upgrade the identified Storage Nodes.

f. Obtain an ordered list of the Storage Nodes to upgrade.

```
kv-> show upgrade-order
Calculating upgrade order, target version: 22.3.15, prerequisite:
20.1.5
sn3 sn4
sn2 sn5
sn6
```

This command displays one or more Storage Nodes on a line. Multiple Storage Nodes on a line are separated by a space. If multiple Storage Nodes appear on a single line, then those Storage Nodes can be safely upgraded simultaneously, if desired. When multiple Storage Nodes are upgraded simultaneously, the upgrade must be completed on all Storage Nodes before the Storage Nodes next on the list can be upgraded. If at some point you lose track of which group of Storage Nodes should be upgraded next, you can always run the `show upgrade-order` command again. In the above example, you can upgrade sn3 and sn4 simultaneously. Then upgrade sn2 and sn5 simultaneously. And, finally, upgrade sn6.

> **Note:**
>
> You must completely upgrade a group of Storage Nodes before continuing to the next group. That is, upgrade sn3 and sn4 before you proceed to upgrade sn2, sn5, or sn6.

2. For each of the Storage Nodes in the first group of Storage Nodes to upgrade (sn3 and sn4, in this example):

a. Place the new release software in a new `$KVHOME` directory. If Storage Nodes share this directory using NFS, you need to do this only once for each shared directory.

b. Stop the Storage Node using the CLI utility.

   If you have configured the Storage Node to automatically start the Storage Node Agent when the system reboots using `/etc/init.d`, or some other mechanism, then you need to modify your system startup script to point to `$NEW_KVHOME`. This is needed so that `$KVHOME` points to the correct directory after any system reboot.

   Once you have modified the system startup script, shutdown the Storage Node:

```
java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar stop -root $kvroot
```

c. Restart the Storage Node using the updated software release:

```
nohup java -Xmx64m -Xms64m \
-jar $NEW_KVHOME/lib/kvstore.jar start -root $kvroot &
```

   (If the system is configured to automatically restart the Storage Node Agent, this step may not be necessary.)

3. Verify the upgrade before upgrading your next set of Storage Nodes. This command shows which Storage Nodes have been successfully upgraded, and which Storage Nodes still need to be upgraded.

```
kv-> verify upgrade
Verify: starting verification of store mystore
based upon topology sequence #315
...
Verification complete, 0 violations, 3 notes found.
Verification note: [sn2]    Node needs to be upgraded from
21.2.15 to version 22.3.15 or newer
Verification note: [sn5]    Node needs to be upgraded from
21.2.15 to version 22.3.15 or newer
Verification note: [sn6]    Node needs to be upgraded from
21.2.15 to version 22.3.15 or newer
```

   Those Storage Nodes which have been upgraded are identified with a verification message that includes the current software version number:

```
Verify upgrade: Storage Node [sn3] on node03:5000
Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false] Status: RUNNING   Ver: 22.3.15 2022-12-24
09:50:01 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
```

Those Storage Nodes which still need to be upgraded are identified in two different ways. First, the verification message for the Storage Node indicates that an upgrade is still necessary.

```
Verification note: [sn2]    Node needs to be upgraded from
21.2.15 to version 22.3.15 or newer
```

Second, the very end of the verification output identifies all the Storage Nodes that still need to be upgraded.

```
Verification complete, 0 violations, 3 notes found.
Verification note: [sn2]    Node needs to be upgraded from
21.2.15 to version 22.3.15 or newer
Verification note: [sn5]    Node needs to be upgraded from
21.2.15 to version 22.3.15 or newer
Verification note: [sn6]    Node needs to be upgraded from
21.2.15 to version 22.3.15 or newer
```

Note:

If the verification shows Storage Nodes you thought were upgraded as being still in need of an upgrade, you *must* resolve that problem before upgrading the other Storage Nodes in your data store. As a sanity check, you can verify just those Storage Nodes that you finished upgrading.

```
kv-> verify upgrade -sn sn3 -sn sn4
Verify: starting verification of store mystore
based upon topology sequence #315
...

Verification complete, no violations.
```

4. You can continue upgrading groups of Storage Nodes, as identified by the `show upgrade-order` command. Follow the procedure outlined above. Stop the existing release Storage Node using the existing release stop command, then restart the Storage Node using the new release start command. Continue doing this until all Storage Nodes have been upgraded.

   If at some point you lose track of which group of nodes should be upgraded next, you can always run the `show upgrade-order` command again:

```
kv-> show upgrade-order
Calculating upgrade order, target version: 22.3.15, prerequisite: 20.1.5
sn2 sn5
sn6
```

5. When you are all done upgrading your Storage Nodes, the `verify upgrade` command will show no verification notes at the end of its output:

```
kv-> verify upgrade
Verify: starting verification of store mystore
```

```
based upon topology sequence #315
...

Verification complete, no violations.
```

Upgrading JDK on your Oracle NoSQL Database deployment

Consider that you have a JDK version, say JDK 11 SE, installed on all the Storage Nodes in your data store deployment. But after some time, Oracle releases a new version of the JDK, say JDK 17 SE, that includes security enhancements and bug fixes. Now, you want to upgrade the existing JDK to a newer version of the JDK.

Additionally, during the upgrade, you need to ensure that store remains online and available to clients.

Consider that your existing Oracle NoSQL Database deployment is a data store and is deployed on 3 Storage Nodes (SN1, SN2, and SN3).

To update JDK on your Oracle NoSQL Database deployment:

1. Based on the OS architecture, download and install the required version of JDK from Java SE Downloads.
2. Update the `$JAVA_HOME` and `$PATH` environment variables to point to the updated JDK directory.
3. Verify that in the Storage Node, the JDK is now pointing to the new JDK by running the `java -version` command and verifying the output.
4. Stop the SNA (Storage Node Agent) process in SN1 by running the following command:

   ```
   java -Xmx64m -Xms64m -jar $KVHOME/lib/kvstore.jar stop -root $kvroot
   ```

5. Restart the SNA process in SN1 by running the following command:

   ```
   nohup java -Xmx64m -Xms64m -jar $KVHOME/lib/kvstore.jar start -
   root $kvroot &
   ```

6. Repeat steps 1 through 5 for each Storage Node. Make sure these steps are run sequentially on all the Storage Nodes. For example, run steps 1 to 5 on SN1, followed by SN2, and so on.

3
Configure

The articles in this section provide steps on how to configure Oracle NoSQL Database.

Configuration Basics

Once you have installed Oracle NoSQL Database on each of the Storage Nodes that you are using in your data store (see Installing Oracle NoSQL Database), you must configure the data store. To do this, you use the Administration command line interface (CLI).

- Installation Configuration Parameters
- Configuring the Firewall

Installation Configuration Parameters

Before you configure Oracle NoSQL Database, you should determine the following parameters for each Storage Node in the data store. Each of these parameters are directives to use with the `makebootconfig` utility:

- **root**

  Where the KVROOT directory should reside. The KVROOT directory stores the data of your data store and security related information. There should be enough disk space on each Storage Node to hold the data to be stored in your data store. The KVROOT disk space requirements can be reduced if the *storagedir* parameter is used to store the data at a different location outside the KVROOT directory. It is recommended that you make the KVROOT directory the same local directory path on each Storage Node (but not a shared or NFS mounted directory).

- **port**

  The TCP/IP port through which the Storage Node connects to the Oracle NoSQL Database. This port should be free (unused) on each Storage Node. The default port used in all examples is 5000. This port is sometimes referred to as the *registry port*.

- **harange**

  The replication nodes and Admin process use the harange (high availability range) ports to communicate between each other. For each Storage Node in the data store, specify sequential port numbers, one port for each replication node on the Storage Node, plus an additional port if the Storage Node hosts an Admin. Specify the port range as `startPort,endPort`.

- **servicerange**

  A range of ports that a Storage Node uses to communicate with other administrative services and its managed services. Some of the managed services are the replication nodes for every Storage Node. This optional parameter is useful when Storage Node services must use specific ports for a firewall or other security purposes. By default, the services use anonymous ports. Specify the port range as `startPort,endPort`. For more information, see Storage Node Parameters.

- **store-security**

  Specifies whether security is in use. While this is an optional parameter, it is strongly advised that you configure Oracle NoSQL Database with security enabled.

  Specifying `none` indicates that security will not be in use.

  Specifying `configure` indicates that you want to configure a secure data store. The `makebootconfig` process will then invoke the `securityconfig` utility as part of its operation.

  Specifying `enable` indicates security will be in use. However, you will need to either explicitly configure security by utilizing the security configuration utility(`securityconfig`), or copy a previously created security configuration from another system.

  > **Note:**
  >
  > If you do not specify the `-store-security` parameter, security is configured by default. To complete a secure installation, you must use the `securityconfig` utility to create the security folder before starting up the Storage Node Agents.

- **capacity**

  The total number of replication nodes the Storage Node can support. Capacity is an optional, but extremely important parameter, representing the number of replication nodes. If the Storage Node you are configuring has the resources to support more than one replication node, set the capacity value to the appropriate number. To host a replication node successfully to handle peak run time demand, you need sufficient disk, cpu, memory, and network bandwidth .

  To have your Storage Node host Arbiter Nodes, set the capacity to `0` . A Storage Node with capacity 0 will be allocated as Arbiter Nodes whenever required. For more information see Deploying an Arbiter Node Enabled Topology.

  Consider the following configuration settings for Storage Nodes with a capacity greater than one:

  1. It is recommended that you configure each Storage Node with a capacity equal to the number of available disks on the machine. Such a configuration permits the placement of each replication node on its own disk, ensuring that replication nodes on the Storage Node are not competing for I/O resources. The `-storagedir` parameter lets you specify the directory location for each replication node disk.

     For example:

     ```
     > java -Xmx64m -Xms64m \
         -jar $KVHOME/lib/kvstore.jar makebootconfig \
         -root /opt/ondb/var/kvroot \
         -port 5000 \
         -host node10
         -harange 5010,5025 \
         -capacity 3 \
         -admindir /disk1/ondb/admin01 \
     ```

```
-admindirsize 5000_MB \
-storagedir /disk1/ondb/data \
-storagedir /disk2/ondb/data \
-storagedir /disk3/ondb/data \
-storagedirsize 1_tb \
-rnlogdir /disk1/ondb/rnlog01 \
-rnlogdir /disk2/ondb/rnlog02 \
-rnlogdir /disk3/ondb/rnlog03
```

where `-capacity 3` represents the number of replication nodes on the Storage Node (`node10`). The three replication nodes are in the corresponding disks (`disk1`, `disk2`, `disk3`).

2. Increase the `-harange` parameter to support additional ports required for the replication and Admin Nodes.
3. Increase the `-servicerange` parameter to account for the additional ports required by the replication nodes.

If no value for capacity is specified, it defaults to `1`.

- **admindir**

  The directory path to contain the environment associated with a Storage Node Admin process.

  It is strongly recommended that the Admin directory path resolves to a separate disk. You can accomplish this by creating suitable entries in the `/etc/fstab` directory that attaches the file system on disk to an appropriate location in the overall directory hierarchy. Placing the Admin environment on a separate disk ensures that the Admin is not competing for I/O resources. It also isolates the impact of a disk failure to a single environment.

  If you do not specify an explicit directory path for `-admindir`, the Admin environment files are located in this directory:

  ```
  $KVROOT/KVSTORE/<SNID>/<AdminId>/
  ```

- **admindirsize**

  The size of the Admin storage directory. This is optional but recommended. For more information, see Managing Admin Directory Size.

- **storagedir**

  A directory path that will contain the environment associated with a replication node. When the `-capacity` parameter is greater than 1, it is recommended that you specify a multiple set of `-storagedir` parameter values, one for each replication node that the Storage Node hosts. Each directory path should resolve to a separate disk. You can accomplish this by creating suitable entries in the `/etc/fstab` directory that attaches the file system on disk to an appropriate location in the overall directory hierarchy. Placing each environment on a separate disk ensures that the shards are not competing for I/O resources. It also isolates the impact of a disk failure to a single location.

- **storagedirsize**

  The size of each storage directory. It is strongly recommended that you specify this parameter for each replication node. The Oracle NoSQL Database uses the storage directory size to enforce disk usage, using the `-storagedirsize` parameter value to

calculate how much data to store on disk before suspending write activity. For more information, see Managing Storage Directory Sizes.

- **rnlogdir**

  The directory path to contain the log files associated with a replication node. For capacity values greater than one, specify multiple `rnlogdir` parameters, one for each replication node that the Storage Node is hosting.

  It is recommended that each `rnlogdir` path resolves to a separate disk partition on a replication node. You can accomplish this by creating suitable entries in the `/etc/fstab` directory that attaches the file system on a disk to an appropriate location in the overall directory hierarchy. Placing `rnlogdir` in a distinct partition on the replication node ensures that metrics and errors can be reported and retained, even if the partition containing the data store log files is full. Separating the `rnlogdir` on a distinct partition also isolates the impact of losing complete replication node log files from a `kvroot` disk failure.

  If you do not specify a location for `rnlogdir`, logs are placed under the `$KVROOT/KVSTORE/log` directory by default.

- **num_cpus**

  The total number of processors on the machine available to the replication nodes. This is an optional parameter, used to coordinate the use of processors across replication nodes. If the value is `0`, the system queries the Storage Node to determine the number of processors on the machine. The default value for `num_cpus` is `0`, and examples in this document use that value.

- **memory_mb**

  The total number of megabytes of memory available to the replication node. The system uses the `memory_mb` value to guide specification of the replication node's heap and cache sizes. This calculation is more critical if a Storage Node hosts multiple replication nodes, and must allocate memory between these processes. If the value is `0`, the system attempts to determine the amount of memory on the replication node. The default value for `memory_mb` is `0`, and examples in this document use that value.

- **force**

  Specifies that the command generates the boot configuration files, even if verifying the configuration against the system finds any inaccurate parameters. That means you force the creation of the boot configuration file, even if the value of any of the parameters discussed above (like port, harange etc) is inaccurate.

See makebootconfig for more details.

Configuring the Firewall

Most of the Storage Nodes, either physical or virtual machines, have built-in firewalls. Additionally, you may have separate firewalls in-between machines. In a NoSQL topology, the Storage Nodes need to communicate with one another, so communication must pass through the firewalls. You need to open the firewall ports used by the communication channels in the data store. To make sure your network firewall works with your topology, you should set the ports specified by the `-port` , `-harange`, `-servicerange`, and `-admin-web-port` parameters of the `makebootconfig` command. These four parameters are used to constraint the data store to a limited set of ports. Setting the ports is usually done for security or data center policy reasons. By

default the services in your data store use anonymous ports. To specify a range of ports, you use the format of `startPort,endPort`.

Configuring security in a data store

- Basics of data store security
- Configuring security using securityconfig tool
- Create users and configure security with remote access

Basics of data store security

Oracle NoSQL Database can be configured securely.

In a secure configuration, network communications between NoSQL clients, utilities, and NoSQL data store components are encrypted using SSL/TLS, and all processes must authenticate themselves to the components to which they connect. It is strongly advised that you configure Oracle NoSQL Database with security enabled.

When you configure the Oracle NoSQL Database, the parameter `store-security` specifies whether security is in use. Specifying ***none*** indicates that security will not be in use. Specifying ***configure*** indicates that you want to configure security. When you specify ***configure*** or do not specify the `store-security` parameter, then the **makebootconfig** process will invoke the **securityconfig** utility as part of its operation. Specifying ***enable*** indicates security will be in use. When you specify ***enable*** , you will need to either explicitly configure security by utilizing the security configuration utility(`securityconfig`), or copy a previously created security configuration from another system.

> **Note:**
>
> If you do not specify the `-store-security` parameter, security is configured by default. To complete a secure installation, you must use the `securityconfig` utility to create the security folder before starting up the Storage Node agents.

Configuring security using securityconfig tool

You can run the `securityconfig` tool before or after the `makebootconfig` process. This tool creates the security directory and also creates security related files. The `makebootconfig` utility automatically invokes the `securityconfig` tool in one of the following two scenarios.

- You specify `store-security configure` in the `makebootconfig` command explicitly requesting to configure a secure data store.
- You omit the `store-security` parameter in the `makebootconfig` command. A secure data store is then configured by default.

Invoke the `securityconfig` tool as shown below:

```
java -Xmx64m -Xms64m
-jar $KVHOME/lib/kvstore.jar
securityconfig \
config create -root $KVROOT -kspwd (******)
```

```
Created files
$KVROOT/security/security.xml
$KVROOT/security/store.keys
$KVROOT/security/store.trust
$KVROOT/security/client.trust
$KVROOT/security/client.security
$KVROOT/security/store.passwd (Generated in CE version)
$KVROOT/security/store.wallet/cwallet.sso (Generated in EE version)
Created
```

See Configuring Security with Securityconfig in the *Security Guide* for more details.

If you have more than one Storage Node in your data store, then the security configuration is configured in the first Storage Node using `-store-security configure`). The security directory and all files contained in it should be copied from the first Storage Node to other Storage Nodes to setup security. Zip all the security related files from the first Storage Node to `security.zip`.

```
cd ;
zip -r $HOME/security.zip $KVROOT/security;
cd -
```

Copy the `security.zip` from first Storage Node to other Storage Nodes. In the other Storage Nodes, you will unzip the `security.zip` file and use this security information (copied from the first Storage Node). You then use `-store-security enable` while configuring the remaining Storage Nodes.

Create users and configure security with remote access

You need to create users for a secure cluster.

To configure security with remote access, see the following steps:

- Create the first admin user. In this case, user `root` is defined.

  ```
  kv->execute 'CREATE USER root IDENTIFIED BY "password" ADMIN'
  ```

- Grant the `readwrite` role to the first admin user:

  ```
  kv->execute "GRANT readwrite TO USER root"
  ```

- Generate a password store for the first admin user. This step creates an `root.passwd` file in the `$KVROOT/security` directory. These are the commands to create `root.passwd`.

  ```
  java -Xmx64m -Xms64m \
  -jar $KVHOME/lib/kvstore.jar securityconfig \
  pwdfile create -file $KVROOT/security/root.passwd

  java -Xmx64m -Xms64m \
  -jar $KVHOME/lib/kvstore.jar securityconfig \
  ```

```
pwdfile secret \
-file $KVROOT/security/root.passwd -set -alias root -secret password
```

- Copy the `client.security` file to another file named `root.login`. This `client.security` was created by the `securityconfig` utility earlier.

```
cp $KVROOT/security/client.security $KVROOT/security/root.login
```

- Zip all the user security files. This needs to be copied to all Storage Nodes of the data store.

```
cd $KVROOT/security;
zip -r root.zip root.* client.trust ;
cd -
```

- From every Storage Node (other than the first Storage Node in the data store), unzip the user security files into `$KVROOT/security`.

```
unzip -o $KVROOT/security/root.zip -d $KVROOT/security
```

- You can now access the Admin node running on a Storage Node from another Storage Node remotely as shown below:

```
java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar runadmin \
-port 5000 -host node01 \
-security $KVROOT/security/root.login
```

Configure a single node KVLite

KVLite is a simplified version of the Oracle NoSQL Database.

KVLite is a single shard data store, that is not replicated. It runs in a single process without requiring any administrative interface. You configure, start, and stop KVLite using a command line interface.

KVLite is intended for use by application developers who need to develop and unit test their Oracle NoSQL Database applications. It can be used as a development platform for developers to get familiar with Oracle NoSQL APIs, and test different ways of interacting with these APIs. This is the simplest configuration of a NoSQL database and helps you get started quickly as it does not need any detailed configuration steps. However it is not intended for production deployment, or for performance measurements.

Start KVLite in a secure mode as shown below.

```
java -Xmx64m -Xms64m -jar lib/kvstore.jar kvlite
```

`kvstore` is the name of the data store that gets configured and `kvroot` is the directory where Oracle NoSQL Database data is placed.

Also, KVLite is secure by default. If you want to run KVLite in unsecure mode, you will have to explicitly provide parameters to disable security while installing KVLite as shown below.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
```

Configuring a single region data store

At a high level, configuring your store requires these steps:

- Configuring your data store installation
- Using Plans
- Start the Administration CLI
- Name your data store
- Create a Zone
- Create an Administration Process on a Specific Storage Node
- Create a Storage Node Pool
- Create the Remainder of your Storage Nodes
- Create and Deploy Replication Nodes
- Smoke Testing the System
- Create a script to configure the data store
- Troubleshooting

Configuring your data store installation

Once you determine your data store's configuration information as described in the previous section (see Installation Configuration Parameters), complete the following tasks to configure your data store.

1. Create the initial bootconfig configuration file using the `makebootconfig` command. Do this on each Storage Node.

> **Note:**
>
> Using the `makebootconfig` command to create the configuration file is integrated with the Storage Node on which you run the command. Such integration, checks and validates all parameters and their values against the Storage Node environment before generating the boot configuration files. To bypass verifying any parameters or values for the boot configuration files, use the ***-force*** flag (`makebootconfig -force`).

Following is an example of using `makebootconfig`, using a sample set of parameters and values. For a list of all the `makebootconfig` parameters, see makebootconfig.

```
> mkdir -p $KVROOT
> java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar \
makebootconfig -root $KVROOT \
               -port 5000 \
               -host $KVHOST \
               -harange 5010,5020 \
               -capacity 1 \
               -admindir /export/admin \
               -admindirsize 5000_MB \
               -storagedir /export/data1 \
               -storagedirsize 1_tb \
               -rnlogdir /export/rnlogs
```

> **Note:**
>
> It is strongly recommended that you specify both `storagedir` and `storagedirsize`. If you specify the `-storagedir` parameter, but not `-storagedirsize`, `makebootconfig` displays a warning.

When the `store-security` parameter is omitted from the `makebootconfig` command , a secure data store is configured by default. The `makebootconfig` command internally invokes the `securityconfig` tool to create the security directory and security related files. To configure a non secure data store, specify `store-security none` in the `makebootconfig` command. However it is recommended to configure a secure data store in production environments.

2. Start the Oracle NoSQL Database Storage Node Agent (SNA) on each of the Oracle NoSQL Database Storage Nodes. The SNA manages the Oracle NoSQL Database administrative processes on each Storage Node. It also owns and manages the registry port, which is the main way to communicate with Oracle NoSQL Database on that Storage Node. Before starting the SNA, on each Storage Node, set the environment variable `MALLOC_ARENA_MAX` to 1. Doing this ensures that memory usage is restricted to the specified heap size. To start the SNA on each Storage Node use the `start` command as follows:

```
nohup java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar start -root $KVROOT &
```

> **Note:**
>
> If the replication node or the Admin Service crashes, the SNA restarts the processes.

3. Use the `jps -m` command to verify that the Oracle NoSQL Database processes are running :

```
> jps -m
2830534 kvstore.jar start -root $KVROOT
2830645 ManagedService -root $KVROOT -secdir $KVROOT/security -
class Admin
-service BootstrapAdmin.5000 -config config.xml
```

4. Using `ssh` to reach the node, issue a `ping` command (in security mode) to be sure that the Oracle NoSQL Database client library can contact the Oracle NoSQL Database Storage Node Agent.

> **Note:**
>
> If your data store is a non secure one, the `-security` option in the below command can be omitted.

```
ssh node01
java -Xmx64m -Xms64m -jar $KVHOME/lib/kvstore.jar ping -
host $KVHOST -port 5000
-security $KVROOT/security/client.security

Login as: Anonymous (Enter any user name here)
Anonymous's password: (Enter any password)

SNA at hostname: node01, registry port: 5000 is not registered.
No further information is available
Can't find store topology:
Could not contact any RepNode at: [node01:5000]
```

This return informs you, that only the Storage Node process is running on the Storage Node `node01`. Once Oracle NoSQL Database is fully configured, you can use the `ping` command again to get more details.

If the client library cannot contact the SNA, the `ping` command displays this message:

```
Unable to connect to the storage node agent at host <hostname>,
port 5000, which may not be running; nested exception is:

java.rmi.ConnectException: Connection refused to host: <hostname>;
nested exception is:
java.net.ConnectException: Connection refused
Can't find store topology:
Could not contact any RepNode at: [<hostname>:5000]
```

If the Storage Nodes do not start up, review the `adminboot` and `snaboot` logs in the `$KVROOT` directory to investigate what occurred and to help identify the problem. When the Storage Nodes have all started successfully, you can configure the data store.

> **Note:**
>
> For best results, configure your Storage Nodes so that the SNA starts automatically when the Storage Node boots up. The details of how to do this are beyond the scope of this document, because they depend on how your operating system is designed. See your operating system documentation for information about launching an application automatically at bootup.

Using Plans

You use plans to configure your data store. A plan consists of administrative operations. Plans can modify the state managed by the Admin service, and issue requests to data store components such as Storage Nodes and replication nodes. Some plans consist of simple state-changing operations, while others perform a set of tasks that affect every Storage Node and replication nodes in the data store. For example, you use a plan to create a zone or Storage Node, or to reconfigure parameters on a replication node.

You use the `plan` command, available from the administrative command line interface, to both create and execute plans, as well as to perform many other tasks. For more about using the `plan` command, see CLI Command Reference.

By default, running a `plan` command executes asynchronously in the background. The command line prompt returns as soon as the background process begins. You can check the progress of a running plan using the `show plan` *`id`* command.

You can run a `plan` command synchronously in two ways:

```
plan action_to_complete —wait

plan wait -id plan_id
```

Using either the `-wait` flag or the `plan wait` command, causes the command line prompt to return only after the command completes.

The `-wait` flag and the `plan wait` command are useful when executing plans from scripts, which typically expect each command to finish before processing the next command.

You can also create a plan, but defer its execution using the optional `-noexecute` flag, as follows:

```
plan action -name plan-name -noexecute
```

Later, you can execute the plan on demand as follows:

```
plan execute -id id_num
```

Tracking Plan Progress

There are several ways to track the progress of a plan.

- The `show plan -id` command provides information about the progress of a running plan. Use the optional `-verbose` flag to get more details.

- The CLI `verify` command gives service status information as the plan is executing and services start.

> **Note:**
>
> The `verify` command is of interest for only topology-related plans. If the plan is modifying parameters, such changes may not be visible using the verify command.

- The CLI's `logtail` command lets you follow the store-wide log.

Plan States

Plans can be in any of the following states. A plan can be in only one state at a time. These are the possible states:

Name	Description
APPROVED	The plan exists with correct operations, but is not running.
CANCELED	A plan that is manually `INTERRUPTED` or that experiences an `ERROR` can be terminated. Use the `cancel` command to terminate a plan.
ERROR	If a plan in the `RUNNING` state encounters a problem, it transitions to this state and ends processing without successfully completing. Storage Nodes and replication nodes can encounter an error before the plan processes the error and transitions to an `ERROR` state.
INTERRUPTED	A `RUNNING` plan transitions to this state after the `interrupt` command in the CLI.
INTERRUPT REQUESTED	When a running plan receives an interrupt request, the plan may have to cleanup or reverse previous steps taken during its execution. If the plan transitions to this state, it is to make sure that the data store remains in a consistent state.
RUNNING	The plan is currently executing its commands.
SUCCEEDED	The plan has completed successfully.

You can use the `plan execute` command whenever a plan enters the `INTERRUPTED`, `INTERRUPT REQUESTED` or `ERROR` state. Retrying is appropriate if the underlying problem was transient or has been rectified. When you retry a Plan, it processes the steps again. Each step is idempotent, and can be safely repeated.

Reviewing Plans

You can use the CLI `show plans` command to review the execution history of plans. The command also lists the plan ID numbers, plan names, and the state of each plan. With the plan ID, use the `show plan -id <plan number>` command to see more details about a specific plan.

The next example shows the output of both the `show plans` command and then the `show plan -id <plan number>` command. The `show plan` command returns the plan name, the number of attempts, the start and end date and times, the total number of tasks the plan completed, and the whether the plan completed successfully.

```
kv-> show plans
1 Deploy KVLite        SUCCEEDED
2 Deploy Storage Node  SUCCEEDED
3 Deploy Admin Service SUCCEEDED
4 Deploy KVStore       SUCCEEDED
kv-> show plan -id 3
Plan Deploy Admin Service (3)
Owner: null
State:          SUCCEEDED
Attempt number: 1
Started:        2022-11-22 22:05:31 UTC
Ended:          2022-11-22 22:05:31 UTC
Total tasks:    1
Successful:     1
```

Plan Ownership

In a secure Oracle NoSQL Database deployment, each plan command is associated with its creator as the owner. Only the plan owner can see and operate it. If a plan is created in an earlier version of Oracle NoSQL Database, or in an non secure data store, the owner is `null`.

> **Note:**
>
> The SYSOPER privilege allows a user to perform cancel, execute, interrupt, and wait on any plan.

Users with the `SYSVIEW` privilege can see plans owned by other users, plans with a `null` owner, and plans whose owners have been removed from the Oracle NoSQL Database.

For more information about user privileges and on configuring Oracle NoSQL Database securely, see the Security Guide.

Pruning Plans

The system automatically prunes plans that should be removed. Plans are removed from the Admin Store if they match both of these conditions:

- Are in a terminal state (`SUCCEEDED` or `CANCELLED`)
- Have a Plan ID number that is 1000 less than the most recent Plan ID

For example, if the most recent Plan ID is 2000, the system prunes all plans with ID numbers 1000 or less that are in a terminal state . The system does not remove plans in a non-terminal state.

While pruning plans occurs automatically, you can detect that pruning has occurred in these situations:

- Attempting to show a plan with a specific ID that has been pruned.

- Specifying a range of plans that contains one or more removed plans.

Start the Administration CLI

Before running the Admin CLI and continuing further, you must have already completed all of the configuration steps described in Configuring your data store installation.

The `runadmin` utility provides the Admin command line interface (CLI). You can use the `runadmin` utility for a number of purposes. In this section, you use it to administer the Storage Nodes in your data store. First, you supply the details of the Storage Node and registry port that `runadmin` can use to connect to the data store.

If this is the first Storage Node you are connecting to that data store using the CLI, the Storage Node is designated as the one on which the master copy of the administration database resides.

> **Note:**
>
> You cannot change whatever Storage Node you use to initially configure the data store, such as `node01` in this example. Carefully plan the Storage Node to which `runadmin` first connects.

In the example below, you use `$KVHOST` to represent the network name of the Storage Node to which `runadmin` connects, and you use `5000` as the registry port.

One of the most important aspects of this Storage Node is that it must run the Storage Node Agent (SNA). All Storage Nodes should have an SNA running on them at this point. If any do not, complete the instructions in Installing Oracle NoSQL Database before proceeding further.

To start `runadmin` to use the Admin command line interface (CLI) for administration purposes, use these commands:

```
ssh node01
> java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar runadmin \
-host $KVHOST -port 5000 \
-security $KVROOT/security/client.security
```

With this `runadmin` example, you specify a single host and port (`-host node01 -port 5000`), permitting one Storage Node host to run an Admin process. The Admin process lets you run Admin CLI commands. If you want more than one Storage Node to support CLI commands, use the `runadmin` utility `-helper-hosts` flag and list two or more Storage Nodes and ports, rather than `-host <name> -port <value>`. For example, the next command starts an Admin process on three different Storage Nodes, which can then service CLI commands (`<host2>`, `<host3>`, and `<host4>`):

```
ssh node01
> java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar runadmin \
```

```
-helper-hosts <host2>:5000, <host3>:5000, <host4>:5000 \
-security $KVROOT/security/client.security
```

> **Note:**
>
> You need to complete the steps in Create users and configure security with remote access , to use the`runadmin` command to access the Admin node running on a Storage Node from any another Storage node.

After starting the Admin CLI, you can invoke the `help` command to describe all of the CLI commands.

You can collect the configuration steps that this section describes into a file, and then pass the script to the CLI utility using the `-script`option. See Create a script to configure the data store for more information.

Name your data store

When you start the Admin CLI , the `kv->` prompt appears. Once you see this, you can name your data store by using the `configure -name` command. The only information this command needs is the name of the data store that you want to configure.

Note that the name of your data store is essentially used to form a path to records kept in the data store. For this reason, you should avoid using characters in the data store name that might interfere with its use within a file path. The command line interface does not allow an invalid data store name. Valid characters are alphanumeric, '-', '_', and '.'.

For example:

```
kv-> configure -name mystore
Store configured: mystore
```

> **Note:**
>
> The data store name must be unique across all instances of NoSQL Database. For more information, see Store Targets .

Create a Zone

After starting the Admin command line interface (CLI) and naming your data store, you need to create at least one zone. It is possible, and even desirable, to create more than one zone. Because zones are complete copies of your data store, using multiple zones improves your data store's availability. This section describes an installation with a single zone. For more directions about creating a store deployment with multiple zones, see Configuring with Multiple Zones.

> **Note:**
>
> Once you add Storage Nodes to a zone, you cannot remove the zone from your data store.

To create a zone, use the `plan deploy-zone` with this usage:

```
plan deploy-zone -name <zone name>
-rf <replication factor>
[-type [primary | secondary]]
[-arbiters | -no-arbiters ]
[-json ]
[-master-affinity | -no-master-affinity]
[-plan-name <name>] [-wait] [-noexecute] [-force]
```

where:

- `-arbiters`

  Specifies that you can allocate Arbiter Nodes on the Storage Node in the zone.

- `-no-arbiters`

  Specifies that you cannot allocate Arbiter Nodes on the Storage Node in the zone. You can specify this flag only on a primary zone.

  > **Note:**
  >
  > Only primary zones can host Arbiter Nodes.

- `-rf`

  A number specifying the Zone Replication Factor. A primary zone can have a Replication Factor equal to zero.

- `-name`

  Identifies the zone name, as a string.

- `-json`

  Formats the command output in JSON.

  > **Note:**
  >
  > Only primary zones can host Arbiter Nodes.

- `-master-affinity`

  Indicates that this zone is a Master Affinity zone.

- `-no-master-affinity`

Specifies that this zone is not a Master Affinity zone.

- `-type`

  Specifies the type of zone to create. If you do not specify a –type, the `plan` utility creates a Primary zone.

For more information on Primary and Secondary Replication Factors, see Configuring with Multiple Zones.

When you execute the `plan deploy-zone` command, the CLI returns the plan number. It also returns instructions on how to check the plan's status, or to wait for it to complete. For example:

```
kv-> plan deploy-zone -name "Boston" -rf 1 -wait
Executed plan 1, waiting for completion...
Plan 1 ended successfully
```

You can show the plans and their status using the `show plans` command.

```
kv-> show plans
1 Deploy Zone (1)          SUCCEEDED
```

A zero Replication Factor zone is useful to host only Arbiter Nodes. You would add zero capacity Storage Nodes to this zone in order to host Arbiter Nodes. For more information see Deploying an Arbiter Node Enabled Topology.

You can also create Master Affinity Zones, which let you prioritize master nodes in primary zones. See Master Affinity Zones for details.

Create an Administration Process on a Specific Storage Node

Every data store has an administration database. The Admin CLI is currently connected to the Storage Node `node01`. Use the `deploy-sn` command to deploy the Storage Node `node01`. You then use the command `deploy-admin` to deploy an Administration process on the same Storage Node `node01` to continue configuring this data store.

The `deploy-admin` command creates an Administration process, with the same type as the Storage Node (SN) zone — if the zone is primary, the Admin is a primary Admin; if a secondary zone, so is the Admin.

Secondary Admins support failover. If a primary Admin fails, it converts to an offline secondary to re-establish quorum using existing Admins. A secondary Admin converts to a primary to take over for the failed primary. For more information on how quorum is applied, see the *Concepts Guide*.

To support failover, ensure that any zones used to continue data store operation after a failure contain at least one Admin node.

> **Note:**
>
> A deployed Admin must be the same type (`PRIMARY` or `SECONDARY`) as its zone. Also, the number of deployed Admins in a zone should be equal to the Zone Replication Factor.

The `deploy-sn` command requires a Zone ID. You can get this ID by using the `show topology` command:

```
kv-> show topology
store=mystore  numPartitions=0 sequence=1
zn: id=zn1 name=Boston repFactor=1 type=PRIMARY
allowArbiters=false masterAffinity=false
```

The zone ID is `zn1` in the output.

When you deploy the Storage Node, provide the zone ID, the node's network name, and its registry port number. For example:

```
kv-> plan deploy-sn -zn zn1 -host <hostname> -port 5000 -wait
Executed plan 2, waiting for completion...
Plan 2 ended successfully
```

Having deployed the Storage Node, create the Admin process on the Storage Node that you just deployed, using the `deploy-admin` command. This command requires the Storage Node ID (which you can obtain using the `show topology` command) and an optional plan name.

```
kv-> plan deploy-admin -sn sn1 -wait
Executed plan 3, waiting for completion...
Plan 3 ended successfully
```

Create a Storage Node Pool

Once you have created your Administration process, you can create a Storage Node Pool. This pool is used to contain all the Storage Nodes in your data store. A Storage Node pool is used for resource distribution when creating or modifying a data store. You use the `pool create` command to create this pool, then you join Storage Nodes to the pool using the `pool join` command.

Note that a default pool called **`AllStorageNodes`** will be created automatically and all SNs will be added to it during the topology deployment process. Therefore, the `pool` commands are optional if you use the `AllStorageNodes` pool as the default pool during deployment. You may have multiple kinds of Storage Nodes in different zones that vary by processor type, speed and/or disk capacity. So the Storage Node pool lets you define a logical grouping of Storage Nodes by whatever specification you pick.

> **Note:**
>
> This section is only to demonstrate how to explicitly create a Storage Node pool. Skip this section if you want to use the default pool `AllStorageNodes` during the topology deployment process.

Remember that you already have a Storage Node created. You did that in the previous step where you used the `deploy-sn` command to deploy the Storage Node. Therefore, after you add the pool, you can immediately join that first Storage Node to the pool.

The `pool create` command only requires you to provide the name of the pool.

The `pool join` command requires the name of the pool to which you want to join the Storage Node, and the Storage Node's ID. You can obtain the Storage Node's ID using the `show topology` command.

For example:

```
kv-> pool create -name BostonPool
Added pool BostonPool

kv-> show topology
store=mystore  numPartitions=0 sequence=2
zn: id=zn1 name=Boston repFactor=1 type=PRIMARY
allowArbiters=false masterAffinity=false
sn=[sn1] zn:[id=zn1 name=Boston] <hostname>:5000 capacity=1 RUNNING

kv-> pool join -name BostonPool -sn sn1
Added Storage Node(s) [sn1] to pool BostonPool
```

Create the Remainder of your Storage Nodes

This section is only applicable if you are configuring multiple Storage Nodes. Skip this section if you are configuring a single Storage Node.

Having created your Storage Node Pool, you can create the remainder of your Storage Nodes. Every Storage Node hosts various Oracle NoSQL Database admin and managed services in the data store. Consequently, you must use the `deploy-sn` command in the same way as you did in Create an Administration Process on a Specific Storage Nodeto add each new Storage Node to your data store. As you deploy each Storage Node, join it to your Storage Node Pool as described in the previous section.

Hint: Storage Node ID numbers increment sequentially with each Storage Node you add. So you do not have to repetitively look up the IDs with `show topology`. If the last Storage Node you created was assigned an ID of `10`, then the next Storage Node is automatically assigned ID `11`.

```
kv-> plan deploy-sn -zn zn1 -host <host2> -port 5000 -wait
Executed plan 4, waiting for completion...
Plan 4 ended successfully
kv-> pool join -name BostonPool -sn sn2
Added Storage Node(s) [sn2] to pool BostonPool
kv-> plan deploy-sn -zn zn1 -host <host3> -port 5000 -wait
```

```
Executed plan 5, waiting for completion...
Plan 5 ended successfully
kv-> pool join -name BostonPool -sn sn3
Added Storage Node(s) [sn3] to pool BostonPool
kv->
....
```

Repeat this process for all new Storage Nodes in your data store.

Create and Deploy Replication Nodes

The final step in your configuration process is to create replication nodes on every Storage Node in your data store. You do this using the `topology create` and `plan deploy-topology` commands. The `topology create` command takes the following arguments:

- *topology name* - A string to identify the topology.
- *pool name* - A string to identify the pool.
- *number of partitions*

The initial configuration is based on the number of Storage Nodes specified by the pool. This number is fixed once the topology is created and it cannot be changed. The command will automatically create an appropriate number of shards and replication nodes based upon the Storage Nodes in the pool.

You should make sure the number of partitions you select is more than the largest number of shards you ever expect your data store to contain, because the total number of partitions is static and cannot be changed. For simpler use cases, you can use the following formula to arrive at a very rough estimate for the number of partitions:

```
(Total number of disks hosted by the Storage Nodes  /
 Replication Factor) * 10
```

To get a more accurate estimate for production use, see Number of Partitions.

The `plan deploy-topology` command requires a topology name.

Once you issue the following commands, your data store is fully installed and configured:

```
kv-> topology create -name topo -pool BostonPool -partitions 300
Created: topo
kv-> plan deploy-topology -name topo -wait
Executed plan 6, waiting for completion...
Plan 6 ended successfully
```

> **Note:**
>
> If you have not created an explicit Storage pool , use `-pool AllStorageNodes` in the above command.

As a final sanity check, you can confirm that all of the plans succeeded using the `show plans` command:

```
kv-> show plans
1 Deploy Zone (1)          SUCCEEDED
2 Deploy Storage Node (2)  SUCCEEDED
3 Deploy Admin Service (3) SUCCEEDED
4 Deploy-RepNodes (4)      SUCCEEDED
```

You can then exit the command line interface.

```
kv-> exit
```

Smoke Testing the System

There are several things you can do to ensure that your data store is up and fully functional. You verify your data store using the `verify configuration` command in the CLI.

1. The `verify configuration` command inspects all the components of the data store. It also checks whether all store services are available. For the available store services, the command also checks for any version or metadata mismatches. The command requires no parameters, and runs in verbose mode, by default. For example:

```
kv-> verify configuration

Verify: starting verification of store mystore based upon topology
sequence #2
0 partitions and 1 storage nodes
Time: 2023-05-24 10:41:15 UTC   Version: 23.1.21
See <hostname>:$KVROOT/mystore/log/mystore_{0..N}.log for progress
messages
Verify: Shard Status: healthy: 0 writable-degraded: 0 read-only: 0
offline: 0 total: 0
Verify: Admin Status: healthy
Verify: Zone [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online: 0 read-only: 0 offline: 0
Verify: == checking storage node sn1 ==
Verify:         sn1: sn1 has 0 RepNodes and is under its capacity limit
of 1
Verify: Storage Node [sn1] on <hostname>: 5000
Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 23.1.21 2023-04-18 21:25:44 UTC  Build id:
477e7f102ab4
Edition: Enterprise    isMasterBalanced: unknown     serviceStartTime:
2023-05-24 10:37:28 UTC
Verify:         Admin [admin1]   Status: RUNNING,MASTER
serviceStartTime: 2023-05-24 10:38:21 UTC
stateChangeTime: 2023-05-24 10:38:21 UTC    availableStorageSize: 999 MB

Verification complete, 0 violations, 1 note found.
```

```
Verification note: [sn1]  sn1 has 0 RepNodes and is under its
capacity limit of 1
```

If the output shows all Storage Nodes and replication nodes as running without any errors, then the data store is configured well and all Storage Nodes are up and active.

2. Run the ping command as shown below:

```
kv-> ping

Pinging components of store mystore based upon topology sequence #2
0 partitions and 1 storage nodes
Time: 2023-05-24 11:36:06 UTC   Version: 23.1.21
Shard Status: healthy: 0 writable-degraded: 0 read-only: 0 offline:
0 total: 0
Admin Status: healthy
Zone [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online: 0 read-only: 0 offline: 0
Storage Node [sn1] on <hostname>: 5000
Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 23.1.21 2023-04-18 21:25:44 UTC  Build id:
477e7f102ab4
Edition: Enterprise  isMasterBalanced: unknown serviceStartTime:
2023-05-24 10:37:28 UTC
Admin [admin1]  Status: RUNNING,MASTER  serviceStartTime:
2023-05-24 10:38:21 UTC
stateChangeTime: 2023-05-24 10:38:21 UTC availableStorageSize: 999
MB
```

If the output shows all Storage Nodes and replication nodes as running without any errors, then the data store is configured well and all Storage Nodes are up and active.

If you run into installation problems or want to start over with a new data store, then on every Storage node in the data store:

1. Stop the Storage Node using:

```
java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar stop -root $KVROOT
```

2. Remove the contents of the KVROOT directory:

```
rm -rf $KVROOT
```

3. Start over with the steps described in Installation Configuration Parameters.

Create a script to configure the data store

> **Note:**
>
> You must follow the configuration steps as mentioned in Configuring your data store installation before running the Admin CLI.

You now know how to configure a data store using an interactive command line interface session. However, you can collect all of the commands used in the prior sections into a script file, and then run the script in a single batch operation. To do this, use the `load` command in the command line interface. For example:

Using the `load -file` command line option:

```
ssh node01
> java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar runadmin -port 5000 -host $KVHOST \
-security $KVROOT/security/client.security \
load -file script.txt
```

Using directly the `load -file` command. First start `runadmin` to use the Admin command line interface (CLI) for administration purposes:

```
 java -Xmx64m -Xms64m \
-jar $KVHOME/lib/kvstore.jar runadmin -port 5000 -host node01 \
-security $KVROOT/security/client.security
```

```
kv-> load -file <path to file>
```

Using this command you can load the named file and interpret its contents as a script of commands to be executed.

The file, `script.txt`, would contain content like the code snippet shown below. Note that the name of the store in this example is `mystore`.

```
### Begin Script ###
configure -name mystore
plan deploy-zone -name "Boston" -rf 3 -wait
plan deploy-sn -zn zn1 -host <hostname> -port 5000 -wait
plan deploy-admin -sn sn1 -wait
pool create -name BostonPool
pool join -name BostonPool -sn sn1
plan deploy-sn -zn zn1 -host <host2> -port 5000 -wait
pool join -name BostonPool -sn sn2
plan deploy-sn -zn zn1 -host <host3> -port 5000 -wait
pool join -name BostonPool -sn sn3
topology create -name topo -pool BostonPool -partitions 300
```

```
plan deploy-topology -name topo -wait
exit
### End Script ###
```

Troubleshooting

Typical errors when bringing up a data store are typos and misconfiguration. It is also possible to run into network port conflicts, especially if the deployment failed and you are starting over. Processes associated with a data store are reported by `jps -m` command. Some examples of them are :

- `kvstore.jar start -root $KVROOT` (SNA process)
- ManagedService

If you kill the SNA process it should also kill its managed processes.

There are detailed log files available in `$KVROOT/storename/log` as well as logs of the bootstrap process in `$KVROOT/*.log`. The bootstrap logs are most useful in diagnosing initial startup problems. The logs in `storename/log` appear once the data store has been configured. The logs on the Storage Node chosen for the admin process are the most detailed and include a store-wide consolidated log file: `$KVROOT/storename/log/storename_*.log`

Each line in the log file is prefixed with the date of the message, its severity, and the name of the component which issued it. For example:

```
2023-05-24 14:28:26.982 UTC INFO [admin1]
Initializing Admin for store: mystore
```

When looking for more context for events at a given time, use the timestamp and component name to narrow down the section of log to peruse.

Error messages in the logs show up with **SEVERE** in them so you can grep for that if you are troubleshooting. SEVERE error messages are also displayed in the CLI's `show events` command, and when you use the `ping` command.

In addition to log files, the log directories may also contain *.perf files, which are performance files for the replication nodes.

In general, `verify configuration` is the tool of choice for understanding the state of the data store. In addition to contacting the data store components, it will cross check each component's parameters against the Admin database. For example, `verify configuration` might report that a replication node's helperHosts parameter was at odds with the Admin. If this was the case then it might explain why a replication node cannot come up. The `Verify configuration` tool also checks on Admins. It also verifies the configuration of Arbiter Nodes in the topology.

Additionally, in order to catch configuration errors early, you can use the diagnostics tool when troubleshooting your data store. Also, you can use this tool to package important information and files to be able to send them to Oracle Support. For more information, see Diagnostics Utility.

Where to Find Error Information

As your data store operates, you can discover information about any problems that may be occurring by looking at the plan history and by looking at error logs.

The plan history indicates if any configuration or operational actions you attempted to take against the store encountered problems. This information is available as the plan executes and finishes. Errors are reported in the plan history each time an attempt to run the plan fails. The plan history can be seen using the CLI `show plan` command.

Other problems may occur asynchronously. You can learn about unexpected failures, service downtime, and performance issues through the CLI's `show events` command. Events come with a time stamp, and the description may contain enough information to diagnose the issue. In other cases, more context may be needed, and the administrator may want to see what else happened around that time.

The store-wide log consolidates logging output from all services. Browsing this file might give you a more complete view of activity during the problem period. It can be viewed using the CLI's `logtail` command, or by directly viewing the <storename>_N.log file in the $KVHOME/<storename>/log directory.

Service States

Oracle NoSQL Database uses four different types of services, all of which should be running correctly in order for your store to be in a healthy state. The four service types are the Admin, Storage Nodes, Replication Nodes and Arbiters Nodes. You should have multiple instances of these services running throughout your store.

Each service has a status that can be viewed using any of the following:

- The `show topology` command in the Administration CLI.
- Using the `ping` command.

The status values can be one of the following:

Name	Description
ERROR_NO_RESTART	The service is in an error state and is not automatically restarted. Administrative intervention is required.
ERROR_RESTARTING	The service is in an error state. Oracle NoSQL Database attempts to restart the service.
RUNNING	The service is running normally.
STARTING	The service is coming up.
STOPPED	The service was stopped intentionally and cleanly.
STOPPING	The service is stopping. This may take some time as some services can be involved in time-consuming activities when they are asked to stop.
SUCCEEDED	The plan has completed successfully.

Name	Description
UNREACHABLE	The service is not reachable by the Admin. If the status was seen using a command issued by the Admin, this state may mask a STOPPED or ERROR state. If an SN is UNREACHABLE, or an RN is having problems and its SN is UNREACHABLE, the first thing to check is the network connectivity between the Admin and the SN. However, if the managing SNA is reachable and the managed Replication Node is not, we can guess that the network is OK and the problem lies elsewhere.
WAITING_FOR_DEPLOY	The service is waiting for commands or acknowledgments from other services during its startup processing. If it is a Storage Node, it is waiting for the initial deploy-SN command. Other services should transition out of this phase without any administrative intervention from the user.

A healthy service begins with `STARTING`. It may transition to `WAITING_FOR_DEPLOY` for a short period before going on to `RUNNING`.

`ERROR_RESTARTING` and `ERROR_NO_RESTART` indicate that there has been a problem that should be investigated. An `UNREACHABLE` service may only be in that state temporarily, although if that state persists, the service may be truly in an `ERROR_RESTARTING` or `ERROR_NO_RESTART` state.

Useful Commands

The following commands may be useful to you when troubleshooting your KVStore.

- ```
  java -Xmx64m -Xms64m \
  -jar kvstore.tmp/kvstore.jar ping -host node01 -port 5000 \
  -security USER/security/admin.security
  ```

  Reports the status of the store running on the specified host and port. This command can be used against any of the host and port pairs used for Storage Nodes.

  > **Note:**
  >
  > This assumes that you have completed the steps in Create users and configure security with remote access .

- `jps -m`

  Reports the Java processes running on a machine. If the Oracle NoSQL Database processes are running, they are reported by this command.

- `ps -eaf | grep kv`

  You can view the list of kvstore processes that are running.
  ```

Configure data store - Advanced scenarios

- Create Additional Admin Processes
- Configuring with Multiple Zones
- Adding Secondary Zone to the Existing Topology

Create Additional Admin Processes

If you have deployed more than one Storage Node, you can add additional Admin processes using the `deploy-admin` plan. You are responsible for creating the appropriate number of Admins.

For example, currently you have a single Admin process deployed in your data store. So far, this has been sufficient to proceed with the data store configuration. However, to increase your data store's reliability, you should deploy multiple Admin processes, each running on a different Storage Node. This way, you can continue to administer your data store even if one Storage Node becomes unreachable and ends its Admin process. Having multiple Admin processes also means that you can continue to monitor your data store, even if you lose a Storage Node that is running an Admin process.

Create the Admin process on a Storage Node you just deployed, using the `plan deploy-admin` command. This command requires the Storage Node ID, which you can get from the `show topology` command:. Below is an example.

```
kv-> show topology
store=mystore  numPartitions=0 sequence=2
zn: id=zn1 name=Boston repFactor=1 type=PRIMARY allowArbiters=false
masterAffinity=false
sn=[sn1] zn:[id=zn1 name=Boston] <hostname>.oraclevcn.com:5000 capacity=1
RUNNING
numShards=0
kv-> plan deploy-admin -sn sn1 -wait
Executed plan 3, waiting for completion...
Plan 3 ended successfully
```

Although Admins are not required for normal data operations on the data store, they are needed to perform various administrative operations, including DDL operations. For example to create or modify tables, and for security operations involving users and roles. It is very important that the Admin services remain available.

Consideration for Admin Quorum
The full availability of the Admin service depends on having a quorum of the total Admin services available at a given time. Having a quorum of Admins operates similarly to the quorum for replication nodes in a shard. For replication nodes, the replication factor controls how many replication nodes can fail and still maintain the service. For example, with a replication factor of 3, the following table describes how failure numbers affect availability:

Failures	Availability
0	Full
1	Full

2	Read-only
3	None

The same failure and availability values exist for Admins. It is strongly recommended that you use the store replication factor to determine how many Admins should exist. This means that the Admin service has the same availability as the data store does for data operations. It is recommended that you use 3 Admins (matching the typical replication factor).

As with the store replication factor, when you use an even number of replicas, to maintain quorum (which is majority of the total number), you need more than half of the total replicas to be available. That means for a total of 4 replicas you need at least 2 replicas to be available to maintain quorum. For example, a replication factor of 4 has this behavior with failures and availability:

Failures	Availability
0	Full
1	Full
2	Read-only
3	Read-only
4	None

So, with a replication factor of 4, the group can still tolerate only a single failure and maintain full availability. Moreover, in addition to the higher Replication Factor value having no benefit during failures, now one more node exists that can fail, and the chance of losing quorum increases. The replication factor described here are for primary Storage Nodes associated with primary zones. For data stores with secondary zones, the nodes in the secondary zones are not included in the quorum.

Available Admins in Zones
Making sure that Admins are available in the right zones is another important consideration. If a data store has multiple primary zones, the zones were presumably set up to provide better availability. In this case, the admins should reflect the same arrangement. It is recommended that each zone has the same number of admins as the zone's replication factor. Unlike replication nodes, where all nodes in the shard can handle read operations, only the admin master responds to admin operations (unless there is no master). So, putting admins in a secondary zone is only useful to support failure recovery.

For example, if a store has primary and secondary zones, and all of the primary zones are lost, the administrator can use the `repair-admin-quorum` and `plan failover` commands to resume operations by converting the secondary zone to a primary zone. But these operations can occur only if an Admin node is available. For this reason, stores with secondary zones should include Admins in the secondary zones.

Configuring with Multiple Zones

To achieve optimal use of all available physical facilities, deploy your data store across multiple zones. Multiple zones provide fault isolation and availability for your data if a single zone fails. Each zone has a copy of your complete data store, including a copy of all the shards. With this configuration, reads are always possible, as long as your

data's consistency guarantees can be met, because at least one replica is located in every zone. Writes can also occur in the event of a zone loss, as long as the database maintains quorum. See *Concepts Guide*.

You can specify a different replication factor to each zone. A replication factor is quantified as one of the following:

Zone Replication Factor
The number of copies, or replicas, maintained in a zone.

Primary Replication Factor
The total number of replicas in all primary zones. This replication factor controls the number of replicas that participate in elections and acknowledgments. For additional information on how to identify the **Primary Replication Factor** and its implications, see Replication Factor.

Secondary Replication Factor
The total number of replicas in all secondary zones. Secondary replicas provide additional read-only copies of the data.

Store Replication Factor
The total number of replicas across the entire data store.

Zones that are located near each other physically benefit by avoiding bottlenecks from throughput limitations, and by reducing latency during elections and commits.

> **Note:**
>
> There are two types of zones: Primary, and Secondary.
>
> *Primary* zones contain nodes which can serve as masters or replicas. Zones are created as primary Zones by default. For good performance, primary zones should be connected by low latency networks so that they can participate efficiently in master elections and commit acknowledgments. Primary zones can also become Master Affinity zones.
>
> *Secondary* zones contain nodes which can only serve as replicas. Secondary zones can be used to provide low latency read access to data at a distant location, or to maintain an extra copy of the data to increase redundancy or increase read capacity. Because the nodes in secondary zones do not participate in master elections or commit acknowledgments, secondary zones can be connected to other zones by higher latency networks, because additional latency will not interfere with those time critical operations.

Using high throughput and low latency networks to connect primary zones leads to better results and improved performance. You can use networks with higher latency to connect to secondary zones so long as the connections provide sufficient throughput to support replication and sufficient reliability that temporary interruptions do not interfere with network throughput.

> **Note:**
>
> Because any primary zone can host master nodes, you can reduce write performance by connecting primary zones through a limited throughput or a high latency network link.

The following steps walk you through the process of deploying six Storage Nodes across three primary zones. You can then verify that each shard has a replica in every zone; service can be continued in the event of a zone failure. You will configure secure data store in all the six Storage Nodes. In the first Storage Node, security will be configured and the security directory and all files contained in it will be copied from the first Storage Node to other Storage Nodes to setup security.

Follow the steps below in the first Storage Node (`node01`):

- Execute the following command:

```
java -jar $KVHOME/lib/kvstore.jar makebootconfig \
-root $KVROOT \
-port 5000 \
-host $KVHOST \
-harange 5010,5020 \
-store-security configure \
-capacity 1 \
-storagedir ${KVDATA}/disk1 \
-storagedirsize 5500-MB \
```

- Start the Storage Node Agent:

```
java -jar $KVHOME/lib/kvstore.jar start -root $KVROOT &
```

- Create a zip file of all the security files created:

```
cd ;
zip -r $HOME/security.zip $KVROOT/security;
cd -
```

- Copy `$HOME/security.zip` from this node (`node01`) to the other five nodes.

Follow these steps in each of the other Storage Nodes (`node02`, `node03`, `node04`, `node05`, `node06`).

- Unzip the security files copied from the first Storage Node (`node01`).

```
cd;
unzip -o security.zip -d /;
cd -;
```

- Execute the following command:

```
java -jar $KVHOME/lib/kvstore.jar makebootconfig \
-root $KVROOT \
-port 5000 \
```

```
-host $KVHOST \
-harange 5010,5020 \
-store-security enable \
-capacity 1 \
-storagedir ${KVDATA}/disk1 \
-storagedirsize 5500-MB \
```

- Start the Storage Node Agent:

```
java -jar $KVHOME/lib/kvstore.jar start -root $KVROOT &
```

From the first Storage Node (`node01`) deploy your data store using the following steps:

- Start the Admin CLI. Here $KVHOST is `node01`.

```
java -Xmx64m -Xms64m \
-jar  $KVHOME/lib/kvstore.jar runadmin \
-port 5000 -host $KVHOST
-security $KVROOT/security/client.security
```

- Name your data store and deploy three primary zones:

```
configure -name MetroArea;
plan deploy-zone -name "Manhattan" -rf 1 -wait;
plan deploy-zone -name "JerseyCity" -rf 1 -wait;
plan deploy-zone -name "Queens" -rf 1 -wait;
```

- Deploy the first Storage Node with administration process in the Manhattan zone.

```
plan deploy-sn -znname Manhattan -host node01 -port 5000 -wait;
plan deploy-admin -sn sn1 -wait;
```

  Deploy a second Storage Node in Manhattan zone:

```
plan deploy-sn -znname Manhattan -host node02 -port 5000 -wait;
```

- Deploy the first Storage Node with administration process in the JerseyCity zone:.

```
plan deploy-sn -znname JerseyCity -host node03 -port 5000 -wait;
plan deploy-admin -sn sn3 -wait;
```

  Deploy a second Storage Node in JerseyCity zone:

```
plan deploy-sn -znname JerseyCity -host node04 -port 5000 -wait;
```

- Deploy the first Storage Node with administration process in the Queens zone:.

```
plan deploy-sn -znname Queens -host node05 -port 5000 -wait;
plan deploy-admin -sn sn5 -wait;
```

Deploy a second Storage Node in Queens zone:

```
plan deploy-sn -znname JerseyCity -host node06 -port 5000 -wait;
```

- Create and deploy a topology:

```
topology create -name Topo1 -pool AllStorageNodes -partitions 300;
plan deploy-topology -name Topo1 -wait;
```

- Follow the instructions mentioned in Create users and configure security with remote access to create the access for the users in the multiple zones.
- Check service status with the `show topology` command:

```
kv-> show topology
store=MetroArea  numPartitions=100 sequence=117
  zn: id=zn1 name=Manhattan repFactor=1 type=PRIMARY
   allowArbiters=false masterAffinity=false
  zn: id=zn2 name=JerseyCity repFactor=1 type=PRIMARY
   allowArbiters=false masterAffinity=false
  zn: id=zn3 name=Queens repFactor=1 type=PRIMARY
   allowArbiters=false masterAffinity=false

  sn=[sn1] zn:[id=zn1 name=Manhattan] node01:5000 capacity=1 RUNNING
    [rg1-rn1] RUNNING
          No performance info available
  sn=[sn2] zn:[id=zn1 name=Manhattan] node02:5000 capacity=1 RUNNING
    [rg2-rn1] RUNNING
          No performance info available
  sn=[sn3] zn:[id=zn2 name=JerseyCity] node03:5000 capacity=1
RUNNING
    [rg1-rn2] RUNNING
          No performance info available
  sn=[sn4] zn:[id=zn2 name=JerseyCity] node04:5000 capacity=1
RUNNING
    [rg2-rn2] RUNNING
          No performance info available
  sn=[sn5] zn:[id=zn3 name=Queens] node05:5000 capacity=1 RUNNING
    [rg1-rn3] RUNNING
          No performance info available
  sn=[sn6] zn:[id=zn3 name=Queens] node06:5000 capacity=1 RUNNING
    [rg2-rn3] RUNNING
          No performance info available

  numShards=2
  shard=[rg1] num partitions=50
    [rg1-rn1] sn=sn1
    [rg1-rn2] sn=sn3
    [rg1-rn3] sn=sn5
  shard=[rg2] num partitions=50
    [rg2-rn1] sn=sn2
    [rg2-rn2] sn=sn4
    [rg2-rn3] sn=sn6
```

- Verify that each shard has a replica in every zone:

```
kv-> verify configuration
Verify: starting verification of store MetroArea
based upon topology sequence #117
100 partitions and 6 storage nodes
Time: 2023-05-24 10:41:15 UTC   Version: 23.1.21
See node01:
$KVROOT/MetroArea/log/MetroArea_{0..N}.log
for progress messages
Verify: Shard Status: healthy:2
writable-degraded:0 read-only:0 offline:0 total:2
Verify: Admin Status: healthy
Verify: Zone [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]   RN Status: online:2 read-only:0 offline:0
Verify: Zone [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]   RN Status: online:2 read-only:0 offline:0
maxDelayMillis:1 maxCatchupTimeSecs:0
Verify: Zone [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]   RN Status: online:2 read-only:0 offline:0
maxDelayMillis:4 maxCatchupTimeSecs:0
Verify: == checking storage node sn1 ==
Verify: Storage Node [sn1] on node01:5000
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 22.3.21 2023-05-24 10:41:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Admin [admin1]        Status: RUNNING,MASTER
Verify:     Rep Node [rg1-rn1]    Status: RUNNING,MASTER
sequenceNumber:1,261 haPort:5011 available storage size:31 GB
Verify: == checking storage node sn2 ==
Verify: Storage Node [sn2] on node02:5000
Zone: [name=Manhattan id=zn1 type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 22.3.21 2023-05-24 10:41:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Rep Node [rg2-rn1]    Status: RUNNING,MASTER
sequenceNumber:1,236 haPort:5012 available storage size:31 GB
Verify: == checking storage node sn3 ==
Verify: Storage Node [sn3] on node03:5000
Zone: [name=JerseyCity id=zn2 type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 22.3.21 2023-05-24 10:41:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Admin [admin2]        Status: RUNNING,REPLICA
Verify:     Rep Node [rg1-rn2]    Status: RUNNING,REPLICA
sequenceNumber:1,261 haPort:5011 available storage size:31 GB
delayMillis:1 catchupTimeSecs:0
Verify: == checking storage node sn4 ==
Verify: Storage Node [sn4] on node04:5000
Zone: [name=JerseyCity id=zn2 type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 22.3.21 2023-05-24 10:41:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Rep Node [rg2-rn2]    Status: RUNNING,REPLICA
```

```
sequenceNumber:1,236 haPort:5012 available storage size:31 GB
delayMillis:0 catchupTimeSecs:0
Verify: == checking storage node sn5 ==
Verify: Storage Node [sn5] on node05:5000
Zone: [name=Queens id=zn3 type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 22.3.21 2023-05-24 10:41:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Admin [admin3]        Status: RUNNING,REPLICA
Verify:     Rep Node [rg1-rn3]    Status: RUNNING,REPLICA
sequenceNumber:1,261 haPort:5011 available storage size:31 GB
delayMillis:1 catchupTimeSecs:0
Verify: == checking storage node sn6 ==
Verify: Storage Node [sn6] on node06:5000
Zone: [name=Queens id=zn3 type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 22.3.21 2023-05-24 10:41:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Rep Node [rg2-rn3]    Status: RUNNING,REPLICA
sequenceNumber:1,236 haPort:5012 available storage size:31 GB
delayMillis:4 catchupTimeSecs:0

Verification complete, no violations.
```

In the above example there are three zones (zn1 = Manhattan, zn2 = JerseyCity, zn3=Queens) with six replication nodes (two masters and four replicas) in the data store. This means that this topology is not only highly available because you have three replicas within each shard, but it is also able to recover from a single zone failure. If any zone fails, the other two zones are enough to elect the new master, so service continues without any interruption.

Adding Secondary Zone to the Existing Topology

This section shows how to add a secondary zone to an existing topology that was created in Configuring with Multiple Zones . The following example adds a secondary zone in a different geographical location, Europe, allowing the users to read the data from the secondary zone either because it is physically located closer to the client or because the primary zone in the New York metro area is unavailable due to a disaster. The steps involve creating and starting two new Storage Nodes with capacity 1, creating a secondary zone, deploying the new Storage Nodes in the secondary zone, and doing a redistribute of the topology so that a replica for each shard is placed in the secondary zone.

Follow these steps in both the new Storage Nodes (`node07` and `node08`).

1. Copy the security zipped files from the first node and unzip the files.

   ```
   unzip -o security.zip -d /;
   ```

2. Invoke the `makebootconfig` utility for the first new Storage Node that will be deployed in the Frankfurt zone. The security configuration will be enabled while invoking the `makebootconfig` utility.

   ```
   java -jar $KVHOME/lib/kvstore.jar makebootconfig \
   -root $KVROOT \
   ```

```
-port 5000 \
-host $KVHOST \
-harange 5010,5020 \
-store-security enable \
-capacity 1 \
-storagedir ${KVDATA}/disk1 \
-storagedirsize 5500-MB
```

3. Start the Storage Node Agent.

```
java -jar $KVHOME/lib/kvstore.jar start -root $KVROOT &
```

To create a secondary zone and deploy the new Storage Nodes, do the following steps:

1. Start the Admin CLI. Here $KVHOST is `node01`.

```
java -Xmx64m -Xms64m \
-jar  $KVHOME/lib/kvstore.jar runadmin \
-port 5000 -host $KVHOST
-security $KVROOT/security/client.security
```

2. Create a secondary zone in Frankfurt.

```
kv-> plan deploy-zone -name Frankfurt -rf 1 -type secondary -wait
Executed plan 14, waiting for completion...
Plan 14 ended successfully
```

3. Deploy Storage Node sn7 in the Frankfurt zone.

```
kv-> plan deploy-sn -znname Frankfurt -host node07 -port 5000 -wait
Executed plan 15, waiting for completion...
Plan 15 ended successfully
```

4. Deploy the Storage Node sn7 with administration process in the Frankfurt zone.

```
kv-> plan deploy-admin -sn sn7 -wait
Executed plan 16, waiting for completion...
Plan 16 ended successfully
```

5. Deploy Storage Node sn8 in the Frankfurt zone.

```
kv-> plan deploy-sn -znname Frankfurt -host node08  -port 5000 -wait
Executed plan 17, waiting for completion...
Plan 17 ended successfully
```

6. Do redistribute and then deploy the new topology to create one replica for every shard in the secondary Frankfurt zone.

```
kv-> topology clone -current -name topo_secondary
Created topo_secondary

kv-> topology redistribute -name topo_secondary -pool AllStorageNodes
Redistributed: topo_secondary
```

```
kv-> topology preview -name topo_secondary
Topology transformation from current deployed topology to
topo_secondary:
Create 2 RN shard rg1 1 new RN : rg1-rn4
            shard rg2 1 new RN : rg2-rn4

kv-> plan deploy-topology -name topo_secondary -wait
Executed plan 19, waiting for completion...
Plan 19 ended successfully
```

7. Follow the instructions mentioned in Create users and configure security with remote access to copy user security files in the new Storage Nodes created.
8. Check service status with the show topology command.

```
kv-> show topology
store=MetroArea  numPartitions=100 sequence=120
  zn: id=zn1 name=Manhattan repFactor=1 type=PRIMARY
    allowArbiters=false masterAffinity=false
  zn: id=zn2 name=JerseyCity repFactor=1 type=PRIMARY
    allowArbiters=false masterAffinity=false
  zn: id=zn3 name=Queens repFactor=1 type=PRIMARY
    allowArbiters=false masterAffinity=false
  zn: id=zn4 name=Frankfurt repFactor=1 type=SECONDARY
    allowArbiters=false masterAffinity=false

  sn=[sn1] zn:[id=zn1 name=Manhattan] node01:5000 capacity=1 RUNNING
    [rg1-rn1] RUNNING
    single-op avg latency=0.21372496 ms multi-op avg latency=0.0 ms
  sn=[sn2] zn:[id=zn1 name=Manhattan] node02:5000 capacity=1 RUNNING
    [rg2-rn1] RUNNING
    single-op avg latency=0.30840763 ms multi-op avg latency=0.0 ms
  sn=[sn3] zn:[id=zn2 name=JerseyCity] node03:5000 capacity=1
RUNNING
    [rg1-rn2] RUNNING
    No performance info available
  sn=[sn4] zn:[id=zn2 name=JerseyCity] node04:5000 capacity=1
RUNNING
    [rg2-rn2] RUNNING
    No performance info available
  sn=[sn5] zn:[id=zn3 name=Queens] node05:5000 capacity=1 RUNNING
    [rg1-rn3] RUNNING
    No performance info available
  sn=[sn6] zn:[id=zn3 name=Queens] node06:5000 capacity=1 RUNNING
    [rg2-rn3] RUNNING
    No performance info available
  sn=[sn7] zn:[id=zn4 name=Frankfurt] node07:5000 capacity=1 RUNNING
    [rg1-rn4] RUNNING
    No performance info available
  sn=[sn8] zn:[id=zn4 name=Frankfurt] node07:5000 capacity=1
RUNNING
    [rg2-rn4] RUNNING
    No performance info available
```

```
numShards=2
shard=[rg1] num partitions=50
  [rg1-rn1] sn=sn1
  [rg1-rn2] sn=sn3
  [rg1-rn3] sn=sn5
  [rg1-rn4] sn=sn7
shard=[rg2] num partitions=50
  [rg2-rn1] sn=sn2
  [rg2-rn2] sn=sn4
  [rg2-rn3] sn=sn6
  [rg2-rn4] sn=sn8
```

9. Verify that the secondary zone has a replica for each shard.

```
kv-> verify configuration
Verify: starting verification of store MetroArea
based upon topology sequence #120
100 partitions and 7 storage nodes
Time: 2023-05-24 10:52:15 UTC   Version: 23.1.21
See node01:
$KVROOT/Disk1/MetroArea/log/MetroArea_{0..N}.log
for progress messages
Verify: Shard Status: healthy:2
writable-degraded:0 read-only:0 offline:0 total:2
Verify: Admin Status: healthy
Verify: Zone [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]   RN Status: online:2 read-only:0 offline:0
Verify: Zone [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]   RN Status: online:2 read-only:0 offline:0
maxDelayMillis:1 maxCatchupTimeSecs:0
Verify: Zone [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]   RN Status: online:2 read-only:0 offline:0
maxDelayMillis:1 maxCatchupTimeSecs:0
Verify: Zone [name=Frankfurt id=zn4 type=SECONDARY allowArbiters=false
masterAffinity=false]   RN Status: online:1 read-only:0 offline:0
maxDelayMillis:1 maxCatchupTimeSecs:0
Verify: == checking storage node sn1 ==
Verify: Storage Node [sn1] on node01:5000
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 23.1.21 2023-05-24 10:52:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Admin [admin1]        Status: RUNNING,MASTER
Verify:     Rep Node [rg1-rn1]    Status: RUNNING,MASTER
sequenceNumber:1,261 haPort:5011 available storage size:31 GB
Verify: == checking storage node sn2 ==
Verify: Storage Node [sn2] on node02:5000
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 23.1.21 2023-05-24 10:52:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Rep Node [rg2-rn1]    Status: RUNNING,MASTER
sequenceNumber:1,236 haPort:5012 available storage size:31 GB
Verify: == checking storage node sn3 ==
Verify: Storage Node [sn3] on node03:5000
```

```
Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 23.1.21 2023-05-24 10:52:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Admin [admin2]        Status: RUNNING,REPLICA
Verify:     Rep Node [rg1-rn2]    Status: RUNNING,REPLICA
sequenceNumber:1,261 haPort:5011 available storage size:31 GB
delayMillis:0 catchupTimeSecs:0
Verify: == checking storage node sn4 ==
Verify: Storage Node [sn4] on node04:5000
Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 23.1.21 2023-05-24 10:52:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Rep Node [rg2-rn2]    Status: RUNNING,REPLICA
sequenceNumber:1,236 haPort:5012 available storage size:31 GB
delayMillis:1 catchupTimeSecs:0
Verify: == checking storage node sn5 ==
Verify: Storage Node [sn5] on node05:5000
Zone: [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 23.1.21 2023-05-24 10:52:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Admin [admin3]        Status: RUNNING,REPLICA
Verify:     Rep Node [rg1-rn3]    Status: RUNNING,REPLICA
sequenceNumber:1,261 haPort:5011 available storage size:31 GB
delayMillis:1 catchupTimeSecs:0
Verify: == checking storage node sn6 ==
Verify: Storage Node [sn6] on node06:5000
Zone: [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING  Ver: 23.1.21 2023-05-24 10:52:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Rep Node [rg2-rn3]    Status: RUNNING,REPLICA
sequenceNumber:1,236 haPort:5012 available storage size:31 GB
delayMillis:0 catchupTimeSecs:0
Verify: == checking storage node sn7 ==
Verify: Storage Node [sn7] on node07:5000
Zone: [name=Frankfurt id=zn4 type=SECONDARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 23.1.21 2023-05-24 10:52:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify:     Admin [admin4]        Status: RUNNING,REPLICA
Verify:     Rep Node [rg1-rn4]    Status: RUNNING,REPLICA
sequenceNumber:1,261 haPort:5011 available storage size:31 GB
delayMillis:1 catchupTimeSecs:0
Verify: == checking storage node sn8 ==
Verify: Storage Node [sn8] on node08:5000
Zone: [name=Frankfurt id=zn4 type=SECONDARY allowArbiters=false
masterAffinity=false]
Status: RUNNING Ver: 23.1.21 2023-05-24 10:52:15 UTC
Build id: c8998e4a8aa5 Edition: Enterprise
Verify: Rep Node [rg2-rn4] Status: RUNNING,REPLICA
sequenceNumber:1,238 haPort:5012 available storage size:31 GB
delayMillis:0 catchupTimeSecs:0
```

```
Verification complete, no violations.
```

Oracle NoSQL Database Proxy

Learn how to set up Oracle NoSQL Database Proxy in Oracle NoSQL Database.

Topics:

- About the Oracle NoSQL Database Proxy
- Configuring the Proxy
- Using the Proxy in a non-secure data store
- Using the Proxy in a secure data store

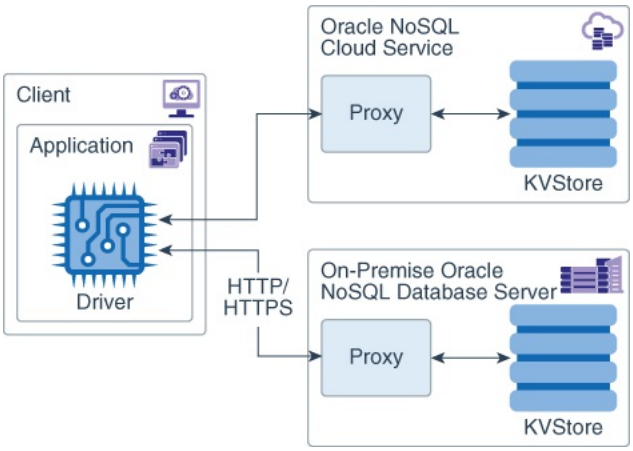
About the Oracle NoSQL Database Proxy

The Oracle NoSQL Database Proxy is a middle-tier component that lets the Oracle NoSQL Database drivers communicate with the Oracle NoSQL Database data store. The Oracle NoSQL Database drivers are available in various programming languages that are used in the client application. Currently, Java, Python, Go, Node.js, C# and Spring Data language drivers are supported.

The Oracle NoSQL Database Proxy is a server that accepts requests from Oracle NoSQL Database drivers and processes them using the Oracle NoSQL Database. The Oracle NoSQL Database drivers can be used to access either the Oracle NoSQL Database Cloud Service or an on-premises installation via the Oracle NoSQL Database Proxy. Since the drivers and APIs are identical, applications can be moved between these two options. However, an application connecting simultaneously to both the on-premises and Oracle NoSQL Database Cloud Service is not recommended.

For example, you can deploy a local Oracle NoSQL Database data store first for a prototype project, and move forward to Oracle NoSQL Database Cloud Service for a production project.

Figure 3-1 Oracle NoSQL Database Proxy and Driver



The JAR file for the Oracle NoSQL Database Proxy is included in the Enterprise Edition distribution and the Community Edition distribution of Oracle NoSQL Database. Users can

download the JAR for the Oracle NoSQL Database Proxy from the Oracle Technology Network.

Configuring the Proxy

The Oracle NoSQL Database Proxy should be configured after deploying a data store.

The following information should be obtained from the secure data store deployment:

- data store name. See ping.
- data store helper host:port list. See Obtaining a KVStore Handle in the *Java Direct Driver Developer's Guide*.

Proxy Parameters

The following parameters can be provided as the command line arguments to start up the proxy.

Parameter	Required ?	Default Value	Description
-helperHosts	Required		Helper hosts are hostname and port pairs that identify how to contact helper nodes within the data store. Use an array of strings to identify multiple helper hosts . Typically, you will get these hostname and port pairs from the data store's deployer or administrator. Example pattern is "hostname1:port1,hostname2:port2,...hostnameX:portX" Confirm that the ports in helper host list are left open by the firewall rules for connection between the proxy and data store server.
-storeName	Required		Name of the data store. This name is obtained from data store deployment process.
-hostname	No	localhost	The host name of the machine which is starting up the proxy instance.
-httpPort	No	80	The HTTP port of the proxy machine which will be used by the proxy to accept non-secure connections from HTTP requests. This parameter is mutually exclusive with the `-httpsPort` parameter. Only one of these parameters can be specified. Confirm that the port is left open by the firewall rules for connection between the proxy and the driver.
-httpsPort	No	443	The HTTPS port of the proxy machine which will be used by the proxy to accept secure connections from HTTPS requests. This parameter is mutually exclusive with the `-httpPort` parameter. Only one of these parameters can be specified. Confirm that the port is left open by the firewall rules for connection between the proxy and the driver.
-numAcceptThreads	No	3	This value determines the thread pool size for the threads that are used to handle the incoming connections to the proxy.
-numRequestThreads	No	32	This value determines the thread pool size for the threads that are used to handle the request input/output traffic, after the connection has been registered by the "AcceptThread" and handed over to the "RequestThread".

Parameter	Required ?	Default Value	Description
-verbose	No	false	Displays the proxy start-up information. Can take either "true" or "false" as values.
-sslCertificate	Required for secure proxy only.		Path to the SSL certificate file in pem file format. You can either generate a self-signed certificate using OpenSSL, or send a request to a public CA to generate a certificate. See Generating Certificate and Private Key for the Oracle NoSQL Database Proxy in the *Security Guide*.
-sslPrivateKey	Required for secure proxy only.		Path to the SSL private key file. You can either generate a private key using OpenSSL, or send a request to a public CA to generate a private key. See Generating Certificate and Private Key for the Oracle NoSQL Database Proxy in the *Security Guide*.
-sslPrivateKeyPass	Required for secure proxy only.		Password for the private key, if the private key is encrypted. This parameter is not required if the private key is not encrypted.
-storeSecurityFile	Required for secure proxy only.		Path to the security login file which is generated by the client user of the data store. The client user of the data store should be a non-admin proxy bootstrap user. To generate a login file, see Create users and configure security with remote access .

Note:

The Oracle NoSQL Database Proxy can run in one or multiple dedicated hosts. It can be hosted inside the nodes of the data store. You can use a load balancer as frontend which has a backend set of multiple NoSQL proxies on different hosts. When configuring a Load Balancer, you can add an HTTP health check. The Oracle NoSQL Database Proxy provides the following URI **/V2/health** endpoint to do it. An HTTP request to this URI will return a successful response 200 OK. You can find an example configuring HA proxy here: https://github.com/oracle/nosql-examples/tree/master/examples-nosql-cluster-deployment

Using the Proxy in a non-secure data store

Starting up the Proxy

Use the following command to start up the proxy for a non-secure data store.

```
java -jar lib/httpproxy.jar \
-storeName <kvstore_name> \
-helperHosts <kvstore_helper_host> \
[-hostname <proxy_host>] \
[-httpPort <proxy_http_port>]
```

where,

- `kvstore_name` is the data store name obtained from the data store deployment. See ping.
- `kvstore_helper_host` is the data store's helper host:port list obtained from the data store deployment. See Obtaining a KVStore Handle in the *Java Direct Driver Developer's Guide*.
- `proxy_host` is the hostname of the machine to host the proxy service. If the proxy is to be accessed from machines other than the one on which it is started this should be the hostname of the machine running the proxy. This parameter is optional and defaults to `localhost`.
- `proxy_http_port` is the port on which the proxy is watching for requests on its host machine. This is an optional parameter and defaults to 80.

> **Note:**
>
> Use of port 80 may require additional privileges, depending on your machine.

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

The Oracle NoSQL Database Java Driver contains the jar files that enable a Java application to communicate with the proxy.

Install the Java driver in the application's classpath and use the following code to connect to the proxy.

```
String endpoint = "http://<proxy_host>:<proxy_http_port>";
StoreAccessTokenProvider atProvider = new StoreAccessTokenProvider();
NoSQLHandleConfig config = new NoSQLHandleConfig(endpoint);
config.setAuthorizationProvider(atProvider);
NoSQLHandle handle = NoSQLHandleFactory.createNoSQLHandle(config);
```

where,

- `proxy_host` is the hostname of the machine to host the proxy service. This should match the host you configured earlier.
- `proxy_http_port` is the port on which the proxy is watching for requests on its host machine. This should match the http port you configured earlier.

Python

The onPremises configuration requires a running instance of the Oracle NoSQL database. In addition a running proxy service is required.

If the data store is not secure, an empty instance of `borneo.kv.StoreAccessTokenProvider` is used. For example:

```
from borneo import NoSQLHandle, NoSQLHandleConfig
from borneo.kv import StoreAccessTokenProvider
#
# Assume the proxy is running on localhost:8080
#
endpoint = 'http://localhost:8080'
#
# Create the AuthorizationProvider for a not secure store:
#
ap = StoreAccessTokenProvider()
#
# create a configuration object
#
config = NoSQLHandleConfig(endpoint).set_authorization_provider(ap)
#
# create a handle from the configuration object
#
handle = NoSQLHandle(config)
```

Go

The onPremises configuration requires a running instance of the Oracle NoSQL database. In addition a running proxy service is required. In this case, the `Endpoint` config parameter should point to the NoSQL proxy host and port location.

Use the following code to connect to the proxy.

```
...cfg:= nosqldb.Config{
    // EDIT: set desired endpoint for the Proxy server accordingly in your
environment.
    Endpoint: "http://localhost:8080",
    Mode:     "onprem",
}
client, err:=nosqldb.NewClient(cfg)
iferr!=nil {
    fmt.Printf("failed to create a NoSQL client: %v\n", err)
    return
}
deferclient.Close()
// Perform database operations using client APIs.// ...
```

Node.js

Your application will connect and use a running NoSQL database via the proxy service.

In non-secure mode, the driver communicates with the proxy via the HTTP protocol. The only information required is the communication `endpoint`. For on-premise NoSQL Database, the endpoint specifies the url of the proxy, in the form `http://proxy_host:proxy_http_port`

Use the following code to connect to the proxy.

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const ServiceType = require('oracle-nosqldb').ServiceType;
const client = new NoSQLClient({
    serviceType: ServiceType.KVSTORE,
    endpoint: 'myhost:8080'
});
```

You may also choose to store the same configuration in a file. Create file config.json with following contents:

```
{
    "serviceType": "KVSTORE",
    "endpoint": "myhost:8080",
}
```

Then you may use this file to create NoSQLClient instance:

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const client = new NoSQLClient('/path/to/config.json');
```

C#

Your application will connect and use a running NoSQL database via the proxy service.

In non-secure mode, the driver communicates with the proxy via the HTTP protocol. The only information required is the communication `endpoint`. For on-premise NoSQL Database, the endpoint specifies the url of the proxy, in the form `http://proxy_host:proxy_http_port`

To connect to the proxy in non-secure mode, you need to specify communication endpoint and the service type as `ServiceType.KVStore`. You can provide an instance of NoSQLConfig either directly or in a JSON configuration file.

```
var client = new NoSQLClient(
    new NoSQLConfig
    {
        ServiceType = ServiceType.KVStore,
        Endpoint = "myhost:8080"
    });
```

You may also choose to provide the same configuration in JSON configuration file. Create file config.json with following contents:

```
{
    "ServiceType": "KVStore",
```

```
    "Endpoint": "myhost:8080"
}
```

Then you may use this file to create NoSQLClient instance:

```
var client = new NoSQLClient("/path/to/config.json");
```

Spring Data

The Oracle NoSQL Database Spring Data SDK contains the files that enable a Spring Data application to communicate with the proxy.

Install the Java driver in the application's classpath. Use the following code to connect to the proxy.

The `configuration` Spring bean provides a `NosqlDbConfig` object. You can use the `StoreAccessTokenProvider` class to configure the Spring Data Framework to connect to an Oracle NoSQL Database store.

```
import com.oracle.nosql.spring.data.config.AbstractNosqlConfiguration;
import com.oracle.nosql.spring.data.config.NosqlDbConfig;
import
com.oracle.nosql.spring.data.repository.config.EnableNosqlRepositories;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import oracle.nosql.driver.kv.StoreAccessTokenProvider;

@Configuration

@EnableNosqlRepositories

public class AppConfig extends AbstractNosqlConfiguration {
    @Bean
    public NosqlDbConfig nosqlDbConfig() {
        AuthorizationProvider authorizationProvider;
        authorizationProvider = new StoreAccessTokenProvider();
        return new NosqlDbConfig("http://<proxy_host:proxy_http_port>",
authorizationProvider);
    }
}
```

where,

- `proxy_host` is the hostname of the machine to host the proxy service.
- `proxy_http_port` is the port on which the proxy is watching for requests on its host machine.

Using the Proxy in a secure data store

Starting up the Proxy

The Oracle NoSQL Database Proxy can be started on a secure data store using the following steps.

1. A secure proxy connection should be bootstrapped. Before you start up the proxy, you need to create a bootstrap user in the secure data store for the proxy to bootstrap its security connection. In SQL shell, the following command will create a bootstrap user for the proxy. See Developers Guide for getting started with SQL commands.

   ```
   sql-> CREATE USER <proxy_user> IDENTIFIED BY "<proxy_password>";
   ```

   where,

   - `proxy_user` is the user name.
   - `proxy_password` is the password for the `proxy_user`.

   > **Note:**
   >
   > The default privilege is sufficient for a bootstrap user. It is not recommended to grant admin privilege or any other additional privileges to the bootstrap user.

   > **Note:**
   >
   > Any user-supplied name can be given for the bootstrap user.

2. Create a directory `./proxy` where the proxy related files can be stored.
3. Create a login file `./proxy/proxy.login` for the bootstrap user with the following information in it.

   ```
   oracle.kv.auth.username=<proxy_user>
   oracle.kv.auth.pwdfile.file=proxy.passwd
   oracle.kv.transport=ssl
   oracle.kv.ssl.trustStore=client.trust
   ```

   where,

   - `proxy.passwd` is the file to store the password value of the `proxy_user` user.
   - `client.trust` is the certificate trust file obtained from the kvstore deployment.

   See Create users and configure security with remote access to know how to generate the `proxy.passwd` and `client.trust` files in kvstore client machine. In this case, the proxy runs as a kvstore client. These files must exist in order for the `proxy.login` to work properly.

4. Create a `certificate.pem` file and `key-pkcs8.pem` file for the proxy and driver to configure and establish a secure communication. If the Java driver is used, the `driver.trust` file should also be generated. See Generating Certificate and Private Key for the Oracle NoSQL Database Proxy in the *Security Guide*.
5. Use the following command to start up the proxy for a secure data store:

```
java -jar lib/httpproxy.jar \
-storeName <kvstore_name> \
-helperHosts <kvstore_helper_host> \
[-hostname <proxy_host>] \
[-httpsPort <proxy_https_port>] \
-storeSecurityFile proxy/proxy.login \
-sslCertificate certificate.pem \
-sslPrivateKey key-pkcs8.pem \
-sslPrivateKeyPass <privatekey_password> \
[-verbose true]
```

where,

- `kvstore_name` is the data store name obtained from the data store deployment. See ping.
- `kvstore_helper_host` is the data store's helper host:port list obtained from the data store deployment. See Obtaining a KVStore Handle in the *Java Direct Driver Developer's Guide*.
- `proxy_host` is the hostname of the machine to host the proxy service. If the proxy is to be accessed from machines other than the one on which it is started this should be the hostname of the machine running the proxy. This parameter is optional and defaults to `localhost` which means that the proxy will only be available from the machine running the proxy.
- `proxy_https_port` is the port on which the proxy is watching for requests on its host machine. This is an optional parameter and defaults to 443.

> **Note:**
>
> Use of port 80 may require additional privileges, depending on your machine.

- `proxy.login` is the security login file generated in the earlier step for accessing the secure kvstore.
- `certificate.pem` is the certificate file generated in the previous step.
- `key-pkcs8.pem` is the private key file generated in the previous step.
- `privatekey_password` is the password for the encrypted `key-pkcs8.pem` file.

> **Note:**
>
> The proxy start-up only accepts private key file in PKCS#8 format. If your private key is already in PKCS#8 (start with -----BEGIN ENCRYPTED PRIVATE KEY----- or -----BEGIN PRIVATE KEY-----), you don't need any additional conversion. Otherwise, you can use OpenSSL to do the conversion.

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

The Oracle NoSQL Database Java Driver contains the jar files that enables an application to communicate with the Oracle NoSQL Database Proxy. You can connect to the proxy using the following steps.

1. Create a user for the driver which is used by the application to access the data store through the proxy.

   ```
   sql-> CREATE USER <driver_user> IDENTIFIED BY "<driver_password>"
       sql-> GRANT READWRITE TO USER <driver_user>
   ```

   where, the `driver_user` is the username and `driver_password` is the password for the `driver_user` user. In this example, the user `driver_user` is granted `readwrite` role, which allows the application to perform only read and write operation. See Configuring Authorization in the *Security Guide*.

   > **Note:**
   >
   > If the user needs to create, drop, or alter tables or indexes, the `driver_user` should be granted `dbadmin` role, which allows the application to perform DDL operations.
   >
   > ```
   > sql-> GRANT DBADMIN TO USER <driver_user>
   > ```

2. Install the Oracle NoSQL Database Java Driver in the application's classpath and use the following code to connect to the proxy.

```
String endpoint = "https://<proxy_host>:<proxy_https_port>";
StoreAccessTokenProvider atProvider =
    new
StoreAccessTokenProvider("<driver_user>","<driver_password>".toCharArray()
);
NoSQLHandleConfig config = new NoSQLHandleConfig(endpoint);
config.setAuthorizationProvider(atProvider);
NoSQLHandle handle = NoSQLHandleFactory.createNoSQLHandle(config);
```

   where,

   - `proxy_host` is the hostname of the machine to host the proxy service. This should match the proxy host you configured earlier.
   - `proxy_https_port` is the port on which the proxy is watching for requests on its host machine. This should match the proxy https port configured earlier.
   - `driver_user` is the driver username. This should match the user created in the previous step.
   - `driver_password` is the password of the driver user.

3. Start-up the application program and set the `driver.trust` file's path to the Java trust store by using the following command. This is required as the proxy is already set up using the `certificate.pem` and `key-pkcs8.pem` files.

```
java -Djavax.net.ssl.trustStore=driver.trust \
-javax.net.ssl.trustStorePassword=<password of driver.trust> \
-cp .:lib/nosqldriver.jar application_program
```

   The `driver.trust` contains the `certificate.pem` or `rootCA.crt` certificate. If the certificate `certificate.pem` is in a chain signed by a trusted CA that is listed in `JAVA_HOME/jre/lib/security/cacerts`, then you don't need to append Java environment parameter `-Djavax.net.ssl.trustStore` in the Java command.

Python

The on-Premises configuration requires a running instance of the Oracle NoSQL database. In addition a running proxy service is required.

If running a secure store, a certificate path should be specified through the `REQUESTS_CA_BUNDLE` environment variable:

```
$ export REQUESTS_CA_BUNDLE=
<path-to-certificate>/certificate.pem:$REQUESTS_CA_BUNDLE
```

or `borneo.NoSQLHandleConfig.set_ssl_ca_certs()`.

In addition, a user identity must be created in the store (separately) that has permission to perform the required operations of the application, such as manipulated tables and data. The identity is used in the `borneo.kv.StoreAccessTokenProvider`.

```
from borneo import NoSQLHandle, NoSQLHandleConfig
from borneo.kv import StoreAccessTokenProvider
#
# Assume the proxy is secure and running on localhost:443
#
endpoint = 'https://localhost:443'
#
# Create the AuthorizationProvider for a secure store:
#
ap = StoreAccessTokenProvider(user_name, password)
#
# create a configuration object
#
config = NoSQLHandleConfig(endpoint).set_authorization_provider(ap)
#
# set the certificate path if running a secure store
#
config.set_ssl_ca_certs(<ca_certs>)
#
# create a handle from the configuration object
#
handle = NoSQLHandle(config)
```

Go

The on-Premises configuration requires a running instance of the Oracle NoSQL database. In addition a running proxy service is required. In this case, the `Endpoint` config parameter should point to the NoSQL proxy host and port location.

If running a secure store, a user identity must be created in the store (separately) that has permission to perform the required operations of the application, such as manipulating tables and data. If the secure server has installed a certificate that is self-signed or is not trusted by the default system CA, specify `InsecureSkipVerify` to instruct the client to skip verifying server's certificate, or specify the `CertPath` and `ServerName` that used to verify server's certificate and hostname.

```
import (
    "fmt"
    "github.com/oracle/nosql-go-sdk/nosqldb"
    "github.com/oracle/nosql-go-sdk/nosqldb/httputil"
)
...cfg:= nosqldb.Config{
    Endpoint: "https://nosql.mycompany.com:8080",
    Mode:     "onprem",
    Username: "testUser",
    Password: []byte("F;0s2M0;-Tdr"),
    // Specify InsecureSkipVerify
    HTTPConfig: httputil.HTTPConfig{
        InsecureSkipVerify: true,
    },
```

```
    // Alternatively, specify the CertPath and ServerName
    // HTTPConfig: httputil.HTTPConfig{
    //     CertPath: "/path/to/certificate-used-by-server",
    //     ServerName: "nosql.mycompany.com",
    // set to the "CN" subject value from the certificate
    // },
}
client, err:=nosqldb.NewClient(cfg)
iferr!=nil {
    fmt.Printf("failed to create a NoSQL client: %v\n", err)
    return
}
deferclient.Close()
// Perform database operations using client APIs.
// ...
```

Node.js

Your application will connect and use a running NoSQL database via the proxy service.

To connect to the proxy in secure mode, in addition to communication endpoint, you need to specify user name and password of the driver user. This information is passed in `Config#auth` object under `kvstore` property and can be specified in one of 3 ways as described below.

You may choose to specify user name and password directly:

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const client = new NoSQLClient({
    endpoint: 'https://myhost:8081',
    auth: {
        kvstore: {
            user: 'John',
            password: johnsPassword
        }
    }
});
```

This option is less secure because the password is stored in plain text in memory.

You may choose to store credentials in a separate file which is protected by file system permissions, thus making it more secure than previous option, because the credentials will not be stored in memory, but will be accessed from this file only when login is needed. Credentials file should have the following format:

```
{
    "user":     "<Driver user name>",
    "password": "<Driver user password>"
}
```

Then you may reference this credentials file as following:

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const client = new NoSQLClient({
    endpoint: 'https://myhost:8081',
    auth: {
        kvstore: {
            credentials: 'path/to/credentials.json'
        }
    }
});
```

You may also reference `credentials.json` in the configuration file used to create `NoSQLClient` instance.

```
{
    "endpoint": "https://myhost:8081",
    "auth": {
        "kvstore": {
            "credentials": "path/to/credentials.json"
        }
    }
}
```

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const client = new NoSQLClient('/path/to/config.json');
```

C#

Your application will connect and use a running NoSQL database via the proxy service.

To connect to the proxy in secure mode, in addition to communication endpoint, you need to specify user name and password of the driver user. This information is passed in the instance of `KVStoreAuthorizationProvider` and can be specified in any of the ways as described below.

You may choose to specify user name and password directly:

```
var client = new NoSQLClient(
    new NoSQLConfig
    {
        Endpoint = "https://myhost:8081",
        AuthorizationProvider = new KVStoreAuthorizationProvider(
            userName, // user name as string
            password) // password as char[]
    });
```

This option is less secure because the password is stored in plain text in memory for the lifetime of `NoSQLClient` instance. Note that the password is specified as char[] which allows you to erase it after you are finished using NoSQLClient.

You may choose to store credentials in a separate file which is protected by file system permissions, thus making it more secure than the previous option, because the credentials will not be stored in memory, but will be accessed from this file only when the login to the store is required. Credentials file should have the following format:

```
{
    "UserName": "<Driver user name>",
    "Password": "<Driver user password>"
}
```

Then you may use this credentials file as following:

```
var client = new NoSQLClient(
    new NoSQLConfig
    {
        Endpoint: 'https://myhost:8081',
        AuthorizationProvider = new KVStoreAuthorizationProvider(
            "path/to/credentials.json")
    });
```

You may also reference credentials.json in the JSON configuration file used to create `NoSQLClient` instance:

```
{
    "Endpoint": "https://myhost:8081",
    "AuthorizationProvider": {
        "AuthorizationType": "KVStore",
        "CredentialsFile": "path/to/credentials.json"
    }
}


var client = new NoSQLClient("/path/to/config.json");
```

Note that in `config.json` the authorization provider is represented as a JSON object with the properties for `KVStoreAuthorizationProvider` and an additional AuthorizationType property indicating the type of the authorization provider, which is KVStore for the secure on-premise store.

Spring Data

The Oracle NoSQL Database Spring Data SDK contains the files that enable a Spring Data application to communicate with the proxy.

Install the Java driver in the application's classpath. Use the following code to connect to the proxy.

The configuration Spring bean provides a `NosqlDbConfig` object. You can use the `StoreAccessTokenProvider` class with the username and password of the Oracle NoSQL

Database cluster. The `StoreAccessTokenProvider` class configures the Spring Data Framework to connect and authenticate to a secure Oracle NoSQL Database store.

```
import com.oracle.nosql.spring.data.config.AbstractNosqlConfiguration;
import com.oracle.nosql.spring.data.config.NosqlDbConfig;
import
com.oracle.nosql.spring.data.repository.config.EnableNosqlRepositories;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import oracle.nosql.driver.kv.StoreAccessTokenProvider;

@Configuration

@EnableNosqlRepositories

public class AppConfig extends AbstractNosqlConfiguration {
    @Bean
    public NosqlDbConfig nosqlDbConfig() {
        AuthorizationProvider authorizationProvider;
        authorizationProvider = new StoreAccessTokenProvider(user,
password);
        return new NosqlDbConfig("http://
<proxy_host:proxy_http_port>", authorizationProvider);
    }
}
```

where,

- `proxy_host` is the hostname of the machine to host the proxy service.
- `proxy_http_port` is the port on which the proxy is watching for requests on its host machine.

Oracle NoSQL Database Drivers

Learn about how to access the Oracle NoSQL Database using various Oracle NoSQL Database Drivers.

Topics:

- About Oracle NoSQL Database SDK drivers
- Obtaining a NoSQL handle
- Creating Regions
- Creating Tables and Indexes
- Adding Data
- Reading Data
- Using Queries

- Deleting Data
- Drop Table and Indexes
- Drop Regions
- Handling Errors

About Oracle NoSQL Database SDK drivers

Learn about Oracle NoSQL Database SDK drivers.

The Oracle NoSQL Database SDK Driver contains the files that enable an application to communicate with the on-premises or the Oracle NoSQL Database Cloud Service or the Oracle NoSQL Database Cloud Simulator.

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

The Oracle NoSQL Database SDK for Java is available in Maven Central repository, details available here. The main location of the project is in GitHub.

You can get all the required files for running the SDK with the following POM file dependencies.

> **Note:**
>
> The version changes with each release.

```
<dependency>
  <groupId>com.oracle.nosql.sdk</groupId>
  <artifactId>nosqldriver</artifactId>
  <version>5.2.31</version>
</dependency>
```

The Oracle NoSQL Database SDK for Java provides you with all the Java classes, methods, interfaces and examples. Documentation is available as javadoc in GitHub or from Java API Reference Guide.

Python

You can install the Python SDK through the Python Package Index with the command given below.

```
pip3 install borneo
```

The Oracle NoSQL SDK for Python provides you with all the Python classes, methods, interfaces and examples. Documentation is available in Python API Reference Guide.

Go

Open the Go Downloads page in a browser and click the download tab corresponding to your operating system. Save the file to your home folder.

Install Go in your operating system.

- On Windows systems, Open the MSI file you downloaded and follow the prompts to install Go.
- On Linux systems, Extract the archive you downloaded into `/usr/local`, creating a Go tree in `/usr/local/go`. Add `/usr/local/go/bin` to the PATH environment variable.

Access the online godoc for information on using the SDK and to reference Go driver packages, types, and methods.

Node.js

Download and install Node.js 12.0.0 or higher version from Node.js Downloads. Ensure that Node Package Manager (npm) is installed along with Node.js. Install the node SDK for Oracle NoSQL Database as shown below.

```
npm install oracle-nosqldb
```

Access the Node.js API Reference Guide to reference Node.js classes, events, and global objects.

C#

You can install the SDK from NuGet Package Manager either by adding it as a reference to your project or independently.

- Add the SDK as a Project Reference: You may add the SDK NuGet Package as a reference to your project by using .Net CLI.

  ```
  cd <your-project-directory>
  dotnet add package Oracle.NoSQL.SDK
  ```

  Alternatively, you may perform the same using NuGet Package Manager in Visual Studio.

- Independent Install: You may install the SDK independently into a directory of your choice by using nuget.exe CLI.

```
nuget.exe install Oracle.NoSQL.SDK -OutputDirectory
<your-packages-directory>
```

Spring Data

The Oracle NoSQL Database SDK for Spring Data is available in the Maven Central repository, details are available here. The main development location is the `oracle-spring-sdk` project on GitHub.

You can get all the required files for running the Spring Data Framework with the following POM file dependencies.

> **Note:**
>
> The version changes with each release.

```
<dependency>
    <groupId>com.oracle.nosql.sdk</groupId>
    <artifactId>spring-data-oracle-nosql</artifactId>
    <version>1.4.1</version>
</dependency>
```

Add the additional dependency to use the Spring Data Framework:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.7.0</version>
</dependency>
```

The Oracle NoSQL Database SDK for Spring Data provides you with all the Spring Data classes, methods, interfaces and examples. Documentation is available as nosql-spring-sdk in GitHub or from SDK for Spring Data API Reference.

Obtaining a NoSQL handle

Learn how to access tables using Oracle NoSQL Database Drivers. Start developing your application by creating a NoSQL Handle. Use the NoSQLHandle to access the tables and execute all operations.

- Java

- Python
- Go
- Node.js
- C#
- Spring Data

Java

In your application, create `NoSQLHandle` which will be your connection to the Oracle NoSQL Database Proxy. Using this `NoSQLHandle` you could access the Oracle NoSQL Database tables and execute Oracle NoSQL Database operations. To instantiate `NoSQLHandle`, pass a reference of `NoSQLHandleConfig` class to the `NoSQLHandleFactory.CreateNoSQLHandle` method. Provide the Oracle NoSQL Database Proxy URL as a parameterized constructor to instantiate the `NoSQLHandleConfig` class.

You could configure the proxy in the Oracle NoSQL Database server in either non-secure or secure mode. The `NoSQLHandleConfig` class allows an application to specify the security configuration information which is to be used by the handle. For non-secure access, create an instance of the `StoreAccessTokenProvider` class with the no-argument constructor. For secure access, create an instance of the `StoreAccessTokenProvider` class with the parameterized constructor. Provide the reference of `StoreAccessTokenProvider` class to the `NoSQLHandleConfig` class to establish the appropriate connection.

The following is an example of creating `NoSQLHandle` that connects to a non-secure proxy.

```
// Service URL of the proxy
String endpoint = "http://localhost:5000";

// Create a default StoreAccessTokenProvider for accessing the proxy
StoreAccessTokenProvider provider = new StoreAccessTokenProvider();

// Create a NoSQLHandleConfig
NoSQLHandleConfig config = new NoSQLHandleConfig(endpoint);

// Setup authorization provider using StoreAccessTokenProvider
config.setAuthorizationProvider(provider);

// Create NoSQLHandle using the information provided in the config
NoSQLHandle handle = NoSQLHandleFactory.createNoSQLHandle(config);
```

The following is an example of creating `NoSQLHandle` that connects to a secure proxy.

```
// Service URL of the secure proxy
String endpoint = "https://localhost:5000";

// Username of kvstore
String userName = "driver_user";
```

```
// Password of the driver user
String password = "DriverPass@@123";

//Construct StoreAccessTokenProvider with username and password
StoreAccessTokenProvider provider =
    new StoreAccessTokenProvider(userName, password.toCharArray());

// Create a NoSQLHandleConfig
NoSQLHandleConfig config = new NoSQLHandleConfig(endpoint);

// Setup authorization provider using StoreAccessTokenProvider
config.setAuthorizationProvider(provider);

// Create NoSQLHandle using the information provided in the config
NoSQLHandle handle = NoSQLHandleFactory.createNoSQLHandle(config);
```

For secure access, the `StoreAccessTokenProvider` parameterized constructor takes the following arguments.

- `username` is the username of the kvstore.
- `password` is the password of the kvstore user.

> **Note:**
>
> The client driver program should include a `driver.trust` file path in its JVM environment parameter `javax.net.ssl.trustStore` to make the secure connection work. The `driver.trust` should be distributed when the proxy is configured and started. This file is to allow the client driver to certify the proxy server's identity to make a secured connection.

User should generate `driver.trust` file for the java driver to access the secure proxy. See Generating Certificate and Private Key for the Oracle NoSQL Database Proxy in the *Security Guide*.

Following is an example of adding the `driver.trust` file to the client program:

```
java -Djavax.net.ssl.trustStore=driver.trust -cp .:/lib/nosqldriver.jar
Example
```

A handle has memory and network resources associated with it. Therefore, invoke the `NoSQLHandle.close` method to free up the resources when the application finishes using the handle.

To minimize network activity, and resource allocation and deallocation overheads, it's best to avoid repeated creation and closing of handles. For example, creating and closing a handle around each operation would incur large resource allocation overheads resulting in poor application performance. A handle permits concurrent operations, so a single handle is sufficient to access tables in a multi-threaded application. The creation of multiple handles incurs additional resource overheads without providing any performance benefit.

Python

A handle is created by first creating a `borneo.NoSQLHandleConfig` instance to configure the communication endpoint, authorization information, as well as default values for handle configuration. `borneo.NoSQLHandleConfig` represents a connection to the service. Once created it must be closed using the method `borneo.NoSQLHandle.close()` in order to clean up resources. Handles are thread-safe and intended to be shared.

If running a secure store, a certificate path should be specified through the `REQUESTS_CA_BUNDLE` environment variable:

```
$ export
    REQUESTS_CA_BUNDLE=<path-to-certificate>/
certificate.pem:$REQUESTS_CA_BUNDLE
```

In addition, a user identity must be created in the store (separately) that has permission to perform the required operations of the application, such as manipulated tables and data. The identity is used in the `borneo.kv.StoreAccessTokenProvider`. If the store is not secure, an empty instance of `borneo.kv.StoreAccessTokenProvider` is used.

An example of acquiring a NoSQL Handle for Oracle NoSQL Database:

```
from borneo import NoSQLHandle, NoSQLHandleConfig
from borneo.kv import StoreAccessTokenProvider
#
# Assume the proxy is running on localhost:8080
#
endpoint = 'http://localhost:8080'
#
# Assume the proxy is secure and running on localhost:443
#
endpoint = 'https://localhost:443'
#
# Create the AuthorizationProvider for a secure store:
#
ap = StoreAccessTokenProvider(user_name, password)
#
# Create the AuthorizationProvider for a not secure store:
#
ap = StoreAccessTokenProvider()
#
# create a configuration object
#
config = NoSQLHandleConfig(endpoint).set_authorization_provider(ap)

#
# set the certificate path if running a secure store
#
config.set_ssl_ca_certs(<ca_certs>)

#
# create a handle from the configuration object
```

```
#
handle = NoSQLHandle(config)
```

Go

Configure for non-secure on-premise Oracle NoSQL Database

To connect an application to a non-secure NoSQL database, specify the endpoint at which the Proxy server is running, and specify *onprem* as configuration mode.

```
cfg:= nosqldb.Config{
    Mode:     "onprem",
    Endpoint: "http://exampleHostServer:8080",
}
client, err:=nosqldb.NewClient(cfg)
...
```

Configure for secure on-premise Oracle NoSQL Database

To connect an application to a secure NoSQL database, you need to provide user credentials used to authenticate with the server. If the Proxy server is configured with a self-signed certificate or a certificate that is not trusted by the default system CA, you also need to specifiy *CertPath* and *ServerName* for the certificate path and server name used to verify server's certificate.

```
cfg:= nosqldb.Config{
    Mode:     "onprem",
    Endpoint: "https://exampleHostServer",
    Username: "driverUser",
    Password: []byte("ExamplePassword__123"),
    HTTPConfig: httputil.HTTPConfig{
        CertPath: "/path/to/server-certificate",
        ServerName: "exampleHostServer", // should match the CN subject
value from the certificate
    },
}
client, err:=nosqldb.NewClient(cfg)
```

Node.js

Class `NoSQLClient` represents the main access point to the service. To create instance of `NoSQLClient` you need to provide appropriate configuration information. This information is represented by a plain JavaScript object and may be provided to the constructor of `NoSQLClient` as the object literal. Alternatively, you may choose to store this information in a JSON configuration file and the constructor of `NoSQLClient` with the path (absolute or relative to the application's current directory) to that file.

Connecting to a Non-Secure Store

To connect to the proxy in non-secure mode, you need to specify communication endpoint.

For example, if using configuration JavaScript object:

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const ServiceType = require('oracle-nosqldb').ServiceType;
```

```
const client = new NoSQLClient({
    serviceType: ServiceType.KVSTORE,
    endpoint: 'myhost:8080'
});
```

You may also choose to store the same configuration in a file. Create file *config.json* with following contents:

```
{
    "serviceType": "KVSTORE",
    "endpoint": "myhost:8080",
}
```

Then you may use this file to create NoSQLClient instance:

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const client = new NoSQLClient('/path/to/config.json');
```

Connecting to a Secure Store

To connect to the proxy in secure mode, in addition to communication endpoint, you need to specify user name and password of the driver user. This information is passed in Config#auth object under kvstore property and can be specified in one of the 3 ways shown below.

- **Passing user name and password directly** You may choose to specify user name and password directly. This option is less secure because the password is stored in plain text in memory.

  ```
  const NoSQLClient = require('oracle-nosqldb').NoSQLClient;

  const client = new NoSQLClient({
      endpoint: 'https://myhost:8081',
      auth: {
          kvstore: {
              user: 'John',
              password: johnsPassword
          }
      }
  });
  ```

- **Storing credentials in a file:**

  You may choose to store credentials in a separate file which is protected by file system permissions. Credentials file should have the following format:

  ```
  {
      "user":     "<Driver user name>",
      "password": "<Driver user password>"
  }
  ```

Then you may reference this credentials file as following:

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;

const client = new NoSQLClient({
    endpoint: 'https://myhost:8081',
    auth: {
        kvstore: {
            credentials: 'path/to/credentials.json'
        }
    }
});
```

- **Creating Your Own KVStoreCredentialsProvider:**

  You may implement your own credentials provider for secure storage and retrieval of driver credentials as an instance of KVStoreCredentialsProvider interface. The loadCredentials returns a Promise and thus can be implemented as an async function.

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;

class MyKVStoreCredentialsProvider {
    constructor() {
        // Initialize required state information if needed
    }

    async loadCredentials() {
        // Obtain client id, client secret, user name and user password
somehow
        return { // Return credentials object as a result
            user: driverUserName,
            password: driverPassword
        };
    }
}
let client = new NoSQLClient({
    endpoint: 'https://myhost:8081',
    auth: {
        kvstore: {
            credentials: new MyKVStoreCredentialsProvider(myArgs...)
        }
    }
});
```

C#

Class `NoSQLClient` represents the main access point to the service. To create instance of `NoSQLClient` you need to provide appropriate configuration information. This information is represented by `NoSQLConfig` class which instance can be provided to the constructor of `NoSQLClient`. Alternatively, you may choose to store the configuration information in a JSON configuration file and use the constructor of `NoSQLClient` that takes the path (absolute or relative to current directory) to that file.

Connecting to a Non-Secure Store

In non-secure mode, the driver communicates with the proxy via the HTTP protocol. The only information required is the communication *endpoint*. For on-premise NoSQL Database, the endpoint specifies the url of the proxy, in the form *http://proxy_host:proxy_http_port*. To connect to the proxy in non-secure mode, you need to specify communication endpoint and the service type as `ServiceType.KVStore`.

You can provide an instance of NoSQLConfig either directly or in a JSON configuration file.

```
var client = new NoSQLClient(
    new NoSQLConfig
    {
        ServiceType = ServiceType.KVStore,
        Endpoint = "myhost:8080"
    });
```

You may also choose to provide the same configuration in JSON configuration file. Create file *config.json* with following contents:

```
{
    "ServiceType": "KVStore",
    "Endpoint": "myhost:8080"
}
```

Then you may use this file to create `NoSQLClient` instance:

```
varclient = new NoSQLClient("/path/to/config.json");
```

Connecting to a Secure Store :

To connect to the proxy in secure mode, in addition to communication endpoint, you need to specify user name and password of the driver user. This information is passed in the instance of `KVStoreAuthorizationProvider` and can be specified in one of the 3 ways shown below.

Passing user name and password directly

You may choose to specify user name and password directly. This option is less secure because the password is stored in plain text in memory for the lifetime of `NoSQLClient` instance. Note that the password is specified as *char[]* which allows you to erase it after you are finished using `NoSQLClient`.

You may choose to store credentials in a separate file which is protected by file system permissions. Credentials file should have the following format:

```
{
    "user":     "<Driver user name>",
    "password": "<Driver user password>"
}
```

Then you may use this credentials file as following:

```
var client = new NoSQLClient(
    new NoSQLConfig
```

```
    {
        Endpoint: 'https://myhost:8081',
        AuthorizationProvider=newKVStoreAuthorizationProvider(
            "path/to/credentials.json")
    });
```

You may implement your own credentials provider for secure storage and retrieval of driver credentials. This is the most secure option because you are in control of how the credentials are stored and loaded by the driver. The credentials provider is a delegate function that returns `Task<KVStoreCredentials>` and thus may be implemented asynchronously.

```
var client = new NoSQLClient(
    newNoSQLConfig
    {
        "Endpoint": "https://myhost:8081",
        AuthorizationProvider=newKVStoreAuthorizationProvider(
           async (CancellationToken) => {
                // Retrieve the credentials in a preferred
manner.await..........
                returnnewKVStoreCredentials(myUserName, myPassword);
           })
    });
```

Spring Data

Obtaining a NoSQL connection

In a Spring Data application, you must set up the `AppConfig` class that provides a `NosqlDbConfig` Spring bean. The `NosqlDbConfig` Spring bean describes how to connect to the Oracle NoSQL Database.

Create the `AppConfig` class that extends the `AbstractNosqlConfiguration` class. This exposes the connection and security parameters to the Oracle NoSQL Database SDK for Spring Data.

Return a `NosqlDbConfig` instance object with the connection details to the Oracle NoSQL Database Proxy. Provide the `@Configuration` and `@EnableNoSQLRepositories` annotations to this `NosqlDbConfig` class. The `@Configuration` annotation informs the Spring Data Framework that the `AppConfig` class is a configuration class that should be loaded before running the program. The `@EnableNoSQLRepositories` annotation informs the Spring Data Framework that it needs to load the program and look up for the repositories that extend the `NosqlRepository` interface. The `@Bean` annotation is required for the repositories to be instantiated.

Create a `nosqlDbConfig @Bean` annotated method to return an instance of the `NosqlDBConfig` class. The `NosqlDBConfig` instance object will be used by the Spring Data Framework to authenticate the Oracle NoSQL Database.

You can use the `StoreAccessTokenProvider` class to configure the Spring Data Framework to connect and authenticate with an Oracle NoSQL Database. You need to provide the URL of the Oracle NoSQL Database Proxy with non-secure access. For more details on

`StoreAccessTokenProvider` class, see StoreAccessTokenProvider in the *SDK for Spring Data API Reference*.

```
import com.oracle.nosql.spring.data.config.AbstractNosqlConfiguration;
import com.oracle.nosql.spring.data.config.NosqlDbConfig;
import
com.oracle.nosql.spring.data.repository.config.EnableNosqlRepositories;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import oracle.nosql.driver.kv.StoreAccessTokenProvider;

@Configuration

@EnableNosqlRepositories

public class AppConfig extends AbstractNosqlConfiguration {
    @Bean
    public NosqlDbConfig nosqlDbConfig() {
        AuthorizationProvider authorizationProvider;
        authorizationProvider = new StoreAccessTokenProvider();

        /* Provide the hostname and port number of the NoSQL cluster.*/
        return new NosqlDbConfig("http://<host:port>",
authorizationProvider);
    }
}
```

The following example modifies the previous example to connect to a secure Oracle NoSQL Database store.

```
import com.oracle.nosql.spring.data.config.AbstractNosqlConfiguration;
import com.oracle.nosql.spring.data.config.NosqlDbConfig;
import
com.oracle.nosql.spring.data.repository.config.EnableNosqlRepositories;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import oracle.nosql.driver.kv.StoreAccessTokenProvider;

@Configuration

@EnableNosqlRepositories
public class AppConfig extends AbstractNosqlConfiguration {

    @Bean
    public NosqlDbConfig nosqlDbConfig() {
        AuthorizationProvider authorizationProvider;
        /* Provide the username and password of the NoSQL cluster.*/
        authorizationProvider = new StoreAccessTokenProvider(user,
password);
```

```
        /* Provide the hostname and port number of the NoSQL cluster.*/
        return new NosqlDbConfig("http://<host:port>",
authorizationProvider);
    }
}
```

Creating Regions

Learn how to create a region in a Multi-Region Oracle NoSQL Database.

To create a region, use the `CREATE REGION` DDL statement. For example:

```
/* Create the region named us_east */
CREATE REGION us_east;
```

To drop a region from your application, you use the `KVStore` class to pass the DDL statement to the `executeSync` method. For example to create the `us_east` region:

```
KVStore.executeSync("CREATE REGION us_east");
```

Creating Tables and Indexes

Learn how to create tables and indexes.

Creating a table is the first step of developing your application.

Examples of DDL statements are:

```
/* Create a new table called users */
CREATE TABLE IF NOT EXISTS users(id INTEGER,
 name STRING,
 PRIMARY KEY(id))

/* Create a new table called users and set the TTL value to 4 days */
CREATE TABLE IF NOT EXISTS users(id INTEGER,
 name STRING,
 PRIMARY KEY(id))
USING TTL 4 days
/* Create a new multi-region table called users with two regions, and set
the TTL value to 4 days */
CREATE TABLE users(
 id INTEGER,
 name STRING,
 team STRING,
 primary key(id))
USING TTL 4 DAYS IN REGIONS fra, lnd
/* Create a new index called nameIdx on the name field in the users table */
CREATE INDEX IF NOT EXISTS nameIdx ON users(name)
```

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

Create a table and index using the `TableRequest` and its methods. The `TableRequest` class lets you pass a DDL statement to the `TableRequest.setStatement` method.

```
/* Create a simple table with an integer key and a single json data
 * field  and set your desired table capacity.
 * Set the table TTL value to 3 days.
 */
String createTableDDL = "CREATE TABLE IF NOT EXISTS users " +
 "(id INTEGER, name STRING, " +
 "PRIMARY KEY(id)) USING TTL 3 days";


TableRequest treq = new TableRequest().setStatement(createTableDDL);
// start the asynchronous operation
TableResult tres = handle.tableRequest(treq);


// The table request is asynchronous, so wait for the table to become
active.
TableResult.waitForState(handle, tres.getTableName(),
 TableResult.State.ACTIVE,
 60000, // wait for 60 sec
 1000); // delay in ms for poll

// Create an index called nameIdx on the name field in the users table.
treq = new TableRequest().setStatement("CREATE INDEX
  IF NOT EXISTS nameIdx ON users(name)
  ");

// start the asynchronous operation
  handle.tableRequest(treq);
```

Creating a child table: You use the API class and methods to execute DDL statement to create a child table. While creating a child table, Table limits need not be explicitly set as a child table inherits the limits of a parent table.

```
final static String tableName = "users";
final static String childtableName = "userDetails";
String createchildTableDDL = "CREATE TABLE IF NOT EXISTS " +
            tableName + "."+ childtableName + "(address STRING, salary
INTEGER, " +
```

```
            "PRIMARY KEY(address))";
    TableRequest treq = new TableRequest().setStatement(createchildTableDDL);
    System.out.println("Creating child table " + tableName);
    TableResult tres = handle.tableRequest(treq);
    /* The request is async,
    * so wait for the table to become active.
    */
    System.out.println("Waiting for "+ childtableName + " to become active");
    tres.waitForCompletion(handle, 60000, /* wait 60 sec */
         1000); /* delay ms for poll */
    System.out.println("Table " + childtableName + " is active");
```

Find the list of tables:

You can get a list of tables.

```
ListTablesRequest tablereq = new ListTablesRequest();
String [] tablelis = handle.listTables(tablereq).getTables();
if (tablelis.length == 0)
   System.out.println("No tables avaiable");
else {
   System.out.println("The tables available are");
   for (int i=0;i< tablelis.length; i++)  {
      System.out.println(tablelis[i]);
   }
}
```

You can also fetch the schema of a table at any time.

```
GetTableRequest gettblreq = new GetTableRequest();
gettblreq.setTableName(tableName);
System.out.println("The schema details for the table is "
+ handle.getTable(gettblreq).getSchema());
```

Python

DDL statements are executed using the `borneo.TableRequest` class. All calls to `borneo.NoSQLHandle.table_request()` are asynchronous so it is necessary to check the result and call `borneo.TableResult.wait_for_completion()` to wait for the operation to complete.

```
#Create a simple table with an integer key and a single
#json data field and set your desired table capacity.
#Set the table TTL value to 3 days.
from borneo import TableLimits,
TableRequest statement = 'create table if not exists users(id integer,
                                                 name string,
                                                 ' + 'primary key(id)
                                                    USING TTL 3 DAYS'

request = TableRequest().set_statement(statement)
# assume that a handle has been created, as handle, make the request
#wait for 60 seconds, polling every 1 seconds
```

```
result = handle.do_table_request(request, 60000, 1000)
# the above call to do_table_request is equivalent to
# result = handle.table_request(request)
result.wait_for_completion(handle, 60000, 1000)


#Create an index called nameIdx on the name field in the users table.
request = TableRequest().set_statement("CREATE INDEX IF NOT EXISTS
nameIdx
                                        ON users(name)")
# assume that a handle has been created, as handle, make the request
#wait for 60 seconds, polling every 1 seconds
result = handle.do_table_request(request, 60000, 1000)
# the above call to do_table_request is equivalent to
# result = handle.table_request(request)
result.wait_for_completion(handle, 60000, 1000)
```

Creating a child table: You use the API class and methods to execute DDL statement to create a child table. While creating a child table, Table limits need not be explicitly set as a child table inherits the limits of a parent table.

```
statement = 'create table if not exists users.userDetails (address
STRING,
salary integer,  primary key(address))'
    print('Creating table: ' + statement)
    request = TableRequest().set_statement(statement)
    # Ask the cloud service to create the table,
    # waiting for a total of 40000 milliseconds and polling the
service
    # every 3000 milliseconds to see if the table is active
    table_result = handle.do_table_request(request, 40000, 3000)
    table_result.wait_for_completion(handle, 40000, 3000)
    if (table_result.get_state() != State.ACTIVE):
        raise NameError('Table userDetails is in an unexpected state '
+
str(table_result.get_state()))
```

Find the list of tables:

You can get a list of tables.

```
ltr = ListTablesRequest()
list(str)= handle.list_tables(ltr).getTables()
if list(str).len() = 0
   print ("No tables available")
else
   print('The tables available are: ' + list(str))
```

You can also fetch the schema of a table at any time.

```
request = GetTableRequest().set_table_name(table_name)
result = handle.get_table(request)
print('The schema details for the table is: ' + result.get_schema())
```

Go

The following example creates a simple table with an integer key and a single STRING field. The create table request is asynchronous. You wait for the table creation to complete.

```
// Create a simple table with an integer key and a single
// json data field and set your desired table capacity.
// Set the table TTL value to 3 days.
tableName := "users"
stmt := fmt.Sprintf("CREATE TABLE IF NOT EXISTS %s "+
    "(id integer, name STRING, PRIMARY KEY(id) "+
    "USING TTL 3 DAYS)", tableName)

tableReq := &nosqldb.TableRequest{
    Statement: stmt, }
tableRes, err := client.DoTableRequest(tableReq)
if err != nil {
    fmt.Printf("cannot initiate CREATE TABLE request: %v\n", err)
    return
}
_, err = tableRes.WaitForCompletion(client, 60*time.Second, time.Second)
if err != nil {
    fmt.Printf("Error finishing CREATE TABLE request: %v\n", err)
    return
}
fmt.Println("Created table ", tableName)


//Create an index called nameIdx on the name field in the users table
stmt_ind := fmt.Sprintf("CREATE INDEX IF NOT EXISTS nameIdx ON users(name)")
tableReq := &nosqldb.TableRequest{Statement: stmt_ind}
tableRes, err := client.DoTableRequest(tableReq)
if err != nil {
    fmt.Printf("cannot initiate CREATE INDEX request: %v\n", err)
    return
}
_, err = tableRes.WaitForCompletion(client, 60*time.Second, time.Second)
if err != nil {
    fmt.Printf("Error finishing CREATE INDEX request: %v\n", err)
    return
}
fmt.Println("Created index nameIdx ")
```

Creating a child table: You use the API class and methods to execute DDL statement to create a child table. While creating a child table, Table limits need not be explicitly set as a child table inherits the limits of a parent table.

```
// Creates a simple child table with a string key and a single integer field.
    childtableName := "users.userDetails"
    stmt1 := fmt.Sprintf("CREATE TABLE IF NOT EXISTS %s ("+
    "address STRING, "+
    "salary INTEGER, "+
    "PRIMARY KEY(address))",
    childtableName)
    tableReq1 := &nosqldb.TableRequest{Statement: stmt1}
```

```
    tableRes1, err := client.DoTableRequest(tableReq1)
    if err != nil {
        fmt.Printf("cannot initiate CREATE TABLE request: %v\n", err)
        return
    }
    // The create table request is asynchronous, wait for table
creation to complete.
    _, err = tableRes1.WaitForCompletion(client, 60*time.Second,
time.Second)
    if err != nil {
        fmt.Printf("Error finishing CREATE TABLE request: %v\n", err)
        return
    }
    fmt.Println("Created table ", childtableName)
```

Find the list of tables:

You can get a list of tables.

```
req := &nosqldb.ListTablesRequest{Timeout: 3 * time.Second,}
res, err := client.ListTables(req)
if len(res.Tables)== 0{
   fmt.Printf("No tables in the given compartment"
   return
}
fmt.Printf("The tables in the given compartment are:\n" )
for i, table := range res.Tables {
   fmt.Printf(table)
}
```

You can also fetch the schema of a table at any time.

```
req := &nosqldb.GetTableRequest{
TableName: table_name, Timeout: 3 * time.Second, }
res, err := client.GetTable(req)
fmt.Printf("The schema details for the table is:state=%s,
   limits=%v\n", res.State,res.Limits)
```

Node.js

Table DDL statements are executed by `tableDDL` method. Like most other methods of `NoSQLClient` class, this method is asynchronous and it returns a Promise of `TableResult`. `TableResult` is a plain JavaScript object that contains status of DDL operation such as its TableState, name, schema and its TableLimit.

Note that `tableDDL` method only launches the specified DDL operation in the underlying store and does not wait for its completion. The resulting `TableResult` will most likely have one of intermediate table states such as `TableState.CREATING`, `TableState.DROPPING` or `TableState.UPDATING` (the latter happens when table is in the process of being altered by ALTER TABLE statement, table limits are being changed or one of its indexes is being created or dropped).

When the underlying operation completes, the table state should change to `TableState.ACTIVE` or `TableState.DROPPED` (the latter if the DDL operation was DROP TABLE).

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const TableState = require('oracle-nosqldb').TableState;
const client = new NoSQLClient('config.json');

async function createUsersTable() {
    try {
        const statement = 'CREATE TABLE IF NOT EXISTS users(id INTEGER, ' +
            'name STRING, PRIMARY KEY(id))';

let result = await client.tableDDL(statement);
        result = await client.forCompletion(result);
        console.log('Table users created');
    } catch(error) {
        //handle errors
    }
}
```

After the above call returns, result will reflect final state of the operation. Alternatively, to use complete option, substitute the code in try-catch block above with the following:

```
const statement = 'CREATE TABLE IF NOT EXISTS users(id INTEGER, ' +
        'name STRING, PRIMARY KEY(id))';

let result = await client.tableDDL(statement,
        complete: true
    });
    console.log('Table users created');


// Create an index called nameIdx on the name field in the users table.
try {
   const statement = 'CREATE INDEX IF NOT EXISTS nameIdx ON users(name))';
   let result = await client.tableDDL(statement);
   result = await client.forCompletion(result);
   console.log('Index nameIdx created');
} catch(error){
  //handle errors
}
```

Creating a child table: You use the API class and methods to execute DDL statement to create a child table. While creating a child table, Table limits need not be explicitly set as a child table inherits the limits of a parent table.

```
/**
  * This function will create the child table userDetails with two columns,
  * one string column address which will be the primary key and one integer
column
  * which will be the salary.
  * @param {NoSQLClient} handle An instance of NoSQLClient
```

```
  */
const TABLE_NAME = 'users';
const CHILDTABLE_NAME = 'userDetails';
async function createChildTable(handle) {
    const createChildtblDDL = `CREATE TABLE IF NOT EXISTS $
{TABLE_NAME}.${CHILDTABLE_NAME}
   (address STRING, salary INTEGER, PRIMARY KEY(address))`;
    console.log('Create table: ' + createChildtblDDL);
    let res =  await handle.tableDDL(createChildtblDDL, {complete:
true});
}
```

Find the list of tables:

You can get a list of tables.

```
let varListTablesResult = await client.listTables();
console.log("The tables in the given compartment are:")
{res.send(varListTablesResult)}
```

You can also fetch the schema of a table at any time.

```
let resExistingTab = await client.getTable(tablename);
{ await client.forCompletion(resExistingTab);}
console.log("The  schema details for the table is:")
{ res.send(resExistingTab.schema)}
```

C#

To create tables and execute other Data Definition Language (DDL) statements, such as creating, modifying and dropping tables as well as creating and dropping indexes, use methods `ExecuteTableDDLAsync` and `ExecuteTableDDLWithCompletionAsync`. Methods `ExecuteTableDDLAsync` and `ExecuteTableDDLWithCompletionAsync` return `Task<TableResult>`. TableResult instance contains status of DDL operation such as TableState and table schema. Each of these methods comes with several overloads. In particular, you may pass options for the DDL operation as `TableDDLOptions`.

Note that these are potentially long running operations. The method `ExecuteTableDDLAsync` only launches the specified DDL operation by the service and does not wait for its completion. You may asynchronously wait for table DDL operation completion by calling `WaitForCompletionAsync` on the returned TableResult instance.

```
var client = new NoSQLClient("config.json");
try {
  var statement = "CREATE TABLE IF NOT EXISTS users(id INTEGER,"
    + "name STRING, PRIMARY KEY(id))";

var result = await client.ExecuteTableDDLAsync(statement);
  await result.WaitForCompletionAsync();
  Console.WriteLine("Table users created.");
} catch(Exception ex) {
```

```
   // handle exceptions
}
```

Note that `WaitForCompletionAsync` will change the calling TableResult instance to reflect the operation completion.

Alternatively you may use `ExecuteTableDDLWithCompletionAsync`. Substitute the statements in the try-catch block with the following:

```
var statement = "CREATE TABLE IF NOT EXISTS users(id INTEGER,"
    + "name STRING, PRIMARY KEY(id))";


await client.ExecuteTableDDLWithCompletionAsync(statement);
Console.WriteLine("Table users created.");
```

Creating a child table: You use the API class and methods to execute DDL statement to create a child table. While creating a child table, Table limits need not be explicitly set as a child table inherits the limits of a parent table.

```
private const string TableName = "users";
private const string ChildTableName = "userDetails";
// Create a child table
var childtblsql = $"CREATE TABLE IF NOT EXISTS {TableName}.{ChildTableName}
address STRING, salary INTEGER, PRIMARY KEY(address))";
Console.WriteLine("\nCreate table {0}", ChildTableName);
var tableResult = await client.ExecuteTableDDLAsync(childtblsql);
Console.WriteLine("  Creating table {0}", ChildTableName);
Console.WriteLine("  Table state: {0}", tableResult.TableState);
// Wait for the operation completion
await tableResult.WaitForCompletionAsync();
Console.WriteLine("  Table {0} is created",tableResult.TableName);
Console.WriteLine("  Table state: {0}", tableResult.TableState);
```

Find the list of tables:

You can get a list of tables.

```
varresult = await client.ListTablesAsync();
console.WriteLine("The tables in the given compartment are:")
foreach(var tableName inresult.TableNames){
   Console.WriteLine(tableName);
}
```

Spring Data

In Spring data applications, the tables are automatically created at the beginning of the application when the entities are initialized unless `@NosqlTable.autoCreateTable` is set to `false`.

Create a `Users` entity class to persist. This entity class represents a table in the Oracle NoSQL Database and an instance of this entity corresponds to a row in that table.

Provide the `@NosqlId` annotation to indicate the ID field. The `generated=true` attribute specifies that the ID will be auto-generated. You can set the table level TTL by providing the `ttl()` and `ttlUnit()` parameters in the `@NosqlTable` annotation of the entity class. For details on all the Spring Data classes, methods, interfaces, and examples see SDK for Spring Data API Reference.

If the ID field type is a String, a UUID will be used. If the ID field type is int or long, a "GENERATED ALWAYS as IDENTITY (NO CYCLE)" sequence is used.

```
import com.oracle.nosql.spring.data.core.mapping.NosqlId;
import com.oracle.nosql.spring.data.core.mapping.NosqlTable;
/* The @NosqlTable annotation specifies that this class will be mapped
to an Oracle NoSQL Database table. */
/* Sets the table level TTL to 10 Days. */
@NosqlTable(ttl = 10, ttlUnit = NosqlTable.TtlUnit.DAYS)

public class Users {
    @NosqlId(generated = true)
    long id;
    String firstName;
    String lastName;

    /* public or package-protected constructor is required when
retrieving from the database. */
    public Users() {
    }

    @Override
    public String toString() {
        return "Users{" +
                "id=" + id + ", " +
                "firstName=" + firstName + ", " +
                "lastName=" + lastName +
                '}';
    }
}
```

Create the following `UsersRepository` interface. This interface extends the `NosqlRepository` interface and provides the entity class and the data type of the primary key in that class as parameterized types to the `NosqlRepository` interface. This `NosqlRepository` interface provides methods that are used to store or retrieve data from the database.

```
import com.oracle.nosql.spring.data.repository.NosqlRepository;

/* The Users is the entity class and Long is the data type of the
primary key in the Users class.
   This interface provides methods that return iterable instances of
the Users class. */

public interface UsersRepository extends NosqlRepository<Users, Long> {
    /* Search the Users table by the last name and return an iterable
instance of the Users class.*/
```

```
    Iterable<Users> findByLastName(String lastname);
}
```

You can use Spring's `CommandLineRunner` interface to show the application code that implements the run method and has the main method.

> **Note:**
>
> You can code the functionality as per your requirements by implementing any of the various interfaces that the Spring Data Framework provides. For more information on setting up a Spring boot application, see Spring Boot.

```
import com.oracle.nosql.spring.data.core.NosqlTemplate;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

/* The @SpringBootApplication annotation helps you to build an application
using Spring Data Framework rapidly.*/
@SpringBootApplication
public class App implements CommandLineRunner {

    /* The annotation enables Spring Data Framework to look up the
configuration file for a matching bean.*/
    @Autowired
    private UsersRepository repo;

    public static void main(String[] args) {
        ConfigurableApplicationContext ctx =
                SpringApplication.run(App.class, args);
        SpringApplication.exit(ctx, () -> 0);
        ctx.close();
        System.exit(0);
    }

    @Override
    public void run(String... args) throws Exception {
    }
}
```

When a table is created through the Spring Data application, a default schema is created automatically, which includes two columns - the primary key column (types String, int, long, or timestamp) and a JSON column called `kv_json_`.

> **Note:**
>
> If a table exists already, it must comply with the generated schema.

To create an index on a field in the `Users` table, you use `NosqlTemplate.runTableRequest()`.

Create the `AppConfig` class that extends `AbstractNosqlConfiguration` class to provide the connection details of the Oracle NoSQL Database. For details, see Obtaining a NoSQL connection.

In the application, you instantiate the `NosqlTemplate` class by providing the `NosqlTemplate` create (`NosqlDbConfig nosqlDBConfig`) method with the instance of the `AppConfig` class. You then modify the table using the `NosqlTemplate.runTableRequest()` method. You provide the NoSQL statement for the creation of the index in the `NosqlTemplate.runTableRequest()` method.

In this example, you create an index on the `lastName` field in the `Users` table.

```
import com.oracle.nosql.spring.data.core.NosqlTemplate;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

/* Create an Index on the lastName field of the Users Table. */

try {
    AppConfig config = new AppConfig();
    NosqlTemplate idx = NosqlTemplate.create(config.nosqlDbConfig());
    idx.runTableRequest("CREATE INDEX IF NOT EXISTS nameIdx ON
Users(kv_json_.lastName AS STRING)");
    System.out.println("Index created successfully");
} catch (Exception e) {
    System.out.println("Exception creating index" + e);
}
```

For more details on table creation, see Example: Accessing Oracle NoSQL Database Using Spring Data Framework in the *Spring Data SDK Developers Guide*.

Adding Data

Add rows to your table. When you store data in table rows, your application can easily retrieve, add to, or delete information from a table.

- Java
- Python
- Go
- Node.js

- C#
- Spring Data

Java

The `PutRequest` class represents the input to a `NoSQLHandle.put(oracle.nosql.driver.ops.PutRequest)` operation. This request can be used to perform unconditional and conditional puts to:

- Overwrite any existing row. Overwrite is the default functionality.
- Succeed only if the row does not exist. Use the `PutRequest.Option.IfAbsent` method in this case.
- Succeed only if the row exists. Use the `PutRequest.Option.IfPresent` method in this case.
- Succeed only if the row exists and the version matches a specific version. Use the `PutRequest.Option.IfVersion` method for this case and the `setMatchVersion(oracle.nosql.driver.Version)` method to specify the version to match.

> **Note:**
>
> First, connect your client driver to Oracle NoSQL Database to get a handle and then complete other steps. This topic omits the steps for connecting your client driver and creating a table.
> If you do not yet have a table, see Creating Tables and Indexes.

To add rows to your table:

```
/* use the MapValue class and input the contents of a new row */
MapValue value = new MapValue().put("id", 1).put("name", "myname");

/* create the PutRequest, setting the required value and table name */
PutRequest putRequest = new PutRequest().setValue(value)
                                        .setTableName("users");

/* use the handle to execute the PUT request
 * on success, PutResult.getVersion() returns a non-null value
 */
PutResult putRes = handle.put(putRequest);
if (putRes.getVersion() != null) {
  // success
} else {
 // failure
}
```

You can perform a sequence of `PutRequest` operations on a table that share the shard key using the `WriteMultipleRequest` class. If the operation is successful, the `WriteMultipleResult.getSuccess()` method returns true.

You can also add JSON data to your table. You can either convert JSON data into a record for a fixed schema table or you can insert JSON data into a column whose data type is of type JSON.

The `PutRequest` class also provides the `setValueFromJson` method which takes a JSON string and uses that to populate a row to insert into the table. The JSON string should specify field names that correspond to the table field names.

To add JSON data to your table:

```
/* Construct a simple row, specifying the values for each
 * field. The value for the row is this:
 *
 * {
 *   "cookie_id": 123,
 *   "audience_data": {
 *     "ipaddr": "10.0.00.xxx",
 *     "audience_segment": {
 *        "sports_lover": "2018-11-30",
 *        "book_reader": "2018-12-01"
 *      }
 *   }
 * }
 */
MapValue segments = new MapValue()
    .put("sports_lover", new TimestampValue("2018-11-30"))
    .put("book_reader", new TimestampValue("2018-12-01"));
MapValue value = new MapValue()
    .put("cookie_id", 123) // fill in cookie_id field
    .put("ipaddr", "10.0.00.xxx")
    .put("audience_segment", segments);
PutRequest putRequest = new PutRequest()
    .setValue(value)
    .setTableName(tableName);
PutResult putRes = handle.put(putRequest);
```

The same row can be inserted into the table as a JSON string:

```
/* Construct a simple row in JSON */
String jsonString = "{\"cookie_id\":123,\"ipaddr\":\"10.0.00.xxx\",
                      \"audience_segment\":
{\"sports_lover\":\"2018-11-30\",
                      \"book_reader\":\"2018-12-01\"}}";
PutRequest putRequest = new PutRequest()
      .setValueFromJson(jsonString, null) // no options
      .setTableName(tableName);
PutResult putRes = handle.put(putRequest);
```

Python

The `borneo.PutRequest` class represents input to the `borneo.NoSQLHandle.put()` method used to insert single rows. This method can be used for unconditional and conditional puts to:

- Overwrite any existing row. This is the default.

- Succeed only if the row does not exist. Use `borneo.PutOption.IF_ABSENT` for this case.
- Succeed only if the row exists. Use `borneo.PutOption.IF_PRESENT` for this case.
- Succeed only if the row exists and its borneo.Version matches a specific borneo.Version. Use `borneo.PutOption.IF_VERSION` for this case and `borneo.PutRequest.set_match_version()` to specify the version to match.

```
from borneo import PutRequest
# PutRequest requires a table name
request = PutRequest().set_table_name('users')
# set the value
request.set_value({'id': i, 'name': 'Jane'})
result = handle.put(request)
# a successful put returns a non-empty version
if result.get_version() is not None:
# success
```

When adding data the values supplied must accurately correspond to the schema for the table. If they do not, `IllegalArgumentException` is raised. Columns with default or nullable values can be left out without error, but it is recommended that values be provided for all columns to avoid unexpected defaults. By default, unexpected columns are ignored silently, and the value is put using the expected columns.

If you have multiple rows that share the same shard key they can be put in a single request using `borneo.WriteMultipleRequest` which can be created using a number of PutRequest or DeleteRequest objects. You can also add JSON data to your table. In the case of a fixed-schema table the JSON is converted to the target schema. JSON data can be directly inserted into a column of type *JSON*. The use of the JSON data type allows you to create table data without a fixed schema, allowing more flexible use of the data.

The data value provided for a row or key is a Python *dict*. It can be supplied to the relevant requests (GetRequest, PutRequest, DeleteRequest) in multiple ways:

- as a Python dict directly:

  ```
  request.set_value({'id': 1})
  request.set_key({'id': 1 })
  ```

- as a JSON string:

  ```
  request.set_value_from_json('{"id": 1, "name": "Jane"}')
  request.set_key_from_json('{"id": 1}')
  ```

In both cases the keys and values provided must accurately correspond to the schema of the table. If not an `borneo.IllegalArgumentException` exception is raised. If the data is provided as JSON and the JSON cannot be parsed a ValueError is raised.

Go

The `nosqldb.PutRequest` represents an input to the `nosqldb.Put()` function used to insert single rows. This function can be used for unconditional and conditional puts to:

- Overwrite any existing row. This is the default.
- Succeed only if the row does not exist. Specify `types.PutIfAbsent` for the `PutRequest.PutOption` field for this case.

- Succeed only if the row exists. Specify `types.PutIfPresent` for the `PutRequest.PutOption` field for this case.
- Succeed only if the row exists and its version matches a specific version. Specify `types.PutIfVersion` for the `PutRequest.PutOption` field and a desired version for the `PutRequest.MatchVersion` field for this case.

The data value provided for a row (in `PutRequest`) or key (in `GetRequest` and `DeleteRequest`) is a `*types.MapValue`. The `key` portion of each entry in the `MapValue` must match the column name of target table, and the `value` portion must be a valid value for the column. There are several ways to create a `MapValue` for the row to put into a table:

1. Create an empty `MapValue` and put values for each column.

```
value:=&types.MapValue{}
value.Put("id", 1).Put("name", "Jack")
req:=&nosqldb.PutRequest{
    TableName: "users",
    Value: value,
}
res, err:=client.Put(req)
```

2. Create a `MapValue` from a `map[string]interface{}`.

```
m:=map[string]interface{}{
    "id": 1,
    "name": "Jack",
}
value:=types.NewMapValue(m)
req:=&nosqldb.PutRequest{
    TableName: "users",
    Value: value,
}
res, err:=client.Put(req)
```

3. Create a `MapValue` from JSON. This is convenient for setting values for a row in the case of a fixed-schema table where the JSON is converted to the target schema. For example:

```
value, err:=types.NewMapValueFromJSON(`{"id": 1, "name": "Jack"}`)
iferr!=nil {
    return
}
req:=&nosqldb.PutRequest{
    TableName: "users",
    Value: value,
}
res, err:=client.Put(req)
```

JSON data can also be directly inserted into a column of type `JSON`. The use of the JSON data type allows you to create table data without a fixed schema, allowing more flexible use of the data.

Node.js

Method `put` is used to insert a single row into the table. It takes table name, row value as plain JavaScript object and opts as an optional 3rd argument. This method can be used for unconditional and conditional puts to:

- Overwrite existing row with the same primary key if present. This is the default.
- Succeed only if the row with the same primary key does not exist. Specify `ifAbsent` in the opt argument for this case: `{ ifAbsent: true }`. Alternatively, you may use `putIfAbsent` method.
- Succeed only if the row with the same primary key exists. Specify `ifPresent` in the opt argument for this case: `{ ifPresent: true }`. Alternatively, you may use `putIfPresent` method.
- Succeed only if the row with the same primary key exists and its Version matches a specific Version value. Set `matchVersion` in the opt argument for this case to the specific version: `{ matchVersion: my_version }`. Alternatively, you may use `putIfVersion` method and specify the version value as the 3rd argument (after table name and row).

Each put method returns a Promise of `PutResult` which is a plain JavaScript object containing information such as success status and resulting row Version. Note that the property names in the provided row object should be the same as underlying table column names.

To add rows to your table:

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const client = new NoSQLClient('config.json');

async function putRowsIntoUsersTable() {
    const tableName = 'users';
    try {
        // Uncondintional put, should succeed
        let result = await client.put(tableName, { id: 1, name: 'John' });

        // Will fail since the row with the same primary key exists
        result = await client.putIfAbsent(tableName, { id: 1, name:
'Jane' });
        // Expected output: putIfAbsent failed
        console.log('putIfAbsent ' + result.success ? 'succeeded' :
'failed');

        // Will succeed because the row with the same primary key exists
        res = await client.putIfPresent(tableName, { id: 1 , name: 'Jane' });
        // Expected output: putIfAbsent succeeded
        console.log('putIfPresent ' + result.success ?
            'succeeded' : 'failed');

        let version = result.version;
        // Will succeed because the version matches existing row
        result = await client.putIfVersion(tableName, { id: 1, name: 'Kim' },
            version);
        // Expected output: putIfVersion succeeded
        console.log('putIfVersion ' + result.success ? 'succeeded' :
'failed');
```

```
        // Will fail because the previous put has changed the row
version, so
        // the old version no longer matches.
        result = await client.putIfVersion(tableName, { id: 1, name:
'June' },
            version);
        // Expected output: putIfVersion failed
        console.log('putIfVersion ' + result.success ? 'succeeded' :
'failed');

    } catch(error) {
        //handle errors
    }
}
```

Note that `success` results in false value only if conditional put operation fails due to condition not being satisfied (e.g. row exists for `putIfAbsent`, row doesn't exist for `putIfPresent` or version doesn't match for `putIfVersion`). If put operation fails for any other reason, the resulting Promise will reject with an error (which you can catch in async function). For example, this may happen if a column value supplied is of a wrong type, in which case the put will result in `NoSQLArgumentError`.

You can perform a sequence of put operations on a table that share the same shard key using `putMany` method. This sequence will be executed within the scope of single transaction, thus making this operation atomic. The result of this operation is a Promise of `WriteMultipleResult`. You can also use `writeMany` if the sequence includes both puts and deletes.

Using columns of type *JSON* allows more flexibility in the use of data as the data in the JSON column does not have predefined schema. To put data into JSON column, provide either plain JavaScript object or a JSON string as the column value. Note that the data in plain JavaScript object must be of supported JSON types.

C#

Method `PutAsync` and related methods `PutIfAbsentAsync`, `PutIfPresentAsync` and `PutIfVersionAsync` are used to insert a single row into the table or update a single row.

These methods can be used for unconditional and conditional puts:

- Use `PutAsync` (without conditional options) to insert a new row or overwrite existing row with the same primary key if present. This is unconditional put.
- Use `PutIfAbsentAsync` to insert a new row only if the row with the same primary key does not exist.
- Use `PutIfPresentAsync` to overwrite existing row only if the row with the same primary key exists.
- Use `PutIfVersionAsync` to overwrite existing row only if the row with the same primary key exists and its RowVersion matches a specific version.

Each of the `Put` methods above returns `Task<PutResult<RecordValue>>`. `PutResult` instance contains info about a completed `Put` operation, such as success status

(conditional put operations may fail if the corresponding condition was not met) and the resulting RowVersion.

To add rows to your table:

```
var client = new NoSQLClient("config.json");
var tableName = "users";

try {
    // Uncondintional put, should succeed.
    var result = await client.PutAsync(tableName,
        new MapValue
        {
            ["id"] = 1,
            ["name"] = "John"
        });

    // This Put will fail because the row with the same primary
    // key already exists.
    result = await client.PutIfAbsentAsync(tableName,
        new MapValue
        {
            ["id"] = 1,
            ["name"] = "Jane"
        });

    // Expected output: PutIfAbsentAsync failed.
    Console.WriteLine("PutIfAbsentAsync {0}.",
        result.Success ? "succeeded" : "failed");

    // This Put will succeed because the row with the same primary
    // key exists.
    result = await client.PutIfPresentAsync(tableName,
        new MapValue
        {
            ["id"] = 1,
            ["name"] = "Jane"
        });

    // Expected output: PutIfPresentAsync succeeded.
    Console.WriteLine("PutIfPresentAsync {0}.",
        result.Success ? "succeeded" : "failed");
    var rowVersion = result.Version;

    // This Put will succeed because the version matches existing
    // row.
    result = await client.PutIfVersionAsync(
        tableName,
        new MapValue
        {
            ["id"] = 1,
            ["name"] = "Kim"
        }),
        rowVersion);
```

```
    // Expected output: PutIfVersionAsync succeeded.
    Console.WriteLine("PutIfVersionAsync {0}.",
        result.Success ? "succeeded" : "failed");

    // This Put will fail because the previous Put has changed
    // the row version, so the old version no longer matches.
    result = await client.PutIfVersionAsync(
        tableName,
        new MapValue
        {
            ["id"] = 1,
            ["name"] = "June"
        }),
        rowVersion);

    // Expected output: PutIfVersionAsync failed.
    Console.WriteLine("PutIfVersionAsync {0}.",
        result.Success ? "succeeded" : "failed");

    // Put a new row with TTL indicating expiration in 30 days.
    result = await client.PutAsync(tableName,
        new MapValue
        {
            ["id"] = 2,
            ["name"] = "Jack"
        }),
        new PutOptions
        {
            TTL = TimeToLive.OfDays(30)
        });
}
catch(Exception ex) {
    // handle exceptions
}
```

Note that Success property of the result only indicates successful completion as related to conditional Put operations and is always true for unconditional Puts. If the Put operation fails for any other reason, an exception will be thrown.

You can perform a sequence of put operations on a table that share the same shard key using `PutManyAsync` method. This sequence will be executed within the scope of single transaction, thus making this operation atomic. You can also call `WriteManyAsync` to perform a sequence that includes both Put and Delete operations.

Using fields of data type JSON allows more flexibility in the use of data as the data in JSON field does not have a predefined schema. To put value into a JSON field, supply a `MapValue` instance as its field value as part of the row value. You may also create its value from a JSON string via `FieldValue.FromJsonString`.

Spring Data

Use one of these methods to add rows to the table - `NosqlRepository save(entity_object)`, `saveAll(Iterable<T> iterable)`, or `NosqlTemplate insert(entity)`. For details, see SDK for Spring Data API Reference.

In this section, you use the `repository.save(entity_object)` method to add the rows.

> **Note:**
>
> First, create the `AppConfig` class that extends `AbstractNosqlConfiguration` class to provide the connection details of the Oracle NoSQL Database. For more details, see Obtaining a NoSQL connection.

To add rows to your table, you can include the following code in your application.

```
@Override
public void run(String...args) throws Exception {

    /* Create a new User instance and load values into it.*/

    Users u1 = new Users();
    u1.firstName = "John";
    u1.lastName = "Doe";

    /* Save the User instance.*/
    repo.save(u1);

    /* Create a second User instance and load values into it. Save the
instance.*/
    Users u2 = new Users();
    u2.firstName = "Angela";
    u2.lastName = "Willard";

    repo.save(u2);
}
```

This creates and saves two user entities. For each entity, the Spring Data SDK creates two columns:

1. Primary key column
2. JSON data type column

Here, the primary key is auto-generated. The `@NosqlId` annotation in the `Users` class specifies that the id field will act as the ID and be the primary key of the underlying storage table.

The `generated=true` attribute specifies that this ID will be auto-generated by a sequence. The rest of the entity fields, that is, the `firstName` and `lastName` fields are stored in the JSON column.

Reading Data

Learn how to read data from your table.

You can read data from your application by using the different API methods for the language-specific drivers. You can retrieve a record based on a single primary key value, or by using queries.

> **Note:**
>
> First, connect your client driver to Oracle NoSQL Database to get a connection and then complete other steps. This topic omits the steps for connecting your client driver and creating a table.

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

The `GetRequest` class provides a simple and powerful way to read data, while queries can be used for more complex read requests. To read data from a table, specify the target table and target key using the `GetRequest` class and use `NoSQLHandle.get()` to execute your request. The result of the operation is available in `GetResult`.

To read data from your table:

```
/* GET the row, first create the row key */
MapValue key = new MapValue().put("id", 1);
GetRequest getRequest = new GetRequest().setKey(key)
                                        .setTableName("users");
GetResult getRes = handle.get(getRequest);

/* on success, GetResult.getValue() returns a non-null value */
if (getRes.getValue() != null) {
  // success
} else {
  // failure
}
```

> **Note:**
>
> By default, all read operations are eventually consistent. You can change the default Consistency for a NoSQLHandle instance by using the `NoSQLHandleConfig.setConsistency(oracle.nosql.driver.Consistency)` and `GetRequest.setConsistency()` methods.

See the Java API Reference Guide for more information about the `GET` APIs.

Python

Learn how to read data from your table. You can read single rows using the `borneo.NoSQLHandle.get()` method. This method allows you to retrieve a record based on its primary key value. The `borneo.GetRequest` class is used for simple get operations. It contains the primary key value for the target row and returns an instance of `borneo.GetResult`.

```
from borneo import GetRequest
# GetRequest requires a table name
request = GetRequest().set_table_name('users')
# set the primary key to use request.set_key({'id': 1})
result = handle.get(request)
# on success the value is not empty
if result.get_value() is not None:
# success
```

By default all read operations are eventually consistent, using `borneo.Consistency.EVENTUAL`. This type of read is less costly than those using absolute consistency, `borneo.Consistency.ABSOLUTE`. This default can be changed in `borneo.NoSQLHandle` using `borneo.NoSQLHandleConfig.set_consistency()` before creating the handle. It can be changed for a single request using `borneo.GetRequest.set_consistency()`.

Go

You can read single rows using the `Client.Get` function. This function allows you to retrieve a record based on its primary key value. The `nosqldb.GetRequest` is used for simple get operations. It contains the primary key value for the target row and returns an instance of `nosqldb.GetResult`. If the get operation succeeds, a non-nil `GetResult.Version` is returned.

```
key:=&types.MapValue{}
key.Put("id", 1)
req:=&nosqldb.GetRequest{
    TableName: "users",
    Key: key,
}
res, err:=client.Get(req)
```

By default all read operations are eventually consistent, using `types.Eventual`. This type of read is less costly than those using absolute consistency, `types.Absolute`. This default can

be changed in `nosqldb.RequestConfig` using `RequestConfig.Consistency` before creating the client. It can be changed for a single request using `GetRequest.Consistency` field.

1. Change default consistency for all read operations.

```
cfg:= nosqldb.Config{
  RequestConfig: nosqldb.RequestConfig{
        Consistency: types.Absolute,
        ...
    },
    ...
}
client, err:=nosqldb.NewClient(cfg)
```

2. Change consistency for a single read operation.

```
req:=&nosqldb.GetRequest{
    TableName: "users",
    Key: key,
    Consistency: types.Absolute,
}
```

Node.js

You can read single rows using the `get` method. This method allows you to retrieve a record based on its primary key value. You can set consistency of read operation using Consistency enumeration. By default all read operations are eventually consistent, using `Consistency.EVENTUAL`. This type of read is less costly than those using absolute consistency, `Consistency.ABSOLUTE`. This default consistency for read operations can be set in the initial configuration used to create `NoSQLClient` instance using `consistency` property. You may also change it for a single read operation by setting consistency property in the opt argument of the `get` method.

`get` method returns Promise of `GetResult` which which is plain JavaScript object containing the resulting row and its Version. If the provided primary key does not exist in the table, the value of `row` property will be null. Note that the property names in the provided primary key key object should be the same as underlying table column names.

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const Consistency = require('oracle-nosqldb').Consistency;
const client = new NoSQLClient('config.json');
async function getRowsFromUsersTable() {
   const tableName = 'users';
   try {
      let result = await client.get(tableName,
      { id: 1 });
      console.log('Got row: ' + result.row);
      // Use absolute consistency
      result = await client.get(tableName, 'users',
         { id: 1 }, { consistency: Consistency.ABSOLUTE });
      console.log('Got row with absolute consistency: ' + result.row);
   }
```

```
   catch(error){
      //handle errors
   }
}
```

C#

You can read a single row using the `GetAsync` method. This method allows you to retrieve a row based on its primary key value. This method takes the primary key value as `MapValue`. The field names should be the same as the table primary key column names. You may also pass options as `GetOptions`.

You can set consistency of a read operation using `Consistency` enumeration. By default all read operations are eventually consistent. This type of read is less costly than those using absolute consistency. The default consistency for read operations may be set as Consistency property of `NoSQLConfig`. You may also change the consistency for a single Get operation by using Consistency property of `GetOptions`.

`GetAsync` method returns `Task<GetResult<RecordValue>>`. `GetResult` instance contains the returned Row, the row Version and other information. If the row with the provided primary key does not exist in the table, the values of both Row and Version properties will be null.

```
var client = new NoSQLClient("config.json");
..................................................
var tableName = "users";
try{
    var result = await client.GetAsync(tableName,
        new MapValue
        {
            ["id"] =1
        });
    // Continuing from the Put example, the expected output will be:
    // { "id": 1, "name": "Kim" }
    Console.WriteLine("Got row: {0}", result.row);
    // Use absolute consistency.
    result = await client.GetAsync(tableName,
        new MapValue
        {
            ["id"] =2
        }),
         new GetOptions
         {
            Consistency=Consistency.Absolute
         });
   // The expected output will be:
   // { "id": 2, "name": "Jack" }
   Console.WriteLine("Got row with absolute consistency: {0}",
        result.row);
  // Continuing from the Put example, the expiration time should be
  // 30 days from now.
  Console.WriteLine("Expiration time: {0}", result.ExpirationTime)
}
catch(Exception ex){
```

```
  // handle exceptions
}
```

Spring Data

Use one of these methods to read the data from the table - `NosqlRepository findById()`, `findAllById()`, `findAll()` or using `NosqlTemplate find()`, `findAll()`, `findAllById()`. For details, see SDK for Spring Data API Reference.

> **Note:**
>
> First, create the `AppConfig` class that extends `AbstractNosqlConfiguration` class to provide the connection details of the Oracle NoSQL Database. For more details, see Obtaining a NoSQL connection.

In this section, you use the `NosqlRepository findAll()` method.

Create the `UsersRepository` interface. This interface extends the `NosqlRepository` interface and provides the entity class and the data type of the primary key in that class as parameterized types to the `NosqlRepository` interface. This `NosqlRepository` interface provides methods that are used to retrieve data from the database.

```
import com.oracle.nosql.spring.data.repository.NosqlRepository;

/* The Users is the entity class and Long is the data type of the
primary key in the Users class.
   This interface provides methods that return iterable instances of
the Users class. */

public interface UsersRepository extends NosqlRepository<Users, Long> {
    Iterable<Users> findAll();
}
```

In the application, you select all the rows from the `Users` table and provide them to an iterable instance. Print the values to the output from the iterable object.

```
@Autowired
private UsersRepository repo;

/* Select all the rows in the Users table and provides them into an
iterable instance.*/

System.out.println("\nfindAll:");
Iterable < Users > allusers = repo.findAll();

/* Print the values to the output from the iterable object.*/
for (Users u: allusers) {
    System.out.println(" User: " + u);
}
```

Run the program to display the output.

```
findAll:

User: Users{id=1, firstName=John, lastName=Doe}
User: Users{id=2, firstName=Angela, lastName=Willard}
```

Using Queries

Learn about some aspects of using queries to your application in Oracle NoSQL Database.

Oracle NoSQL Database provides a rich query language to read and update data. See Developers Guide for a full description of the query language.

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

To execute your query, you use the `NoSQLHandle.query()` API.

To execute a `SELECT` query to read data from your table:

```
/* QUERY a table named "users", using the primary key field "name".
 * The table name is inferred from the query statement.
 */
QueryRequest queryRequest = new QueryRequest().
setStatement("SELECT * FROM users WHERE name = \"Taylor\"");

/* Queries can return partial results. It is necessary to loop,
 * reissuing the request until it is "done"
 */

do {
  QueryResult queryResult = handle.query(queryRequest);

  /* process current set of results */
  List<MapValue> results = queryResult.getResults();
  for (MapValue qval : results) {
    //handle result
  }
} while (!queryRequest.isDone());
```

When using queries, be aware of the following considerations:

- You can use prepared queries when you want to run the same query multiple times. When you use prepared queries, the execution is more efficient than starting with a query string every time. The query language and API support query variables to assist with the reuse.
-

For example, to execute a `SELECT` query to read data from your table using a prepared statement:

```
/* Perform the same query using a prepared statement. This is more
 * efficient if the query is executed repeatedly and required if
 * the query contains any bind variables.
 */
String query = "DECLARE $name STRING; " +
               "SELECT * from users WHERE name = $name";

PrepareRequest prepReq = new PrepareRequest().setStatement(query);
/* prepare the statement */
PrepareResult prepRes = handle.prepare(prepReq);
/* set the bind variable and set the statement in the QueryRequest */
prepRes.getPreparedStatement()
       .setVariable("$name", new StringValue("Taylor"));
QueryRequest queryRequest = new
QueryRequest().setPreparedStatement(prepRes);

/* perform the query in a loop until done */
do {
  QueryResult queryResult = handle.query(queryRequest);
  /* handle result */
} while (!queryRequest.isDone());
```

Python

To execute a query use the `borneo.NoSQLHandle.query()` method. For example, to execute a SELECT query to read data from your table, a `borneo.QueryResult` contains a list of results. And if the `borneo.QueryRequest.is_done()` returns False, there may be more results, so queries should generally be run in a loop. It is possible for single request to return no results but the query still not done, indicating that the query loop should continue. For example:

```
from borneo import QueryRequest
# Query at table named 'users" using the field 'name' where name may
match
# 0 or more rows in the table. The table name is inferred from the
query
statement = 'select * from users where name = "Jane"'
request = QueryRequest().set_statement(statement)
# loop until request is done, handling results as they arrive
while True: result = handle.query(request)
# handle results
handle_results(result)
```

```
# do something with results
if request.is_done(): break
```

When using queries it is important to be aware of the following considerations:

- Oracle NoSQL Database provides the ability to prepare queries for execution and reuse. It is recommended that you use prepared queries when you run the same query for multiple times. When you use prepared queries, the execution is much more efficient than starting with a query string every time. The query language and API support query variables to assist with query reuse.
- The `borneo.QueryRequest` allows you to set the read consistency for a query as well as modifying the maximum amount of resource (read and write) to be used by a single request. This can be important to prevent a query from getting throttled because it uses too much resource too quickly.

Here is an example of using a prepared query with a single variable:

```
from borneo import PrepareRequest, QueryRequest
# Use a similar query to above but make the name a variable
statement = 'declare $name string
select * from users where name = $name'
prequest = PrepareRequest().set_statement(statement)
presult = handle.prepare(prequest)
# use the prepared statement, set the variable
pstatement = presult.get_prepared_statement()
pstatement.set_variable('$name', 'Jane')
qrequest = QueryRequest().set_prepared_statement(pstatement)
# loop until qrequest is done, handling results as they arrive
while True:
# use the prepared query in the query
request qresult = handle.query(qrequest)
# handle results
handle_results(qresult)
# do something with results
if qrequest.is_done(): break
# use a different variable value with the same prepared query
pstatement.set_variable('$name', 'another_name')
qrequest = QueryRequest().set_prepared_statement(pstatement)
# loop until qrequest is done, handling results as they arrive
while True:
# use the prepared query in the query
request qresult = handle.query(qrequest)
# handle results
handle_results(qresult)
# do something with results
if qrequest.is_done(): break
```

Go

To execute a query use the `Client.Query` function. For example, to execute a `SELECT` query to read data from your table:

```
prepReq := &nosqldb.PrepareRequest{
    Statement: "select * from users",
```

```
}
prepRes, err := client.Prepare(prepReq)
if err != nil {
    fmt.Printf("Prepare failed: %v\n", err)
    return
}
queryReq := &nosqldb.QueryRequest{
    PreparedStatement: &prepRes.PreparedStatement,
}
var results []*types.MapValue
for {
    queryRes, err := client.Query(queryReq)
    if err != nil {
        fmt.Printf("Query failed: %v\n", err)
        return
    }
    res, err := queryRes.GetResults()
    if err != nil {
        fmt.Printf("GetResults() failed: %v\n", err)
        return
    }
    results = append(results, res...)
    if queryReq.IsDone() {
        break
    }
}
```

Queries should generally be run in a loop and check `QueryRequest.IsDone()` to determine if the query completes. It is possible for a single request to return no results but still have `QueryRequest.IsDone()` evaluated to false, indicating that the query loop should continue.

When using queries it is important to be aware of the following considerations:

- Oracle NoSQL Database provides the ability to prepare queries for execution and reuse. It is recommended that you use prepared queries when you run the same query multiple times. When you use prepared queries, the execution is much more efficient than starting with a query string every time. The query language and API support query variables to assist with query reuse.
- The `nosqldb.QueryRequest` allows you to set the read consistency for a query (via the `QueryRequest.Consistency` field), as well as modifying the maximum amount of resources (read and write, via the `QueryRequest.MaxReadKB` and `QueryRequest.MaxWriteKB` fields) to be used by a single request. This can be important to prevent a query from getting throttled because it uses too many resources too quickly.

Node.js

To execute a query use `query` method. This method returns returns `Promise` of `QueryResult` which is plain JavaScript object that contains an Array of resulting rows as well as continuation key. The amount of data returned by the query is limited by the system default and could be further limited by setting `maxReadKB` property in the opt argument of `query`, which means that one invocation of `query` method may not return all available results. This situation is dealt with by using `continuationKey` property.

Not-null continuation key means that more query results may be available. This means that queries should generally run in a loop, looping until continuation key becomes null. Note that it is possible for rows to be empty yet have not-null `continuationKey`, which means the query loop should continue. In order to receive all the results, call `query` in a loop. At each iteration, if non-null continuation key is received in `QueryResult`, set `continuationKey` property in the `opt` argument for the next iteration:

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
.....
const client = new NoSQLClient('config.json');
async function queryUsersTable() {
    const opt = {};
    try {
        do {
            const result = await client.query('SELECT * FROM users', opt);
            for(let row of result.rows) {
                console.log(row);
            }
            opt.continuationKey = result.continuationKey;
        } while(opt.continuationKey);
    } catch(error) {
        //handle errors
    }
}
```

When using queries it is important to be aware of the following considerations:

- The Oracle NoSQL Database provides the ability to prepare queries for execution and reuse. It is recommended that you use prepared queries when you run the same query for multiple times. When you use prepared queries, the execution is much more efficient than starting with a query string every time. The query language and API support query variables to assist with query reuse.
- Using opt argument of `query` allows you to set the read consistency for query as well as modifying the maximum amount of data it reads in a single call. This can be important to prevent a query from getting throttled.

Use `prepare` method to prepare the query. This method returns Promise of `PreparedStatement` object. Use set method to bind query variables. To run prepared query, pass `PreparedStatement` to the query or `queryIterable` instead of the statement string.

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
.....
const client = new NoSQLClient('config.json');

async function queryUsersTable() {
    const statement = 'DECLARE $name STRING; SELECT * FROM users WHERE ' +
        'name = $name';
    try {
        let prepStatement = await client.prepare(statement);
        const opt = {};
        // Set value for $name variable
        prepStatement.set('$name', 'Taylor');
        do {
            let result = await client.query(prepStatement);
```

```
            for(let row of result.rows) {
                console.log(row);
            }
           opt.continuationKey = result.continuationKey;
        } while (opt.continuationKey);
        // Set different value for $name and re-execute the query
        prepStatement.set('$name', 'Jane');
        do {
            let result = await client.query(prepStatement);
            for(let row of result.rows) {
                console.log(row);
            }
           opt.continuationKey = result.continuationKey;
        } while (opt.continuationKey);
    } catch(error) {
        //handle errors
    }
}
```

C#

To execute a query, you may call `QueryAsync` method or call `GetQueryAsyncEnumerable` method and iterate over the resulting async enumerable. You may pass options to each of these methods as QueryOptions. `QueryAsync` method return `Task<QueryResult<RecordValue>>`. QueryResult contains query results as a list of RecordValue instances, as well as other information. When your query specifies a complete primary key (or you are executing an INSERT statement), it is sufficient to call `QueryAsync` once.

```
var client = new NoSQLClient("config.json");
try {
    var result = await client.QueryAsync(
    "SELECT * FROM users WHERE id = 1");
    // Because we select by primary key, there can be at most one
record.
    if (result.Rows.Count>0) {
       Console.WriteLine("Got record: {0}.", result.Rows[0]);
    }
    else {
       Console.WriteLine("Got no records.");
    }
}
catch(Exception ex) {
  // handle exceptions
}
```

The amount of data returned by the query is limited by the system. It could also be further limited by setting `MaxReadKB` property of `QueryOptions`. This means that one invocation of `QueryAsync` may not return all available results. This situation is dealt with by using continuation key. Non-null ContinuationKey in `QueryResult` means that more more query results may be available. This means that queries should run in a loop, looping until the continuation key becomes `null`.

Note that it is possible for query to return now rows (`QueryResult.Rows` is empty) yet have not-null continuation key, which means that the query should continue looping. To continue the query, set `ContinuationKey` in the `QueryOptions` for the next call to `QueryAsync` and loop until the continuation key becomes *null*. The following example executes the query and prints query results:

```
var client = new NoSQLClient("config.json");
var options = new QueryOptions();
try {
    do {
        var result = await client.QueryAsync(
            "SELECT id, name FROM users ORDER BY name",
            options);
        foreach(var row of result.Rows) {
            Console.WriteLine(row);
        }
        options.ContinuationKey = result.ContinuationKey;
    }
    while(options.ContinuationKey != null);
}
catch(Exception ex){
    // handle exceptions
}
```

Another way to execute the query in a loop is to use `GetQueryAsyncEnumerable`. It returns an instance of `AsyncEnumerable<QueryResult>` that can be iterated over. Each iteration step returns a portion of the query results as QueryResult.

```
var client = new NoSQLClient("config.json");
try {
    await foreach(var result in client.GetQueryAsyncEnumerable(
        "SELECT id, name FROM users ORDER BY name"))
    {
        foreach(var row of result.Rows) {
            Console.WriteLine(row);
        }
    }
}
catch(Exception ex){
    // handle exceptions
}
```

Oracle NoSQL Database provides the ability to prepare queries for execution and reuse. It is recommended that you use prepared queries when you run the same query for multiple times. When you use prepared queries, the execution is much more efficient than starting with a SQL statement every time. The query language and API support query variables to assist with query reuse.

Use `PrepareAsync` to prepare the query. This method returns `Task<PreparedStatement>`. `PreparedStatement` allows you to set query variables. The query methods `QueryAsync` and

`GetQueryAsyncEnumerable` have overloads that execute prepared queries by taking `PreparedStatement` as a parameter instead of the SQL statement. For example:

```
var client = new NoSQLClient("config.json");
try {
    var sql = "DECLARE $name STRING; SELECT * FROM users WHERE " +
        "name = $name";
    var preparedStatement = await client.PrepareAsync(sql);
    // Set value for $name variable and execute the query
    preparedStatement.Variables["$name"] = "Taylor";
    await foreach(var result in client.GetQueryAsyncEnumerable(
        preparedStatement)) {
        foreach(var row of result.Rows) {
            Console.WriteLine(row);
        }
    }
    // Set different value for $name and re-execute the query.
    preparedStatement.Variables["$name"] = "Jane";
    await foreach(var result in client.GetQueryAsyncEnumerable(
        preparedStatement)) {
        foreach(var row of result.Rows) {
            Console.WriteLine(row);
        }
    }
}
catch(Exception ex){
    // handle exceptions
}
```

Spring Data

Use one of these methods to run your query - The NosqlRepository derived queries, native queries, or using `NosqlTemplate runQuery()`, `runQueryJavaParams()`, `runQueryNosqlParams()`. For details, see SDK for Spring Data API Reference.

> **Note:**
>
> First, create the `AppConfig` class that extends `AbstractNosqlConfiguration` class to provide the connection details of the Oracle NoSQL Database. For more details, see Obtaining a NoSQL connection.

In this section, you use the derived queries. For more details on the derived queries, see Derived Queries in the *Spring Data SDK Developers Guide*.

Create the `UsersRepository` interface. This interface extends the `NosqlRepository` interface and provides the entity class and the data type of the primary key in that class as parameterized types to the `NosqlRepository` interface. The `NosqlRepository` interface provides methods that are used to retrieve data from the database.

```
import com.oracle.nosql.spring.data.repository.NosqlRepository;

/* The Users is the entity class and Long is the data type of the
```

```
primary key in the Users class.
   This interface provides methods that return iterable instances of the
Users class. */

public interface UsersRepository extends NosqlRepository<Users, Long> {
    /* Search the Users table by the last name and return an iterable
instance of the Users class.*/
    Iterable<Users> findByLastName(String lastname);
}
```

In the application, you select the row from the `Users` table with the last name as required and print the values to the output from the object.

```
@Autowired
private UsersRepository repo;

System.out.println("\nfindBylastName: Willard");

/* Use queries to find by the last Name. Search the Users table by the last
name and return an iterable instance of the Users class.*/
allusers = repo.findByLastName("Willard");

for (Users s: allusers) {
    System.out.println(" User: " + s);
}
```

Run the program to display the output.

```
findBylastName: Willard

User: Users{id=2, firstName=Angela, lastName=Willard}
```

Deleting Data

Learn how to delete rows from your table.

After you insert or load data into a table, you can delete the table rows when they are no longer required.

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

To delete a row from a table:

```
/* identify the row to delete */
MapValue delKey = new MapValue().put("id", 2);

/* construct the DeleteRequest */
DeleteRequest delRequest = new DeleteRequest().setKey(delKey)
                                              .setTableName("users");
/* Use the NoSQL handle to execute the delete request */
DeleteResult del = handle.delete(delRequest);
/* on success DeleteResult.getSuccess() returns true */
if (del.getSuccess()) {
  // success, row was deleted
} else {
  // failure, row either did not exist or conditional delete failed
}
```

You can perform a sequence of `DeleteRequest` operations on a table using the `MultiDeleteRequest` class.

Python

Single rows are deleted using `borneo.DeleteRequest` using a primary key value as shown below.

```
from borneo import DeleteRequest
# DeleteRequest requires table name and primary key
request = DeleteRequest().set_table_name('users')
request.set_key({'id': 1})
# perform the operation
result = handle.delete(request)
if result.get_success():
# success -- the row was deleted
# if the row didn't exist or was not deleted for any other reason,
False is returned
```

Delete operations can be conditional based on a `borneo.Version` returned from a get operation.You can perform multiple deletes in a single operation using a value range using `borneo.MultiDeleteRequest` and `borneo.NoSQLHandle.multi_delete()` .

Go

Single rows are deleted using `nosqldb.DeleteRequest` using a primary key value:

```
key := &types.MapValue{}
key.Put("id", 1)
req := &nosqldb.DeleteRequest{
    TableName: "users",
    Key: key,
}
res, err := client.Delete(req)
```

Delete operations can be conditional based on a `types.Version` returned from a get operation.

Node.js

To delete a row, use `delete` method. Pass to it the table name and primary key of the row to delete. In addition, you can make delete operation conditional by specifying on a Version of the row that was previously returned by `get` or `put`. You can pass it as matchVersion property of the opt argument: `{ matchVersion: my_version }`. Alternatively you may use `deleteIfVersion` method.

`delete` and `deleteIfVersion` methods return Promise of `DeleteResult`, which is plain JavaScript object, containing success status of the operation.

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const client = new NoSQLClient('config.json');

async function deleteRowsFromUsersTable() {
    const tableName = 'users';
    try {
        let result = await client.put(tableName, { id: 1, name: 'John' });

        // Unconditional delete, should succeed
        result = await client.delete(tableName, { id: 1 });
        // Expected output: delete succeeded
        console.log('delete ' + result.success ? 'succeeded' : 'failed');

        // Delete with non-existent primary key, will fail
        result = await client.delete(tableName, { id: 2 });
        // Expected output: delete failed
        console.log('delete ' + result.success ? 'succeeded' : 'failed');

        // Re-insert the row
        result = await client.put(tableName, { id: 1, name: 'John' });
        let version = result.version;

        // Will succeed because the version matches existing row
        result = await client.deleteIfVersion(tableName, { id: 1 }, version);
        // Expected output: deleteIfVersion succeeded
        console.log('deleteIfVersion ' + result.success ?
            'succeeded' : 'failed');

        // Re-insert the row
        result = await client.put(tableName, { id: 1, name: 'John' });

        // Will fail because the last put has changed the row version, so
        // the old version no longer matches.  The result will also contain
        // existing row and its version because we specified returnExisting
in
        // the opt argument.
        result = await client.deleteIfVersion(tableName, { id: 1 }, version,
            { returnExisting: true });
        // Expected output: deleteIfVersion failed
        console.log('deleteIfVersion ' + result.success ?
            'succeeded' : 'failed');
```

```
        // Expected output: { id: 1, name: 'John' }
        console.log(result.existingRow);
    } catch(error) {
        //handle errors
    }
}
```

Note that similar to put operations, `success` results in false value only if trying to delete row with non-existent primary key or because of version mismatch when matching version was specified. Failure for any other reason will result in error. You can delete multiple rows having the same shard key in a single atomic operation using `deleteRange` method. This method deletes set of rows based on partial primary key (which must be a shard key or its superset) and optional `FieldRange` which specifies a range of values of one of the other (not included into the partial key) primary key fields.

C#

To delete a row, use `DeleteAsync` method. Pass to it the table name and primary key of the row to delete. This method takes the primary key value as `MapValue`. The field names should be the same as the table primary key column names. You may also pass options as `DeleteOptions`. In addition, you can make delete operation conditional by specifying on a `RowVersion` of the row that was previously returned by `GetAsync` or `PutAsync`. Use `DeleteIfVersionAsync` method that takes the row version to match. Alternatively, you may use `DeleteAsync` method and pass the version as `MatchVersion` property of `DeleteOptions`.

```
var client = new NoSQLClient("config.json");
var tableName = "users";
try
{
    var row = new MapValue
    {
        ["id"] = 1,
        ["name"] = "John"
    };

    var putResult = await client.PutAsync(tableName, row);
    Console.WriteLine("Put {0}.",
       putResult.Success ? "succeeded" : "failed");

    var primaryKey = new MapValue
    {
        ["id"] = 1
    };
    // Unconditional delete, should succeed.
    var deleteResult = await client.DeleteAsync(tableName, primaryKey);
    // Expected output: Delete succeeded.
    Console.WriteLine("Delete {0}.",
        deleteResult.Success ? "succeeded" : "failed");
    // Delete with non-existent primary key, should fail.
    var deleteResult = await client.DeleteAsync(tableName,
        new MapValue
        {
            ["id"] = 200
```

```
        });
    // Expected output: Delete failed.
    Console.WriteLine("Delete {0}.",
        deleteResult.Success ? "succeeded" : "failed");
    // Re-insert the row and get the new row version.
    putResult = await client.PutAsync(tableName, row);
    var version = putResult.Version;
    // Delete should succeed because the version matches existing
    // row.
    deleteResult = await client.DeleteIfVersionAsync(tableName,
        primaryKey, version);
    // Expected output: DeleteIfVersion succeeded.
    Console.WriteLine("DeleteIfVersion {0}.",
        deleteResult.Success ? "succeeded" : "failed");
    // Re-insert the row
    putResult = await client.PutAsync(tableName, row);
    // This delete should fail because the last put operation has
    // changed the row version, so the old version no longer matches.
    // The result will also contain existing row and its version because
    // we specified ReturnExisting in DeleteOptions.
    deleteResult = await client.DeleteIfVersionAsync(tableName,
        primaryKey, version);
    // Expected output: DeleteIfVersion failed.
    Console.WriteLine("DeleteIfVersion {0}.",
        deleteResult.Success ? "succeeded" : "failed");
    // Expected output: { "id": 1, "name": "John" }
    Console.WriteLine(result.existingRow);
}
catch(Exception ex) {
    // handle exceptions
}
```

Note that Success property of the result only indicates whether the row to delete was found and for conditional Delete, whether the provided version was matched. If the Delete operation fails for any other reason, an exception will be thrown. You can delete multiple rows having the same shard key in a single atomic operation using `DeleteRangeAsync` method. This method deletes set of rows based on partial primary key (which must include a shard key) and optional FieldRange which specifies a range of values of one of the other (not included into the partial key) primary key fields.

Spring Data

Use one of these methods to delete the rows from the tables - `NosqlRepository deleteById()`, `delete()`, `deleteAll(Iterable<? extends T> entities)`, `deleteAll()` or using `NosqlTemplate delete()`, `deleteAll()`, `deleteById()`, `deleteInShard()`. For details, see SDK for Spring Data API Reference.

> **Note:**
>
> First, create the `AppConfig` class that extends `AbstractNosqlConfiguration` class to provide the connection details of the Oracle NoSQL Database. For more details, see Obtaining a NoSQL connection.

In this section, you use the `NosqlRepository deleteAll()` method to delete the rows from your table.

Create the `UsersRepository` interface. This interface extends the `NosqlRepository` interface and provides the entity class and the data type of the primary key in that class as parameterized types to the `NosqlRepository` interface. The `NosqlRepository` interface provides methods that are used to retrieve data from the database.

```
import com.oracle.nosql.spring.data.repository.NosqlRepository;

/* The Users is the entity class and Long is the data type of the
primary key in the Users class.
   This interface provides methods that return iterable instances of
the Users class. */

public interface UsersRepository extends NosqlRepository<Users, Long> {
    Iterable<Users> findAll();
}
```

In the application, you use the `deleteAll()` method to delete the existing rows from the table.

```
@Autowired
private UsersRepository repo;

/* Delete all the existing rows if any, from the Users table.*/
repo.deleteAll();
```

Modifying Tables

Learn how to modify tables.

You modify a table to:

- Add new fields to an existing table
- Delete currently existing fields in a table
- To change the default TTL value

Examples of DDL statements are:

```
/* Add a new field to the table */
ALTER TABLE users (ADD age INTEGER)

/* Drop an existing field from the table */
ALTER TABLE users (DROP age)

/* Modify the default TTL value*/
ALTER TABLE users USING TTL 4 days
```

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

You can change the definition of the table. The TTL value is changed below.

```
/* Alter the users table to modify the TTL value to 4 days.
* When modifying the table schema or other table state you cannot also
* modify the table limits. These must be independent operations.
*/
String alterTableDDL = "ALTER TABLE users " + "USING TTL 4 days";
TableRequest treq = new TableRequest().setStatement(alterTableDDL);
/* start the operation, it is asynchronous */
TableResult tres = handle.tableRequest(treq);
/* wait for completion of the operation */
tres.waitForCompletion(handle, 60000, 1000);
/* wait for 60 sec */
/* delay in ms for poll */
```

Python

You can change the TTL value of a table as shown below.

```
/* Alter the users table to modify the TTL value to 4 days.
* When modifying the table schema or other table state you cannot also
* modify the table limits. These must be independent operations.
*/
TableRequest statement = "ALTER TABLE users " + "USING TTL 4 days";
request = TableRequest().set_statement(statement)
# assume that a handle has been created, as handle, make the request
#wait for 60 seconds, polling every 1 seconds
result = handle.do_table_request(request, 60000, 1000)
result.wait_for_completion(handle, 60000, 1000)
```

Go

Specify the DDL statement and other information in a `TableRequest`, and execute the request using the `nosqldb.DoTableRequest()` or `nosqldb.DoTableRequestAndWait()` function.

```
req:=&nosqldb.TableRequest{
    Statement: "ALTER TABLE users (ADD age INTEGER)",
}
res, err:=client.DoTableRequestAndWait(req, 5*time.Second, time.Second)
```

Node.js

You can change the TTL value of a table as shown below.

```
/* Alter the users table to modify the TTL value to 4 days.
*/
const statement = 'ALTER TABLE users ' + 'USING TTL 4 days';
let result = await client.tableDDL(statement, complete: true });
console.log('Table users altered');
```

C#

Use `ExecuteTableDDLAsync` or `ExecuteTableDDLWithCompletionAsync` to modify a table by issuing a DDL statement against this table.

You can change the TTL value of a table as shown below.

```
/* Alter the users table to modify the TTL value to 4 days.
*/
var statement = "ALTER TABLE users " + "USING TTL 4 days";
var result = await client.ExecuteTableDDLAsync(statement);
await result.WaitForCompletionAsync();
Console.WriteLine("Table users altered.");
```

Spring Data

To modify a table, you can use the `NosqlTemplate.runTableRequest()` method. For details, see SDK for Spring Data API Reference.

> **Note:**
>
> While the Spring Data SDK provides an option to modify the tables, it is not recommended to alter the schemas as the Spring Data SDK expects tables to comply with the default schema (two columns - the primary key column of types String, int, long, or timestamp and a JSON column called kv_json_).

Drop Regions

Learn how to delete a region that you have created in a Multi-Region Oracle NoSQL Database.

Even though you can drop a region directly in a multi-region environment, it is recommended that you isolate the region to be dropped from other participating regions before dropping it. To learn more about isolating a region, see

To drop a region, use the `DROP REGION` DDL statement. For example:

```
/* Drop the region named us_east */
DROP REGION us_east;
```

To create a region from your application, you use the `KVStore` class to pass the DDL statement to the `executeSync` method. For example to drop the `us_east` region:

```
KVStore.executeSync("DROP REGION us_east");
```

Drop Tables and Indexes

Learn how to delete a table or index that you have created in Oracle NoSQL Database.

To drop a table or index, use the `DROP TABLE` or `DROP INDEX` DDL statements. For example:

```
/* Drop the table named users */
DROP TABLE users

/* Drop the index called nameIndex on the table users */
DROP INDEX IF EXISTS nameIndex ON users
```

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

The following example considers that the default compartment is specified in `NoSQLHandleConfig` while obtaining the NoSQL handle. See Obtaining a NoSQL Handle. To explore other options of specifying a compartment for the NoSQL tables, see About Compartments .

To drop a table using the `TableRequests.setStatement` method:

```
/* create the TableRequest to drop the users table */
TableRequest tableRequest = new TableRequest().setStatement("drop table
users");

/* start the operation, it is asynchronous */
TableResult tres = handle.tableRequest(tableRequest);

/* wait for completion of the operation */
tres.waitForCompletion(handle,
                       60000,  /* wait for 60 sec */
                       1000);  /* delay in ms for poll */
```

You can drop an MR Table using the `DROP` statement in the same manner as you drop any other Oracle NoSQL Database table. If you choose to drop an MR Table in a particular

region, it continues to remain an MR Table in the other participating regions. In a case where you want to drop a particular MR Table from multiple regions, you must execute the `DROP TABLE` statement in each region separately.

> **Note:**
>
> If you drop an MR Table in all the regions except one, it becomes an MR Table linked with a single region. The difference between an MR Table with a single region and a local table is that you can add new regions to the MR Table with a single region in the future.

To drop a table or index from your application, you use the `TableRequest` class. For example to drop the `users` table:

```
/* Drop the table identified by the tableName */
final String dropStatement = "drop table " + users;

/* Pass the dropStatement string to the TableRequest.setStatement
method */
TableRequest tableRequest = new
TableRequest().setStatement(dropStatement);

/* Wait for the table state to change to DROPPED. */
TableResult tres = handle.tableRequest(tableRequest);
tres = TableResult.waitForState(handle,tres.getTableName(),
       TableResult.State.DROPPED,
                           30000, /* wait 30 sec */
                           1000); /* delay ms for poll */
```

Python

The following example drops the table users.

```
from borneo import TableRequest
# the drop statement
statement = 'drop table users'
request = TableRequest().set_statement(statement)
# perform the operation, wait for 40 seconds, polling every 3 seconds
result = handle.do_table_request(request, 40000, 3000)
```

Go

The following example drops the given table.

```
// Drop the table
dropReq := &nosqldb.TableRequest{Statement: "DROP TABLE IF EXISTS " +
tableName}
tableRes, err = client.DoTableRequestAndWait(dropReq, 60*time.Second,
time.Second)
if err != nil {
    fmt.Printf("failed to drop table: %v\n", err)
    return
```

```
}
fmt.Println("Dropped table " + tableName)
```

Node.js

The following example drops the given table and index.

```
const NoSQLClient = require('oracle-nosqldb').NoSQLClient;
const TableState = require('oracle-nosqldb').TableState;
.....
const client = new NoSQLClient('config.json');

async function dropNameIndexUsersTable() {
    try {
        let result = await client.tableDDL('DROP INDEX nameIndex ON users');
        // Before using the table again, wait for the operation completion
        // (when the table state changes from UPDATING to ACTIVE)
        await client.forCompletion(result);
        console.log('Index dropped');
    } catch(error) {
        //handle errors
    }
}

async function dropTableUsers() {
    try {
        // Here we are waiting until the drop table operation is completed
        // in the underlying store
        let result = await client.tableDDL('DROP TABLE users', {
            completion: true
        });
        console.log('Table dropped');
    } catch(error) {
        //handle errors
    }
}
```

C#

To drop tables, use `ExecuteTableDDLAsync` and `ExecuteTableDDLWithCompletionAsync`.

```
var client = new NoSQLClient("config.json");
try {
    // Drop index "nameIndex" on table "users".
    var result = await client.ExecuteTableDDLAsync(
        "DROP INDEX nameIndex ON users");
    // The following may print: Table state is Updating.
    Console.WriteLine("Table state is {0}", result.TableState);
    await result.WaitForCompletionAsync();
    // Expected output: Table state is Active.
    Console.WriteLine("Table state is {0}.", result.TableState);
    // Drop table "TestTable".
    result = await client.ExecuteTableDDLWithCompletionAsync(
        "DROP TABLE TestTable");
```

```
    // Expected output: Table state is Dropped.
    Console.WriteLine("Table state is {0}.", result.TableState);
}
catch(Exception ex) {
    // handle exceptions
}
```

Spring Data

To drop the tables and Indexes, you use `NosqlTemplate.runTableRequest()` or `NosqlTemplate.dropTableIfExists()` methods. For details, see SDK for Spring Data API Reference.

Create the `AppConfig` class that extends `AbstractNosqlConfiguration` class to provide the connection details of the database. For more details, see Obtaining a NoSQL connection.

In the application, you instantiate the `NosqlTemplate` class by providing the `NosqlTemplate create(NosqlDbConfig nosqlDBConfig)` method with the instance of the `AppConfig` class. You then drop the table using the `NosqlTemplate.dropTableIfExists()` method. The `NosqlTemplate.dropTableIfExists()` method drops the table and returns `true` if the result indicates a change of the table's state to DROPPED or DROPPING.

```
import com.oracle.nosql.spring.data.core.NosqlTemplate;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

/* Drop the Users table.*/

try {
    AppConfig config = new AppConfig();
    NosqlTemplate tabledrop =
NosqlTemplate.create(config.nosqlDbConfig());
    Boolean result = tabledrop.dropTableIfExists("Users");
    if (result == true) {
        System.out.println("Table dropped successfully");
    } else {
        System.out.println("Failed to drop table");
    }
} catch (Exception e) {
    System.out.println("Exception creating index" + e);
}
```

Handling Errors

Learn how to handle errors and exceptions.

- Java
- Python
- Go
- Node.js
- C#
- Spring Data

Java

Java errors are thrown as exceptions when you build or run your application. The `NoSQLException` class is the base for most exceptions thrown by the driver. However, the driver throws exceptions directly for some classes, such as `IllegalArgumentException` and `NullPointerException`.

In general, NoSQL exception instances are split into two broad categories:

- Exceptions that may be retried with the expectation that they may succeed on retry.

  These exceptions are instances of the `RetryableException` class. These exceptions usually indicate resource consumption violations.

- Exceptions that will fail even after retry.

  Examples of exceptions that should not be retried are `IllegalArgumentException`, `TableNotFoundException`, and any other exception indicating a syntactic or semantic error.

Python

Python errors are raised as exceptions defined as part of the API. They are all instances of Python's RuntimeError. Most exceptions are instances of `borneo.NoSQLException` which is a base class for exceptions raised by the Python driver.

Exceptions are split into 2 broad categories: Exceptions that may be retried with the expectation that they may succeed on retry. These are all instances of `borneo.RetryableException`. Examples of these are the instances of `borneo.ThrottlingException` which is raised when resource consumption limits are exceeded. Exceptions that should not be retried, as they will fail again. Examples of these include `borneo.IllegalArgumentException` , `borneo.TableNotFoundException`, etc.

`borneo.ThrottlingException` instances will never be thrown in an on-premise configuration as there are no relevant limits.

Go

Go SDK errors are reported as `nosqlerr.Error` values defined as part of the API. Errors are split into 2 broad categories:

- Errors that may be retried with the expectation that they may succeed on retry. These are retryable errors on which the `Error.Retryable()` method call returns `true`. Examples of these include `nosqlerr.OperationLimitExceeded`, `nosqlerr.ReadLimitExceeded`,

nosqlerr.WriteLimitExceeded, which are raised when resource consumption limits are exceeded.

- Errors that should not be retried, as they will fail again. Examples of these include nosqlerr.IllegalArgumentError, nosqlerr.TableNotFoundError, etc.

Node.js

Asynchronous methods of NoSQLClient return Promise as a result and if an error occurs it results in the Promise rejection with that error. For synchronous methods such as NoSQLClient constructor errors are thrown as exceptions. All errors used by the SDK are instances of NoSQLError or one of its subclasses. In addition to the error message, each error has errorCode property set to one of standard error codes defined by the ErrorCode enumeration. errorCode may be useful to execute conditional logic depending on the nature of the error.

For some error codes, specific subclasses of NoSQLError are defined, such as NoSQLArgumentError, NoSQLProtocolError, NoSQLTimeoutError, etc. NoSQLAuthorizationError may have one of several error codes depending on the cause of authorization failure. In addition, errors may have cause property set to the underlying error that caused the current error. Note that the cause is optional and may be an instance of an error that is not part of the SDK.

In addition, error codes are split into 2 broad categories:

- Errors that may be retried with the expectation that the operation may succeed on retry. Examples of these are ErrorCode.READ_LIMIT_EXCEEDED and ErrorCode.WRITE_LIMIT_EXCEEDED which are throttling errors (relevant for the Cloud environment), and also ErrorCode.NETWORK_ERROR since most network conditions are temporary.
- Errors that should not be retried, as the operation will most likely fail again. Examples of these include ErrorCode.ILLEGAL_ARGUMENT (represented by NoSQLArgumentError), ErrorCode.TABLE_NOT_FOUND, etc.

You can determine if the NoSQLError is retryable by checking retryable property. Its value is set to true for retryable errors and is false or undefined for non-retryable errors.

Retry Handler

The driver will automatically retry operations on a retryable error. Retry handler determines:

- Whether and how many times the operation will be retried.
- How long to wait before each retry.

RetryHandler is an interface with with 2 properties:

- RetryHandler#doRetry that determines whether the operation should be retried based on the operation, number of retries happened so far and the error occurred. This property is usually a function, but may be also be set to boolean false to disable automatic retries.
- RetryHandler#delay that determines how long to wait before each successive retry based on the same information as provided to RetryHandler#doRetry. This property is usually a function, but may also be set to number of milliseconds for constant delay.

C#

NoSQLException serves as a base class for many exceptions thrown by the driver. However, in certain cases the driver uses standard exception types such as:

- `ArgumentException` and its subclasses such as `ArgumentNullException`. They are thrown when an invalid argument is passed to a method or when an invalid configuration (in code or in JSON) is passed to create NoSQLClient instance.
- `TimeoutException` is thrown when an operation issued by `NoSQLClient` has timed out. If you are getting many timeout exceptions, you may try to increase the timeout values in `NoSQLConfig` or in options argument passed to the `NoSQLClient` method.
- `InvalidOperationException` is thrown when the service is an invalid state to perform an operation. It may also be thrown if the query has failed be cause its processing exceeded the memory limit specifed in `QueryOptions.MaxMemoryMB` or `NoSQLConfig.MaxMemoryMB`. In this case, you may increase the corresponding memory limit. Otherwise, you may retry the operation.
- `InvalidCastException` and `OverflowException` may occur when working with sublcasses of FieldValue and trying to cast a value to a type it doesn't support or cast a numeric value to a smaller type causing arithmetic overflow.
- `OperationCanceledException` and `TaskCanceledException` if you issued a cancellation of the operation started by a method of `NoSQLClient` using the provided CancellationToken.

In addition, exceptions may be split into two broad categories:

- Exceptions that may be retried with the expectation that the operation may succeed on retry. In general these are subclasses of `RetryableException`. These include throttling exceptions as well as other exceptions where a resource is temporarily unavailable. Some other subclasses of `NoSQLException` may also be retryable depending on the conditions under which the exception occurred. In addition, network-related errors are retryable because most network conditions are temporary.
- Exceptions that should not be retried because they will still fail after retry. They include exceptions such as `TableNotFoundException`, `TableExistsException` and others as well as standard exceptions such as `ArgumentException`.

You can determine if a given instance of `NoSQLException` is retryable by checking its `IsRetryable` property.

Retry Handler

By default, the driver will automatically retry operations that threw a retryable exception (see above). The driver uses retry handler to control operation retries. The retry handler determines:

- Whether and how many times the operation will be retried.
- How long to wait before each retry.

All retry handlers implement `IRetryHandler` interface. This interface provides two methods, one to determine if the operation in its current state should be retried and another to determine a retry delay before the next retry. You have a choice to use default retry handler or set your own retry handler as `RetryHandler` property of `NoSQLConfig` when creating `NoSQLClient` instance.

> **Note:**
>
> Retries are only performed within the timeout period alloted to the operation and configured as one of timeout properties in `NoSQLConfig` or in options passed to the `NoSQLClient` method. If the operation or its retries have not succeded before the timeout is reached, `TimeoutException` is thrown.

By default, the driver uses `NoSQLRetryHandler` class which controls retires based on operation type, exception type and whether the number of retries performed has reached a preconfigured maximum. It also uses exponential backoff delay to wait between retries starting with a pre configured base delay. You may customize the properties such as maximum number of retries, base delay and others by creating your own instance of `NoSQLRetryHandler` and setting it as a `RetryHandler` property in `NoSQLConfig`. For example:

```
var client = new NoSQLClient(
    new NoSQLConfig
    {
        Region = .....,
        ...............
        RetryHandler = new NoSQLRetryHandler
        {
            MaxRetryAttempts = 20,
            BaseDelay = TimeSpan.FromSeconds(2)
        }
    });
```

If you don't specify the retry handler, the driver will use an instance of `NoSQLRetryHandler` with default values for all parameters. Alternatively, you may choose to create your own retry handler class by implementing `IRetryHandler` interface. The last option is to disable retries all together. You may do this if you plan to retry the operations within your application instead. To disable retries, set `RetryHandler` property of `NoSQLConfig` to `NoRetries`:

```
var client = new NoSQLClient(
    new NoSQLConfig
    {
        Region = .....,
        ...............
        RetryHandler = NoSQLConfig.NoRetries
    });
```

Handle Resource Limits: Programming in a resource-limited environment can be challenging. Tables have user-specified throughput limits and if an application exceeds those limits it may be throttled, which means an operation may fail with one of the throttling exceptions such as `ReadThrottlingException` or `WriteThrottlingException`. This is most common when using queries, which can read a lot of data, using up capacity very quickly. It can also happen for get and put operations that run in a tight loop.

Even though throttling errors will be retried and using custom `RetryHandler` may allow more direct control over retries, an application should not rely on retries to handle throttling as this will result in poor performance and inability to use all of the throughput available for the table. The better approach would be to avoid throttling entirely by rate-limiting your application. In this context rate-limiting means keeping operation rates under the limits for the table.

Spring Data

The Spring Data errors are thrown as exceptions when you build or run your application.

The Spring Data SDK uses `IllegalArgumentException` for invalid parameters and passes through Java SDK and Spring Framework exceptions. The Spring framework throws exceptions directly for some classes, such as `BeansException`.

You can add an `exception` code block to catch any error that might be thrown such as authentication failure during connection setup. Additionally, you can enable logging in Oracle NoSQL Database SDK for Spring Data at ERROR and DEBUG levels. For more details, see Activating Logging in the *Spring Data SDK Developers Guide*.

Configuring Multi-Region Data Stores

Oracle NoSQL Database supports Multi-Region Architecture in which you can create tables in multiple data stores, and still maintain consistent data across these clusters. Each data store in a Multi-Region Oracle NoSQL Database setup is called a *Region*. A Multi-Region Table or *MR Table* is a global logical table that is stored and maintained in different regions. MR Tables maintain consistent data in all the regions. That is, any updates made to an MR Table in one region automatically applies to the corresponding MR Table in all the other participating regions. To learn more about Oracle NoSQL Database Multi-Region Architecture and MR Tables, see Multi-Region Architecture in the *Concepts Guide*.

You can configure a Multi-Region Oracle NoSQL Database, and create and manipulate the MR Tables using the Oracle NoSQL Database command-line interface (CLI).The remainder of this chapter is organized into four use cases to demonstrate the different features of the Multi-Region Oracle NoSQL Database and MR Tables. The examples provided show you which commands to use and how. For a complete list of all the commands available in the CLI, see Admin CLI Reference.

Child MR Tables:

You can create child tables in the Multi-Region architecture. That means an existing Multi-Region table can have child tables. You can add a child table to a top-level table that is already a Multi-Region table, and the child table will be automatically Multi-Region enabled . That is an entire table hierarchy is Multi-Region or none of it is. You can drop a child from a top-level table, and the child table will be removed from the hierarchy. The child table will also be removed from the MR Table graph such that it no longer participates in cross region replication.

Use Cases

- Use Case 1: Set up Multi-Region Environment
- Use Case 2: Expand a Multi-Region Table
- Use Case 3: Contract a Multi-Region Table

- Use Case 4: Drop a Region

Use Case 1: Set up Multi-Region Environment

An organization deploys two on-premise data stores, one each at Frankfurt and London. As per their requirement, they create a few MR Tables in both the regions. The `Users` table is one of the many MR Tables created and maintained by this organization. In the next few topics, let us discuss how to set up the Frankfurt and London regions and how to create and work with an MR Table called `Users` in these two regions.

To configure a Multi-Region NoSQL Database, you need to execute the below listed tasks in each region. For the use case under discussion, you must execute all the below listed steps in both the participating regions, Frankfurt and London.

1. Deploy the data store
2. Set Local Region Name
3. Configure XRegion Service
4. Start XRegion Service
5. Create Remote Regions
6. Create Multi-Region Tables
7. Access and Manipulate Multi-Region Tables

Deploy the data store

In each region in the Multi-Region NoSQL Database setup, you must deploy its own data store independently.

Steps:

To deploy the data store:

1. Follow the instructions given in Configuring your data store installation.
2. After deploying the data store of your desired topology, you can check the health of the data store by executing the `ping` command from the command line interface.

   ```
   [~]$ java -jar $KVHOME/lib/kvstore.jar ping -port <port number> -
   host <host name>
   ```

3. You can also verify the topology of the data store by executing the `show topology` command from the `kv` prompt. See show topology.

   ```
   kv-> show topology
   ```

Example:

For the use case under discussion, you must set up data stores for the two regions proposed.

```
# Connect to the data stores deployed at host1, host2, and host3 from the kv
prompt
[~]$java -jar $KVHOME/lib/kvstore.jar runadmin \
-helper-hosts host1:5000,host2:5000,host3:5000

# View the topology of the data store
kv-> show topology
store=mrtstore  numPartitions=1000 sequence=1008
  zn: id=zn1 name=zn1 repFactor=3 type=PRIMARY allowArbiters=false
masterAffinity=false

  sn=[sn1] zn:[id=zn1 name=zn1] host1:5000 capacity=1 RUNNING
    [rg1-rn1] RUNNING
             single-op avg latency=0.8630216 ms   multi-op avg
latency=1.7694647 ms
  sn=[sn2] zn:[id=zn1 name=zn1] host2:5000 capacity=1 RUNNING
    [rg1-rn2] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=2.0211697 ms
  sn=[sn3] zn:[id=zn1 name=zn1] host3:5000 capacity=1 RUNNING
    [rg1-rn3] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=1.8524266 ms

  numShards=1
  shard=[rg1] num partitions=1000
    [rg1-rn1] sn=sn1
    [rg1-rn2] sn=sn2
    [rg1-rn3] sn=sn3


# Connect to the data store deployed at host4, host5, and host6 from the kv
prompt
[~]$java -jar $KVHOME/lib/kvstore.jar runadmin \
-helper-hosts host4:5000,host5:5000,host6:5000

# View the topology of the data store
kv-> show topology
store=mrtstore  numPartitions=1000 sequence=1008
  zn: id=zn1 name=zn1 repFactor=3 type=PRIMARY allowArbiters=false
masterAffinity=false

  sn=[sn1] zn:[id=zn1 name=zn1] host4:5000 capacity=1 RUNNING
    [rg1-rn1] RUNNING
             single-op avg latency=0.7519707 ms   multi-op avg
latency=2.000658 ms
  sn=[sn2] zn:[id=zn1 name=zn1] host5:5000 capacity=1 RUNNING
    [rg1-rn2] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=3.2067895 ms
  sn=[sn3] zn:[id=zn1 name=zn1] host6:5000 capacity=1 RUNNING
    [rg1-rn3] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=1.9516457 ms
```

```
  numShards=1
  shard=[rg1] num partitions=1000
    [rg1-rn1] sn=sn1
    [rg1-rn2] sn=sn2
    [rg1-rn3] sn=sn3
```

Set Local Region Name

Learn how to set a name to the local region in a Multi-Region NoSQL Database.

After deploying the data store and before creating the first MR Table in each participating region, you must set a *local region name*. You can change the local region name as long as no MR Tables are created in that region. After creating the first MR Table, the local region name becomes immutable.

Steps:

To set the local region name:

1. Connect to the `sql` prompt from the local region, and connect to the local data store.
2. Execute the following command from the `sql` prompt.

   ```
   sql-> SET LOCAL REGION <local region name>;
   ```

3. Optionally, you can execute the following command to verify that the local region name is set successfully.

   ```
   sql-> SHOW REGIONS;
   ```

Example:

Set the local region name for the two proposed regions, Frankfurt and London.

```
# Connect to the data store deployed at host1, host2, and host3 from
the SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

-- Set the local region name to 'fra'
sql-> SET LOCAL REGION fra;
Statement completed successfully

-- List the regions
sql-> SHOW REGIONS;
regions
    fra (local, active)


# Connect to the data store deployed at host4, host5, and host6 from
the SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
```

```
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore

-- Set the local region name to 'lnd'
sql-> SET LOCAL REGION lnd;
Statement completed successfully

-- List the regions
sql-> SHOW REGIONS;
regions
    lnd (local, active)
```

Configure XRegion Service

Learn how to configure the XRegion Service in a Multi-Region Oracle NoSQL Database

Before creating any MR Table, you must deploy an XRegion Service. In simple terms, this is also called an agent. The XRegion Service runs independently with the local data store and it is recommended to deploy it close to the local data store. To know more about agent and agent groups, see Cross-Region Service in the *Concepts Guide*.

Steps:

To configure the XRegion Service, execute the following tasks in each region:

1. Create a home directory for the XRegion Service.
2. Create a JSON config file in the home directory created in the step 1. The structure of the `json.config` file is shown below.

```
{
  "path" : "<entire path to the home directory for the XRegion Service>",
  "agentGroupSize" : <number of service agents>,
  "agentId" : <agent id using 0-based numbering>,
  "region" : "<local region name>",
  "store" : "<local store name>",
  "helpers" : [
    "<host1>:<port>",
    "<host2>:<port>",
    …
    "<hostn>:<port>"
  ],
  "security" : "<entire path to the security file of the local store>",
  "regions" : [
    {
        "name" : "<remote region name>",
        "store" : "<remote store name>",
        "security" : "<entire path to the security file of the remote
store>",
        "helpers" : [
            "<host1>:<port>",
            "<host2>:<port>",
            …
            "<hostn>:<port>
         ]
    },
```

```
    {
        "name" : "<remote region name>",
        "store" : "<remote store name>",
        "security" : "<entire path to the security file of the
remote store>",
        "helpers" : [
            "<host1>:<port>",
            "<host2>:<port>",
            …
            "<hostn>:<port>
        ]
    },
    ...
  ]
  "durability" : "<durability setting>"
}
```

Where each attribute in the `json.config` file is explained below:

`path`	This is the root directory of the XRegion Service. The agents use this directory to dump logs, statistics and other auxiliary files. The directory shall be readable and writable to the agents.
`agentGroupSize` and `agentId`	Specifies the number of service agents and the Agent ID in the agent group. The Agent ID is specified as numbers starting from 0. These details are used to form a group of agents that serve the local region. Forming a group of agents achieves horizontal scalability. **Note:** The current release supports only a single service for each local region. Therefore the `agentGroupSize` is set to 1 and the `agentId` is set to 0.
`security`	Specifies the security file used by the agent. This attribute must be defined for the local store as well as the remote stores.
`region`	Specifies the local region name defined for the region where you are configuring the agent.
`store`	Specifies the name of the store in the local region.

`helpers`	Specifies the list of host and port numbers used for configuring the local store. These helper hosts are those you used to create a KV client. For XRegion Service to connect to the local and remote regions, each region's firewall must be configured to open the registry port and HA ports.
`regions`	After defining the local region, you must define a list of remote regions. At least one remote region shall be defined in order to create an MR Table. Specifies the region name, store name, and helper hosts for each remote region you want to include. **Note:** The remote region names listed here must be same as the local region names defined for them.
`durability`	This is an optional parameter. It specifies the durability setting for Master commit synchronization. The possible values are: • `COMMIT_NO_SYNC` • `COMMIT_SYNC` • `COMMIT_WRITE_NO_SYNC` The default durability setting is `COMMIT_NO_SYNC`.

3. Grant the following privileges to the XRegion Service Agent:
   - Write permission to system table
   - Read and Write permission to all the user tables

```
— create role for the agent --
CREATE ROLE <Agent Role>

— grant privileges to the role --
GRANT WRITE_SYSTEM_TABLE to <Agent Role>
GRANT READ_ANY_TABLE to <Agent Role>
GRANT INSERT_ANY_TABLE to <Agent Role>

— grant role to the agent user --
GRANT <Agent Role> to user <Agent User>
```

> **Note:**
>
> This step is required only for secure data stores. In a non-secure data store setup, this step can be skipped.

Example:

Create a `json.config` file for each proposed region, Frankfurt and London.

```
# Contents of the configuration file in the 'fra' Region
{
  "path": "<path to the json config file>",
  "agentGroupSize": 1,
  "agentId": 0,
  "region": "fra",
  "store": "<storename at the fra region>",
  "security": "<path to the security file>",
  "helpers": [
    "host1:5000",
    "host2:5000",
    "host3:5000"
  ],
  "regions": [
    {
      "name": "lnd",
      "store": "<storename at the lnd region>",
      "security": "<path to the security file>",
      "helpers": [
        "host4:5000",
        "host5:5000",
        "host6:5000"
      ]
    }
  ]
}


# Contents of the configuration file in the 'lnd' Region
{
  "path": "<path to the json config file>",
  "agentGroupSize": 1,
  "agentId": 0,
  "region": "lnd",
  "store": "<storename at the lnd region>",
  "security": "<path to the security file>",
  "helpers": [
    "host4:5000",
    "host5:5000",
    "host6:5000"
  ],
  "regions": [
    {
      "name": "fra",
```

```
      "store": "<storename at the fra region>",
      "security": "<path to the security file>",
      "helpers": [
        "host1:5000",
        "host2:5000",
        "host3:5000"
      ]
    }
  ]
}
```

Start XRegion Service

The Xregion service in each region can be started using the `xrstart` command. The `xrstart` command has to be executed for each data store separately. The status of the `xrstart` command execution can be viewed by reading the contents of `nohup.out` file. To get more details about `xrstart` command and its various parameters, see xrstart .

Example:

Start the XRegion Service in both the regions, Frankfurt and London.

```
# Start the XRegion Service in the 'fra' Region
[oracle@host1 xrshome]$nohup java -Xms256m -Xmx2048m -jar $KVHOME/lib/
kvstore.jar xrstart \
-config <path to the json config file> > \
<path to the home directory of the xregion service>/nohup.out &

[1] 24356
[oracle@host1 xrshome]$ nohup: ignoring input and redirecting stderr to
stdout

# View the status of the xrstart command in the 'fra' Region
[oracle@host1 xrshome]$ cat nohup.out
Cross-region agent (region=fra,store=mrtstore, helpers=[host1:5000,
host2:5000, host3:5000])
starts up from config file=/home/oracle/xrshome/ json.config at 2019-11-07
08:57:34 UTC


# Start the XRegion Service in the 'lnd' Region
[oracle@host4 xrshome]$ nohup java -Xms256m -Xmx2048m -jar $KVHOME/lib/
kvstore.jar xrstart \
-config <path to the json config file> > \
<path to the home directory of the xregion service>/nohup.out &

[1] 17587
[oracle@host4 xrshome]$ nohup: ignoring input and redirecting stderr to
stdout

# View the status of the xrstart command in the 'lnd' Region
[oracle@host4 xrshome]$ cat nohup.out
Cross-region agent (region=lnd,store=mrtstore, helpers=[host4:5000,
host5:5000, host6:5000])
```

```
starts up from config file=/home/oracle/xrshome/ json.config at
2019-11-07 08:57:34 UTC
```

Create Remote Regions

Learn to create remote regions from each region in a Multi-Region NoSQL Database.

Before creating and operating on an MR table, you must define the remote regions. You have already set the local region name for each region, in an earlier step. In this step, you define all the remote regions for each region. A remote region is different from the local region where the command is executed.

Steps:

To create the remote regions:

1. Connect to the `sql` prompt from the local region, and connect to the local data store.
2. Execute the following command from the `sql` prompt.

   ```
   sql-> CREATE REGION <remote region name>;
   ```

3. Optionally, you can execute the following command to list the remote regions that are created successfully.

   ```
   sql-> SHOW REGIONS;
   ```

Example:

Create the remote regions in both the regions, Frankfurt and London.

```
# Connect to the data store deployed in the 'fra' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

-- Create a remote region 'lnd'
sql-> CREATE REGION lnd;
Statement completed successfully

- List the regions
sql-> SHOW REGIONS;
regions

    fra (local, active)
    lnd (remote, active)


# Connect to the data store deployed in the 'lnd' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore
```

```
-- Create a remote region 'fra'
sql-> CREATE REGION fra;
Statement completed successfully

- List the regions
sql-> SHOW REGIONS;
regions

    lnd (local, active)
    fra (remote, active)
```

Create Multi-Region Tables

You must create an MR Table on each data store in the connected graph, and specify the list of regions that the table should span. For the use case under discussion, you must create the `users` table as an MR Table at both the regions, in any order.

Steps:

To create an MR Table:

1. To create a table definition, use a `CREATE TABLE` statement. See Create Table in the Developers Guide.
2. Optionally, you can verify the regions associated with the MR Table by executing the following command from the `kv` prompt.

```
kv-> SHOW TABLE -NAME <table name>
```

Example:

Create an MR Table called `users` in both the regions, Frankfurt and London.

```
# Connect to the data store deployed in the 'fra' region from the SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

-- Create the users MR Table
sql-> CREATE TABLE users(
   -> id INTEGER,
   -> name STRING,
   -> team STRING,
   -> PRIMARY KEY (id))
   -> IN REGIONS fra,lnd;
Statement completed successfully

# Connect to the data store deployed in the 'fra' region from the kv prompt
[~]$java -jar $KVHOME/lib/kvstore.jar runadmin \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

# Verify the regions associated with the users MR table
kv-> SHOW TABLE -NAME users
```

```
{
  "json_version": 1,
  "type": "table",
  "name": "users",
  "regions": {
    "1": "fra",
    "2": "lnd"
  },
  "fields": [
    {
      "name": "id",
      "type": "INTEGER",
      "nullable": false
    },
    {
      "name": "name",
      "type": "STRING",
      "nullable": true
    },
    {
      "name": "team",
      "type": "STRING",
      "nullable": true
    }
  ],
  "primaryKey": [
    "id"
  ],
  "shardKey": [
    "id"
  ]
}
```

```
# Connect to the data store deployed in the 'lnd' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore

-- Create the users MR Table
sql-> CREATE TABLE users(
   -> id INTEGER,
   -> name STRING,
   -> team STRING,
   -> PRIMARY KEY (id))
   -> IN REGIONS lnd,fra;
Statement completed successfully

# Connect to the data store deployed in the 'lnd' region from the kv
prompt
[~]$java -jar $KVHOME/lib/kvstore.jar runadmin \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore
```

```
# Verify the regions associated with the users MR table
kv-> SHOW TABLE -NAME users
{
  "json_version": 1,
  "type": "table",
  "name": "users",
  "regions": {
    "2": "fra",
    "1": "lnd"
  },
  "fields": [
    {
      "name": "id",
      "type": "INTEGER",
      "nullable": false
    },
    {
      "name": "name",
      "type": "STRING",
      "nullable": true
    },
    {
      "name": "team",
      "type": "STRING",
      "nullable": true
    }
  ],
  "primaryKey": [
    "id"
  ],
  "shardKey": [
    "id"
  ]
}
```

Create multi-region table with an MR_COUNTER column

You can create a multi-region table containing a column of MR_COUNTER datatype. MR_COUNTER datatype is used in such situations to take care of conflict resolution that may arise when the same data is modified across different regions. MR_COUNTER ensures that though data modifications happen simultaneously in different regions, the data can always be merged into a consistent state. This merge is performed automatically by the MR_COUNTER data type without requiring any special conflict resolution code or user intervention. To learn more about MR_COUNTER datatype, see Using CRDT datatype in a multi-region table section in the Concepts Guide.

Example:

Create an MR Table called `users` with a `MR_COUNTER` datatype in both the regions, Frankfurt and London.

```
-- Create the users MR Table
sql-> CREATE TABLE users(
   -> id INTEGER,
   -> name STRING,
```

```
   -> team STRING,
   -> count INTEGER AS MR_COUNTER,
   -> PRIMARY KEY (id))
   -> IN REGIONS fra,lnd;
Statement completed successfully


# Verify the regions associated with the users MR table
sql-> DESC AS JSON TABLE users
{
   "json_version": 1,
   "type": "table",
   "name": "users",
   "regions": {
      "1": "fra",
      "2": "lnd"
   },
   "fields": [
   {
      "name": "id",
      "type": "INTEGER",
      "nullable": false
   },
   {
      "name": "name",
      "type": "STRING",
      "nullable": true
   },
   {
       "name": "team",
       "type": "STRING",
       "nullable": true
   },
   {
       "name" : "count",
       "type" : "INTEGER",
       "nullable" : false,
       "default" : 0,
       "MRCounter" : true
  }
],
   "primaryKey": [
       "id"
   ],
   "shardKey": [
       "id"
   ]
}


# Connect to the KVStore deployed in the 'lnd' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
     -helper-hosts host4:5000,host5:5000,host6:5000 \
     -store mrtstore
```

```
-- Create the users MR Table
sql-> CREATE TABLE users(
   -> id INTEGER,
   -> name STRING,
   -> team STRING,
   -> count INTEGER AS MR_COUNTER,
   -> PRIMARY KEY (id))
   -> IN REGIONS lnd,fra;
Statement completed successfully


# Verify the regions associated with the users MR table
sql-> DESC AS JSON TABLE users
{
   "json_version": 1,
   "type": "table",
   "name": "users",
   "regions": {
      "2": "fra",
      "1": "lnd"
   },
   "fields": [
   {
      "name": "id",
      "type": "INTEGER",
      "nullable": false
   },
   {
      "name": "name",
      "type": "STRING",
      "nullable": true
   },
   {
       "name": "team",
       "type": "STRING",
       "nullable": true
   },
   {
        "name" : "count",
        "type" : "INTEGER",
        "nullable" : false,
        "default" : 0,
        "MRCounter" : true
   }
],
   "primaryKey": [
       "id"
   ],
   "shardKey": [
       "id"
   ]
}
```

To know more details about how to create and use an MR_COUNTER datatype, See Using the MR_COUNTER datatype section in the SQL Reference Guide.

You can use the MR_COUNTER data type in a schema-less JSON field, which means if your Multi-Region table has a JSON column, you can use MR_COUNTER data type inside the JSON column. One or more fields in the JSON column can be of MR_COUNTER data type. The MR_COUNTER data type is a subtype of the INTEGER or LONG or NUMBER data type.

Example:

```
CREATE TABLE demoJSONMR(name STRING,
  jsonWithCounter JSON(counter as INTEGER MR_COUNTER,
                       person.count as LONG MR_COUNTER),
  PRIMARY KEY(name)) IN REGIONS fra,lnd;
```

Access and Manipulate Multi-Region Tables

After creating the MR Table, you can perform read or write operations on the table using the existing data access APIs or DML statements. There is no change to any existing data access APIs or DML statements to work with the MR Tables. See Data Row Management in the *SQL Reference Guide*.

Example:

Perform DML operations on the `users` table in one region, and verify if the changes are replicated to the other region.

```
# To be executed in the fra region
-- Insert two rows into the users MR Table
sql-> INSERT INTO users(id,name,team) VALUES(1,"Amy","HR");
{"NumRowsInserted":1}
1 row returned
sql-> INSERT INTO users(id,name,team) VALUES(2,"Jack","HR");
{"NumRowsInserted":1}
1 row returned

# To be executed in the lnd region
-- Verify if the rows are replicated from the fra region
sql-> SELECT * FROM users;
{"id":1,"name":"Amy","team":"HR"}
{"id":2,"name":"Jack","team":"HR"}

2 rows returned

-- Update the row with id = 2 in the users MR Table
sql-> UPDATE users SET team = "IT" WHERE id = 2;
{"NumRowsUpdated":1}
1 row returned

-- Delete the row with id = 1 from the users MR Table
sql-> DELETE FROM users WHERE id = 1;
{"NumRowsDeleted":1}
1 row returned

# To be executed in the fra region
-- Verify if the changes are replicated from the lnd region
sql-> SELECT * FROM users;
```

```
{"id":2,"name":"Jack","team":"IT"}
1 row returned
```

Stop XRegion Service

In a multi-region setup, you can stop any running Xregion service using `xrstop` command. To get more details about the `xrstop` command, see xrstop.

Example:

Stop the XRegion Service in both the regions, Frankfurt and London.

```
# Stopping the XRegion Service in the fra region
[~]$ java -Xmx1024m -Xms256m -jar $KVHOME/lib/kvstore.jar xrstop \
-config <path to the json config file>
```

Similarly, you must stop the XRegion Service in the other region, lnd.

Use Case 2: Expand a Multi-Region Table

An organization deploys two on-premise data stores, one each at Frankfurt and London. As per their requirement, they create a few MR Tables in both the regions. The `users` table is one of the many MR Tables created and maintained by this organization. Now, they decide to expand their organization by adding another NoSQL Database in Dublin. After creating Dublin as the new region, they want to expand the existing MR Tables to the new region. In the next few topics, you learn how to add the Dublin region to the `users` table that you already created in the previous use case.

If you have not created the `users` MR Table earlier, execute the steps outlined in Use Case 1: Set up Multi-Region Environment.

Prerequisites

Steps:

Before expanding the `users` table to the new region, you must have set up the new region by executing the following tasks:

1. Set up a multi-region NoSQL Database with two regions Frankfurt (`fra`) and London (`lnd`). See Use Case 1: Set up Multi-Region Environment.
2. Deploy a local data store with store name as `dubstore` in the new region. See Configuring your data store installation.
3. Set the new region's local region name to `dub`. See Set Local Region Name.
4. Configure and start the XRegion Service in the `dub` region. See Configure XRegion Service and Start XRegion Service.
5. Update the `json.config` file in the existing regions, that is, Frankfurt (`fra`) and London (`lnd`) to include `dub` (Dublin) as a remote region.

> **Note:**
>
> You must restart the agent at existing regions to pick up the new region (`dub`) from the `json.config` file.

6. Create two remote regions, `fra` and `lnd` in the new region `dub`. See Create Remote Regions.

Example:

1. Set the local region name for the new region, Dublin.

```
# Connect to the data store deployed at host7, host8, and host9
from the SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host7:5000,host8:5000,host9:5000 \
-store dubstore

-- Set the local region name to 'dub'
sql-> SET LOCAL REGION dub;
Statement completed successfully

-- List the regions
sql-> SHOW REGIONS;
regions
    dub (local, active)
```

2. Create a `json.config` file for the new region, Dublin.

```
# Contents of the configuration file in the 'dub' Region
{
  "path": "<entire path to the home directory for the XRegion
Service>",
  "agentGroupSize": 1,
  "agentId": 0,
  "region": "dub",
  "store": "<storename at the dub region>",
  "security": "<path to the security file>",
  "helpers": [
    "host7:5000",
    "host8:5000",
    "host9:5000"
  ],
  "regions": [
    {
      "name": "fra",
      "store": "<store name at the fra region>",
      "security": "<path to the security file>",
      "helpers": [
        "host1:5000",
        "host2:5000",
        "host3:5000"
      ]
    },
```

```
    {
      "name": "lnd",
      "store": "<store name at the lnd region>",
      "security": "<path to the security file>",
      "helpers": [
        "host4:5000",
        "host5:5000",
        "host6:5000"
      ]
    }
  ]
}
```

3. Start the XRegion Service in the new region, Dublin.

```
# Start the XRegion Service in the 'dub' Region
[oracle@host7 xrshome]$nohup java -Xms256m -Xmx2048m -jar $KVHOME/lib/
kvstore.jar xrstart \
-config <path to the json config file> > \
<path to the home directory of the xregion service>/nohup.out &

[1] 24123
[oracle@host7 xrshome]$ nohup: ignoring input and redirecting stderr to
stdout

# View the status of the xrstart command in the 'dub' Region
[oracle@host7 xrshome]$ cat nohup.out
Cross-region agent (region=fra,store=mrtstore, helpers=[host7:5000,
host8:5000, host9:5000])
starts up from config file=/home/oracle/xrshome/ json.config at
2020-11-07 08:57:34 UTC
```

4. Modify the `json.config` files in the existing regions (Frankfurt and London) to include Dublin as a remote region.

```
# Contents of the configuration file in the 'fra' Region
{
  "path": "<path to the json config file>",
  "agentGroupSize": 1,
  "agentId": 0,
  "region": "fra",
  "store": "<storename at the fra region>",
  "security": "<path to the security file>",
  "helpers": [
    "host1:5000",
    "host2:5000",
    "host3:5000"
  ],
  "regions": [
    {
      "name": "lnd",
      "store": "<storename at the lnd region>",
      "security": "<path to the security file>",
      "helpers": [
        "host4:5000",
```

```
        "host5:5000",
        "host6:5000"
      ]
    },
    {
      "name": "dub",
      "store": "<storename at the dub region>",
      "security": "<path to the security file>",
      "helpers": [
        "host7:5000",
        "host8:5000",
        "host9:5000"
      ]
    }
  ]
}
```

 # Contents of the configuration file in the 'lnd' Region

```
{
  "path": "<path to the json config file>",
  "agentGroupSize": 1,
  "agentId": 0,
  "region": "lnd",
  "store": "<storename at the lnd region>",
  "security": "<path to the security file>",
  "helpers": [
    "host4:5000",
    "host5:5000",
    "host6:5000"
  ],
  "regions": [
    {
      "name": "fra",
      "store": "<storename at the fra region>",
      "security": "<path to the security file>",
      "helpers": [
        "host1:5000",
        "host2:5000",
        "host3:5000"
      ]
    },
    {
      "name": "dub",
      "store": "<storename at the dub region>",
      "security": "<path to the security file>",
      "helpers": [
        "host7:5000",
        "host8:5000",
        "host9:5000"
      ]
    }
  ]
}
```

5. Create two remote regions, `fra` and `lnd` in the new region, Dublin.

```
# Connect to the data store deployed in the 'dub' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host7:5000,host8:5000,host9:5000 \
-store dubstore

-- Create remote regions 'fra' and 'lnd'
sql-> CREATE REGION fra;
Statement completed successfully
sql-> CREATE REGION lnd;
Statement completed successfully

- List the regions
sql-> SHOW REGIONS;
regions

    dub (local, active)
    fra (remote, active)
    lnd (remote, active)
```

Create MR Table in New Region

Steps:

As a first step in expanding an MR Table to a new region, you must create the MR Table in the new region using the `CREATE TABLE` statement. See Create Multi-Region Tables.

> **Note:**
>
> Creating the MR Table in the new region alone does not ensure replicating the data from the existing regions. This is because you have not yet linked the new region to this MR Table from the existing regions.

Example:

Create the `users` MR Table in the new region, Dublin.

```
# Connect to the data store deployed in the 'dub' region from the SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host7:5000,host8:5000,host9:5000 \
-store dubstore

-- Create the users MR Table
sql-> CREATE TABLE users(
   -> id INTEGER,
   -> name STRING,
   -> team STRING,
   -> PRIMARY KEY (id))
   -> IN REGIONS dub,fra,lnd;
Statement completed successfully
```

```
# Connect to the data store deployed in the 'dub' region from the kv
prompt
[~]$java -jar $KVHOME/lib/kvstore.jar runadmin \
-helper-hosts host7:5000,host8:5000,host9:5000 \
-store dubstore

# Verify the regions associated with the users MR table
kv-> SHOW TABLE -NAME users
{
  "json_version": 1,
  "type": "table",
  "name": "users",
  "regions": {
    "1": "dub",
    "2": "fra"
    "3": "lnd"
  },
  "fields": [
    {
      "name": "id",
      "type": "INTEGER",
      "nullable": false
    },
    {
      "name": "name",
      "type": "STRING",
      "nullable": true
    },
    {
      "name": "team",
      "type": "STRING",
      "nullable": true
    }
  ],
  "primaryKey": [
    "id"
  ],
  "shardKey": [
    "id"
  ]
}
```

Add New Region to Existing Regions

As a next step, you must create the new region as a remote region in the existing regions. Then, you must associate the new region with the MR Table in the existing regions.

Steps:

Execute the following steps *from each existing region*:

1. Add the new region as a remote region. See Create Remote Regions.

2. Associate the new region with the existing MR Table using the DDL command shown below.

```
ALTER TABLE <table name> ADD REGIONS <region name>;
```

Example:

1. Add the new region, Dublin as a remote region from the existing regions, Frankfurt and London.

```
# Connect to the data store deployed in the 'fra' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

-- Create a remote region 'dub'
sql-> CREATE REGION dub;
Statement completed successfully

- List the regions
sql-> SHOW REGIONS;
regions

    fra (local, active)
    lnd (remote, active)
    dub (remote, active)


# Connect to the data store deployed in the 'lnd' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore

-- Create a remote region 'dub'
sql-> CREATE REGION dub;
Statement completed successfully

- List the regions
sql-> SHOW REGIONS;
regions

    lnd (local, active)
    fra (remote, active)
    dub (remote, active)
```

2. In the existing regions, alter the `users` MR Table to add the new region, Dublin.

```
# Connect to the data store deployed in the 'fra' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore
```

```
-- Add the 'dub' region to the users MR Table
sql-> ALTER TABLE users ADD REGIONS dub;
Statement completed successfully

# Connect to the data store deployed in the 'fra' region from the
kv prompt
[~]$java -jar $KVHOME/lib/kvstore.jar runadmin \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

# Verify the regions associated with the users MR table
kv-> SHOW TABLE -NAME users
{
  "json_version": 1,
  "type": "table",
  "name": "users",
  "regions": {
    "1": "fra",
    "2": "lnd"
    "3": "dub"
  },
  "fields": [
    {
      "name": "id",
      "type": "INTEGER",
      "nullable": false
    },
    {
      "name": "name",
      "type": "STRING",
      "nullable": true
    },
    {
      "name": "team",
      "type": "STRING",
      "nullable": true
    }
  ],
  "primaryKey": [
    "id"
  ],
  "shardKey": [
    "id"
  ]
}


# Connect to the data store deployed in the 'lnd' region from the
SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore

-- Add the 'dub' region to the users MR Table
```

```
sql-> ALTER TABLE users ADD REGIONS dub;
Statement completed successfully

# Connect to the data store deployed in the 'lnd' region from the kv
prompt
[~]$java -jar $KVHOME/lib/kvstore.jar runadmin \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore

# Verify the regions associated with the users MR table
kv-> SHOW TABLE -NAME users
{
  "json_version": 1,
  "type": "table",
  "name": "users",
  "regions": {
    "1": "lnd",
    "2": "fra"
    "3": "dub"
  },
  "fields": [
    {
      "name": "id",
      "type": "INTEGER",
      "nullable": false
    },
    {
      "name": "name",
      "type": "STRING",
      "nullable": true
    },
    {
      "name": "team",
      "type": "STRING",
      "nullable": true
    }
  ],
  "primaryKey": [
    "id"
  ],
  "shardKey": [
    "id"
  ]
}
```

Access MR Table in New and Existing Regions

After performing the tasks discussed in the previous sections, you can perform read/write operations on the MR Table from the new region without any disruption. However, the table may not return the complete data from the existing regions until the initialization completes in the background. Especially if the MR Table has a huge volume of data in the existing regions, it may take a while for the new table to see the data from the remote regions.

Similarly, you can continue performing read/write operations on the MR Table from the existing regions without any disruption. Adding a new region is transparent to the customers

accessing the MR Table from the existing regions. However, the MR Table at the existing regions may also need initialization to see the writes from the new region. If the table at the new region is empty or small, the existing regions will quickly sync up with it. To learn how to access the MR Tables, see Access and Manipulate Multi-Region Tables.

Use Case 3: Contract a Multi-Region Table

An organization deploys three on-premise data stores, one each at Frankfurt, London, and Dublin. As per their requirement, they created a few MR Tables in all three regions. The `users` table is one of the many MR tables created and maintained by this organization. As per some changes in their business requirements, they decided to remove the `users` table from the Dublin region. In the next few topics, you learn how to contract an MR Table, that is, how to remove an MR Table from specific regions.

If you have not created the `users` MR table earlier, execute the steps outlined in Use Case 1: Set up Multi-Region Environment.

If you have not added the Dublin region to the `users` MR table, execute the steps outlined in Use Case 2: Expand a Multi-Region Table.

Alter MR Table to Drop Regions

Learn how to contract a Multi-Region table and reduce the regions it spans across.

Steps:

To remove an MR Table from a specific region in a Multi-Region NoSQL Database setup, you must execute the following steps from all the other participating regions.

1. Execute the following command from the `sql` prompt.

   ```
   ALTER TABLE <table name> DROP REGIONS <comma separated list of
   regions>
   ```

2. Optionally, you can execute the following command from the kv prompt to verify that the region is dropped successfully.

   ```
   SHOW TABLE -NAME <table name>
   ```

> **Note:**
>
> Suppose you drop region A from an MR table created in region B. Then:
>
> - Region B can't see any new writes on this MR table from the region A.
> - Region A continues to see the writes on this MR Table from the region B.
>
> If you want to isolate the MR table in the region A from other regions, you must drop those regions from the MR table created in region A. This is only a recommendation and not a mandatory step in contracting an MR Table.

Example:

Drop the Dublin region from the `users` MR table in the other two regions, Frankfurt and London.

```
# Connect to the data store deployed in the 'fra' region from the SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

-- drop the 'dub' region from the 'users' MR table
sql-> ALTER TABLE users DROP REGIONS dub;
Statement completed successfully


# Connect to the data store deployed in the 'lnd' region from the SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore

-- drop the 'dub' region from the 'users' MR table
sql-> ALTER TABLE users DROP REGIONS dub;
Statement completed successfully
```

Use Case 4: Drop a Region

An organization deploys three on-premise data stores, one each at Frankfurt, London, and Dublin. As per their requirement, they created a few MR Tables in all three regions. As part of business down-sizing, they decided to exclude the Dublin region resulting in a two-region NoSQL Database. In the next few topics, you learn how to drop an existing region from the NoSQL environment that you had set up in the previous sections.

If you have not set up a Multi-Region NoSQL Database with three regions already, execute the steps outlined in:

- Use Case 1: Set up Multi-Region Environment
- Use Case 2: Expand a Multi-Region Table

Prerequisites

Learn about the conditions to be satisfied before dropping a region from a Multi-Region NoSQL Database.

Before removing a region from a Multi-Region NoSQL Database, it is recommended to:

- Stop writing to all the MR Tables linked to that region.
- Ensure that all writes to the MR Tables in that region have replicated to the other regions. This helps in maintaining consistent data across the different regions.

> **Note:**
>
> As of the current release, there is no provision in Oracle NoSQL Database to make a table read-only. Hence, you must stop writes to the identified MR Tables at the application level.

Isolate the Region

Learn how to isolate a region from a Multi-Region NoSQL Database.

When you decide to drop a region, it is a good practice to isolate that region from all the other participating regions. Isolating a region disconnects it from all the MR Tables in the Multi-Region NoSQL Database.

Isolating a region ensures that:

- The isolated region cannot see writes from the other regions.
- The other regions cannot see writes from the isolated region.

> **Note:**
>
> Even though it is not mandatory to isolate the region before dropping it from a Multi-Region NoSQL Database, this is considered a cleaner approach and hence suggested.

Steps:

Isolating a region from the Multi-Region NoSQL Database environment involves two tasks. They are:

1. Drop the target region from all the MR Tables in the other regions using the DDL command shown below.
2. Drop all the other regions from all the MR Tables in the region to be isolated.

See Alter MR Table to Drop Regions.

Example:

1. Drop the Dublin region from the `users` MR table in the other two regions, Frankfurt and London.

```
# Connect to the data store deployed in the 'fra' region from the
SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

-- drop the 'dub' region from the 'users' MR table
```

```
sql-> ALTER TABLE users DROP REGIONS dub;
Statement completed successfully


# Connect to the data store deployed in the 'lnd' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore

-- drop the 'dub' region from the 'users' MR table
sql-> ALTER TABLE users DROP REGIONS dub;
Statement completed successfully
```

2. Drop the other regions (Frankfurt and London) from the `users` MR table in the Dublin region.

```
# Connect to the data store deployed in the 'dub' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host7:5000,host8:5000,host9:5000 \
-store dubstore

-- drop 'fra' and 'lnd' regions from the 'users' MR table
sql-> ALTER TABLE users DROP REGIONS fra,lnd;
Statement completed successfully
```

Drop MR Tables in the Isolated Region

After you ensure that the region to be dropped is isolated, you can drop all the MR Tables created in that region safely. Dropping an MR Table is exactly similar to dropping a local table.

Example:

Drop `users` MR table from the isolated region, Dublin.

```
# Connect to the data store deployed in the 'dub' region from the SQL shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host7:5000,host8:5000,host9:5000 \
-store dubstore

-- drop the 'users' MR table
sql-> DROP TABLE users;
Statement completed successfully
```

Drop the Isolated Region

Finally, you can drop the isolated region from all the other regions.

> **Note:**
>
> Dropping an isolated region is not mandatory. You can retain the isolated region without dropping from other regions, for any future use.

Steps:

To drop the isolated region from other regions:

1. Connect to the `sql` prompt, and connect to the local KVStore.
2. Execute the following DDL command from the SQL prompt.

   ```
   DROP REGION <region name>;
   ```

3. Optionally, you can execute the following command to verify that the isolated region is dropped successfully.

   ```
   SHOW REGIONS;
   ```

Example:

Drop the Dublin region from the other two regions, Frankfurt and London.

```
# Connect to the data store deployed in the 'fra' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host1:5000,host2:5000,host3:5000 \
-store mrtstore

-- drop the 'dub' region
sql-> DROP REGION dub;
Statement completed successfully

- List the regions
sql-> SHOW REGIONS;
regions

    fra (local, active)
    lnd (remote, active)


# Connect to the data store deployed in the 'lnd' region from the SQL
shell
[~]$java -jar $KVHOME/lib/sql.jar \
-helper-hosts host4:5000,host5:5000,host6:5000 \
-store mrtstore

-- drop the 'dub' region
sql-> DROP REGION dub;
Statement completed successfully

- List the regions
sql-> SHOW REGIONS;
```

```
regions

    lnd (local, active)
    fra (remote, active)
```

Use Case 5: Backup and Restore a Multi-Region Table

An organization deploys three on-premise data stores, one each at Frankfurt, London, and Dublin, and they have created MR Tables spanning all three regions. A multi-region table is a single table that spans multiple regions and is kept in sync all the time.

One day due to an application bug or illegal modifications, the organization suffers table-level data loss or corruption for an MR table in one region. Since the system keeps the MR table in sync in all the regions, the corruption or data loss gets replicated that to the other regions as well. Therefore, the organization wants to restore the MR table in all the regions from the MR table backup that they have been regularly creating as part of their data safety policy.

In this topic, you learn how to backup and restore an MR Table.

Backup a Multi-Region Table

Creating a backup of MR tables helps you in restoring the table data later in case you suffer inadvertent application corruption of the data.

To create a backup of an MR table:

1. Using multi-region table statistics, find the most up-to-date region for the table that you want to backup. Run the following command in the Admin Command Line Interface (CLI):

   ```
   show mrtable-agent-statistics -agent 0 -json
   ```

   For more information about MR table statistics, see show mrtable-agent-statistics.

2. In the statistics returned by the show mrtable-agent-statistics command, locate `"returnValue"[]."statistics"."regionStat"[]."laggingMs"."max"` attribute and find the region that has the smallest value for the `max` attribute in the `laggingMS` field. That region contains the most up-to-date data of your MR table.
   In the example below, the Frankfurt region has the smallest value for the `max` attribute in the `laggingMS` field, and hence it has the most up-to-date data for the MR table.

   ```
   kv-> show mrtable-agent-statistics -agent 0 -json
   {
     "operation": "show mrtable-agent-statistics",
     "returnCode": 5000,
     "description": "Operation ends successfully",
     "returnValue": {
       "XRegionService-1_0": {
         "timestamp": 1592901180001,
         "statistics": {
           "agentId": "XRegionService-1_0",
           "beginMs": 1592901120001,
           "dels": 1024,
           "endMs": 1592901180001,
           "incompatibleRows": 100,
           "intervalMs": 60000,
   ```

```
"localRegion": "slc1",
"persistStreamBytes": 524288,
"puts": 2048,
"regionStat": {
  "fra": {
    "completeWriteOps": 10,
    "laggingMs": {
      "avg": 502,
      "max": 885,
      "min": 26
    },
    "lastMessageMs": 1591594977587,
    "lastModificationMs": 1591594941686,
    "latencyMs": {
      "avg": 20,
      "max": 40,
      "min": 10
    }
  },
  "lnd": {
    "completeWriteOps": 10,
    "laggingMs": {
      "avg": 512,
      "max": 998,
      "min": 31
    },
    "lastMessageMs": 1591594977587,
    "lastModificationMs": 1591594941686,
    "latencyMs": {
      "avg": 20,
      "max": 40,
      "min": 10
    }
  },
  "dub": {
    "completeWriteOps": 20,
    "laggingMs": {
      "avg": 535,
      "max": 1024,
      "min": 45
    },
    "lastMessageMs": 1591594978254,
    "lastModificationMs": 1591594956786,
    "latencyMs": {
      "avg": 30,
      "max": 45,
      "min": 15
    }
  }
},
"requests": 12,
"responses": 12,
"streamBytes": 1048576,
"winDels": 1024,
"winPuts": 2048
```

```
        }
      }
    }
  }
```

3. Using the Oracle NoSQL Database Migrator connect to the region identified in **Step#2** in the Source configuration to export the MR tables. And use the appropriate Sink type based on your requirement to import the MR tables. For more information on the source and sink see, Using Oracle NoSQL Data Migrator.

> **Note:**
>
> Make sure that you save the backup of the MR table on remote storage, which is not local to a NoSQL Storage Node in the NoSQL topology.

Restore a Multi-Region Table

You can restore an MR table from an MR table backup in case you suffer data loss or data corruption and want to revert to the most up-to-date version of the MR table.

> **Tip:**
>
> Oracle recommends that you stop all the write activity to the MR tables that are being restored.

To restore an MR table from backup:

1. Find the list of regions associated with the MR Table by executing the following command from the kv prompt.

   ```
   kv-> SHOW TABLE -NAME <table name>
   ```

   For example,

   ```
   kv-> SHOW TABLE -NAME users
   {
     "json_version": 1,
     "type": "table",
     "name": "users",
     "regions": {
       "1": "fra",
       "2": "lnd"
     },
    .....
    .....
   }
   ```

2. Using the `DROP TABLE` statement, drop the MR table in each region with which the MR table is associated. For more information on how to drop an MR table, see Drop Tables and Indexes.

3. Re-create the MR table in every region, specifying the remote regions you want to associate with the MR table. For more information, see Create Multi-Region Tables.
4. Using the Oracle NoSQL Database Migrator connect to any one region identified in Step#1 in the Sink configuration to restore the MR tables. And use the appropriate Source configuration type based on the where the MR table backup resides. During the loading of the backup, the Oracle NoSQL Database synchronizes the table in each remote region. For more information on see, Using Oracle NoSQL Data Migrator.

Troubleshooting multi-region data store setup

1. **Find agent logs for a multi-region setup:**
   Users can find the logs of an XRegion agent at the path specified in the JSON config file. The agent logs, like data store logs, contain all diagnostic information from the service agent. To learn more about the JSON config file used by the XRegion agent, see Configure XRegion Service.
2. **Access the statistics of an XRegion agent**
   The XRegion agent collects statistics periodically and posts it to a system table in the local region. You can query the system table for XRegion agent statistics by using the standard CLI command “SHOW” that returns a JSON string of agent statistics.

   ```
   show mrtable-agent-statistics
           [-agent <agentID>][-table <tableName>][-json]
   ```

   The `show` command with `mrtable-agent-statistics` option shows the latest statistics as of the last one minute for the XRegion agent. With no arguments, this command shows the combined statistics over all regions that the multi-region table spans. You can limit the statistics to a particular agent by specifying the agent id. If a table name is specified in the command, the statistics is limited to a particular multi-region table. To understand more details about using the show command to obtain statistics for a multi-region setup, see show mrtable-agent-statistics.
3. **Display the status of a multi-region table syncing up with remote regions**
   The statistic `lastModificationMs` in the `show mrtable-agent-statistics` command is the timestamp of the last operation performed in each remote region, in milliseconds. By comparing the values of this statistic of the local region and the remote region, you can determine if the remote region has caught up with the local region or still lagging behind.

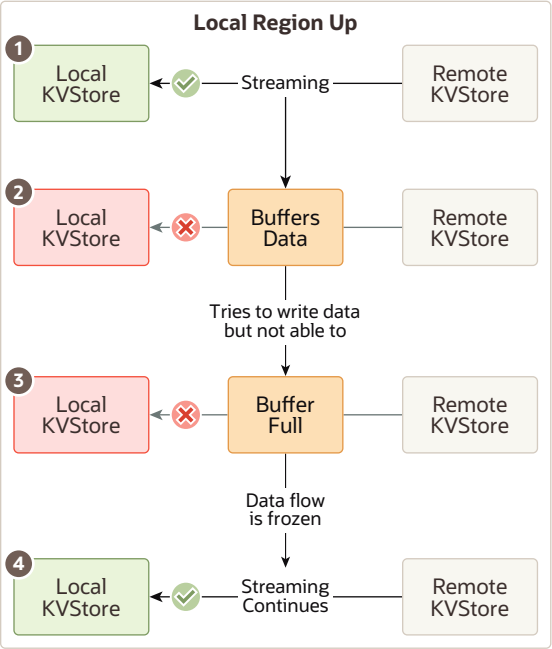
   For example, suppose the time of the last write made to a remote region is T1, while the statistic lastModificationMs for the local region is T2. If T2 < T1, it means that the multi-region table has caught up with that remote region for all writes up to T2 and will continue catching up for all writes made in between T2 and T1. If T2 = T1, that means the multi-region table has caught up with all writes made at the remote region. However T2 can never be greater than T1.

   ```
   # MR table agent statistics for a specific agent
   kv-> show mrtable-agent-statistics -agent 0 -json
   {
      "operation": "show mrtable-agent-statistics",
      "returnCode": 5000,
      "description": "Operation ends successfully",
   ```

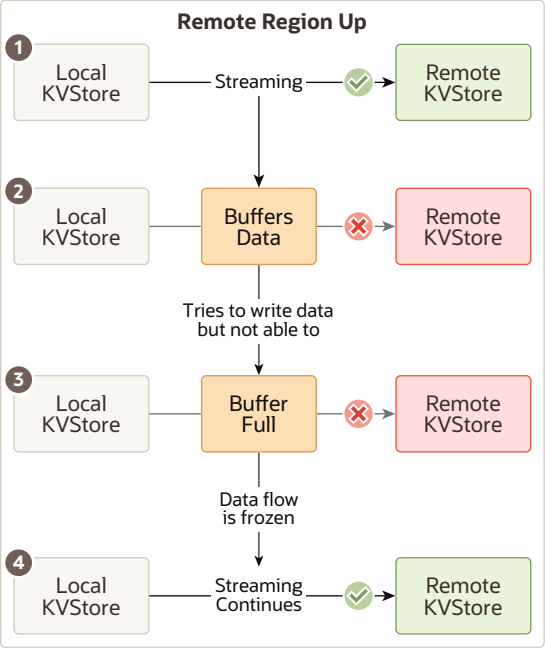
```
    "returnValue": {
       "XRegionService-1_0": {
       "timestamp": 1592901180001,
       "statistics": {
          "agentId": "XRegionService-1_0",
          "beginMs": 1592901120001,
          "dels": 1024,
          "endMs": 1592901180001,
          "incompatibleRows": 100,
          "intervalMs": 60000,
          "localRegion": "slc1",
          "persistStreamBytes": 524288,
          "puts": 2048,
          "regionStat": {
             "lnd": {
                "completeWriteOps": 10,
                "laggingMs": {
                   "avg": 512,
                   "max": 998,
                   "min": 31
                },
                "lastMessageMs": 1591594977587,
                "lastModificationMs": 1591594941686,
                "latencyMs": {
                   "avg": 20,
                   "max": 40,
                   "min": 10
                }
             },
             "dub": {
                "completeWriteOps": 20,
                "laggingMs": {
                   "avg": 535,
                   "max": 1024,
                   "min": 45
                },
                "lastMessageMs": 1591594978254,
                "lastModificationMs": 1591594956786,
                "latencyMs": {
                   "avg": 30,
                   "max": 45,
                   "min": 15
                }
             }
          },
          "requests": 12,
          "responses": 12,
          "streamBytes": 1048576,
          "winDels": 1024,
          "winPuts": 2048
           }
         }
       }
}
```

4. **Troubleshoot problems with XRegion Agent**
   If the XRegion agent encounters a problem, for example if the network connection is dropped, you should investigate the reason of the connection failure and come up with a solution to fix the connection. Meanwhile the XRegion agent would try to re-connect to the remote region until the remote region is up again. After successfully re-connecting to the remote region, the XRegion agent will resume from the stream position or the last checkpoint made, before the connection was dropped. During re-connection, the agent may dump warning messages in the log to alert users that the connection to a region or a shard in that region is lost.

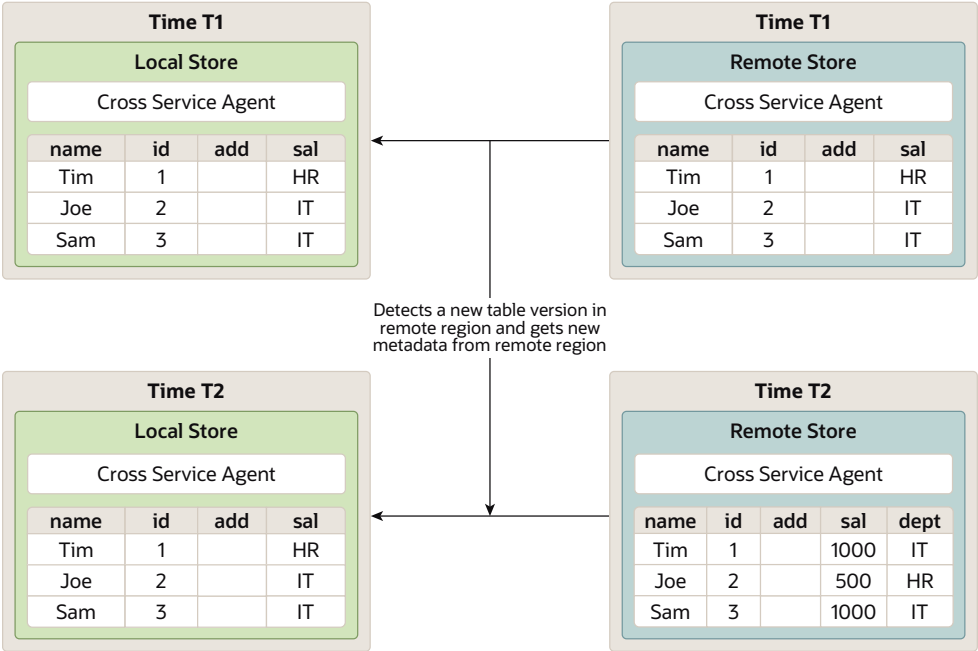
5. **Troubleshoot when the local region or remote region goes down**
   The XRegion agent streams changes to the multi-region table from each remote region and persists them in the local region. Therefore, if the local region is down, the agent will keep retrying but won't be able to write any changes. After a period of time, when the buffer in the XRegion agent is full, the XRegion agent will stop streaming data from the remote regions and the data flow gets frozen. When the local region is back, the XRegion agent will just resume the stream and the workflow. No manual intervention to the XRegion agent is needed here. However you may have to fix the issue with the local region manually.



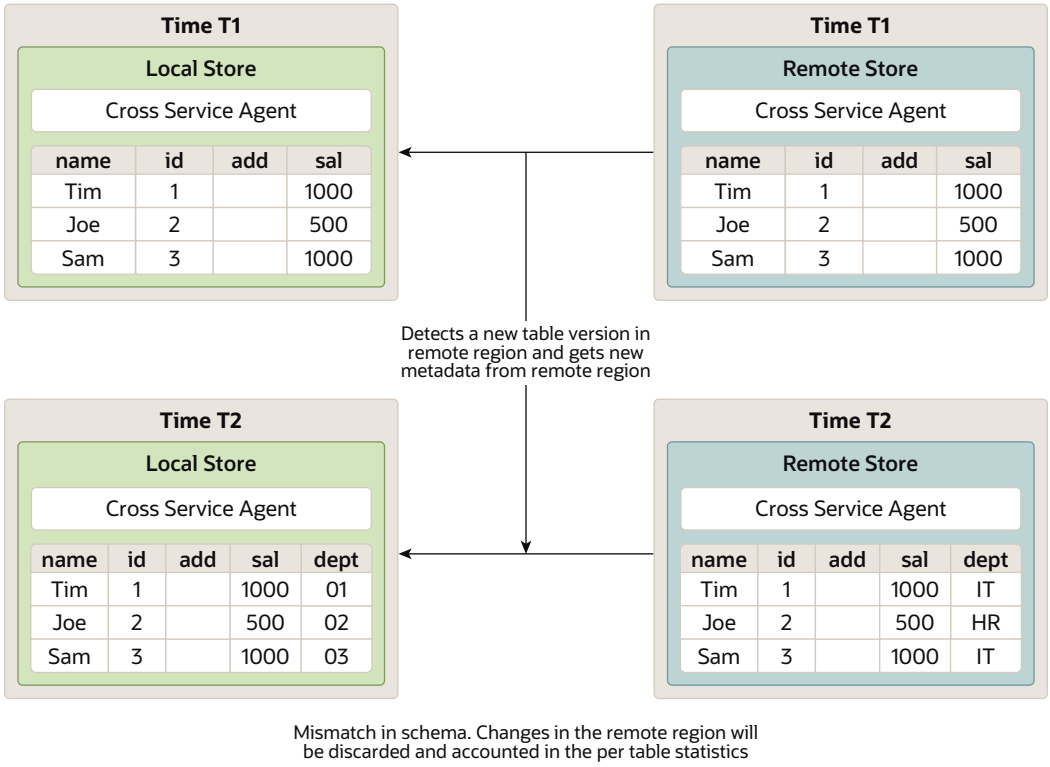
If a particular remote region is down, the XRegion agent will just keep retrying till that remote region is back. This issue is similar to any network connection problem with the XRegion agent. Until the connection to the remote region is established again, the multi-region table at the local region won't be able to see the changes in that remote region. But changes in the other remote regions are not affected as long as the XRegion agent is able to maintain the connection to these regions.



6. **Handle schema evolution in a multi-region setup**
   Schema evolution happens when there is a schema change in any of the remote regions. Then the schema of a multi-region table at the local region differs from that in the remote region. In such a situation the XRegion agent will try to solve the difference by converting a row from the remote region to the schema of local region. For example, if you add a new column to a multi-region table at a remote region but this new column is not yet added in the local region. The multi-region table at the local region will not be able to see the new column in the changes streamed from the remote region, but the local region should still see the other columns. This would last until you fix the problem by adding the same column in the local region to end the schema divergence. In a multi-region table, there is no automatic notification to other regions when a schema changes in one region. The XRegion agent of local region is able to detect the change when it sees the data from a remote region with higher table version, and it will refresh its table metadata from the remote region to get the latest schema.



Consider the situation when the schema in different regions diverge in a way that the agent is not able to fix the schema differences by refreshing the local region table metadata from the remote region. For example, if you add a new column “Foo” with type “STRING” to the remote region but adds the same column with type “LONG” in the local region, these changes at the remote region are considered incompatible to the local region, and the agent cannot fix this difference. These changes from the remote region will not be persisted locally. Consequently the changes in the remote regions will be discarded and accounted in the per-table statistic `incompatibleRows`. See the details about persistence of remote data in the show mrtable-agent-statistics section.



7. **Handle difference in software versions between regions**
   For any particular region, you need to upgrade the data store first and then upgrade the agent to the same version. If a multi-region table has different versions of software on different regions, the agent with old version may not be able to process the rows streamed from regions with a newer version of the software correctly, and some data may be treated by the old agent as incompatible for operations. For example, if the local region is upgraded to support TTL (Time to Live) while the remote region has not yet upgraded, the changes made to the remote region will be persisted to the local region, but without any expiration information, that means the row will never expire. The same is the case if the remote region has upgraded to support TTL while the local region has not. Then all changes to the remote region with TTL will lose their TTL when applied to the local region, which means these rows will never expire. If this is undesirable, you should upgrade all regions first before writing the data to the table to ensure every region can process the data correctly. Any feature will be completely available to a multi-region table only after all the regions have upgraded to the same version.

4
Administer

The articles in this section include various tasks to administer an Oracle NoSQL Database.

Changing the Store's Topology

Topics:

- Determining Your Store's Configuration
- Steps for Changing the Store's Topology
- Deploying an Arbiter Node Enabled Topology

Determining Your Store's Configuration

A store consists of a number of Storage Nodes. Each Storage Node can host one or more Replication Nodes, based on its storage capacity. The term `topology` is used to describe the distribution of Replication Nodes. A topology is derived from the number and capacity of available Storage Nodes, the number of partitions in the store, and the replication factors of the store's zones. Topology layouts are also governed by a set of rules that maximize the availability of the store.

All topologies must adhere to the following rules:

1. Each Replication Node from the same shard must reside on a different Storage Node. This rule prevents a single Storage Node failure causing multiple points of failure for a single shard.
2. The number of Replication Nodes assigned to a Storage Node must be less than or equal to the capacity of Storage Nodes.
3. A zone must have one or more Replication Nodes from each shard.
4. A valid Arbiter Node distribution is one in which the Arbiter Node is hosted on a Storage Node that does not contain other members of its shard.

The store’s initial configuration, or topology, is set when you create the store. Over time, it can be necessary to change the store topology. There are several reasons for such a change:

1. You need to replace or upgrade an existing Storage Node.
2. You need to increase read throughput. This is done by increasing the replication factor and creating more copies of the store's data which can be used to service read only requests.
3. You need to increase write throughput. Since each shard has a single master node, distributing the data in the store over a larger number of shards provides the store with more nodes to execute write operations.

You change the store's configuration by changing the number or capacity of available Storage Nodes, or the replication factor of a zone. To change from one configuration to another, you either create a new initial topology, or `clone` an existing topology and modify it into your target topology. You then deploy this target topology.

> **Note:**
>
> Deploying a target topology can be a lengthy operation. Plus, the time required scales with the amount of data to move. During the deployment, the system updates the topology at each step. Because of that, the store passes through intermediate topologies which you did not explicitly create.

This chapter discusses how to make configuration or topological changes to a store. It also describes how to deploy a topology enabled with Arbiter Nodes.

> **Note:**
>
> Do not make configuration changes while a taking a snapshot, or take a snapshot when changing the configuration. Before making configuration changes, we recommend you first create a snapshot as a backup. For additional information on creating snapshots, see Taking a Snapshot.

Steps for Changing the Store's Topology

When you change your topology, you should go through these steps:

1. Make the Topology Candidate
2. Transforming the Topology Candidate
3. View the Topology Candidate
4. Validate the Topology Candidate
5. Preview the Topology Candidate
6. Deploy the Topology Candidate
7. Verify the Store's Current Topology

Creating a new topology is typically an iterative process, trying different options to see what is best before deploying changes. After trying options, examine the topology candidate and decide if it is satisfactory. If not, apply more transformations, or start over with different parameters. You can view and validate topology candidates to determine if they are appropriate.

The possible transformations to expand the store include redistributing data, increasing the replication factor, and rebalancing. These are described in Transforming the Topology Candidate.

You can also decrease the current topology by removing Storage Nodes. See Contracting a Topology.

The following sections walk you through the process of changing your store’s configuration using the Administration CLI.

Make the Topology Candidate

To create the first topology candidate for an initial deployment, before any Replication Nodes exist, use the `topology create` command. The `topology create` command requires a topology name, a pool name and the number of partitions as arguments.

> **Note:**
>
> Avoid using the dollar sign ($) character in topology candidate names. The CLI displays a warning if you try to create or clone topologies whose names contain the reserved character.

For example:

```
kv-> topology create -name firstTopo -pool BostonPool
-partitions 300
Created: firstTopo
```

Use the `plan deploy-topology` command to deploy this initial topology candidate without further transformations.

After your store is initially deployed, you can create candidate topologies with the `topology clone` command. The source of a clone can be another topology candidate, or the current, deployed topology. The `topology clone` command takes the following arguments:

- `-from <from topology>`

  The name of the source topology candidate.

- `-name <to topology>`

  The name of the clone.

For example:

```
kv-> topology clone -from topo -name CloneTopo
Created CloneTopo
```

This variant of the topology clone command that takes the following arguments:

- `-current`

  Specifies using the current deployed topology as a source, so the argument requires no name.

- `-name <to topology>`

  The name of the topology clone.

For example:

```
kv-> topology clone -current -name ClonedTopo
Created ClonedTopo
```

Transforming the Topology Candidate

After you initially deploy your store, you can change it by deploying another topology candidate that differs from the current topology. This target topology is generated by transforming a topology candidate to expand the store by using these commands:

- `topology redistribute`
- `rebalance`
- `change-repfactor`

Alternatively, you can contract the target topology candidate using the `topology contract` command.
Transformations follow the topology rules described in the previous section.

The topology rebalance, redistribute or change-repfactor commands can only make changes to the topology candidate if there are additional, or changed, Storage Nodes available. It uses the new resources to rearrange Replication Nodes and partitions so the topology complies with the topology rules and the store improves on read or write throughput.

The following are scenarios in how you might expand or contract the store.

Increase Data Distribution

Use the `topology redistribute` command to increase data distribution to enhance write throughput. The redistribute command works only if new Storage Nodes are added to make creating new replication nodes possible for new shards. With new shards, the system distributes partitions across the new shards, resulting in more Replication Nodes to service write operations.

The following example demonstrates adding a set of Storage Nodes (`node04` — `node07`) and redistributing the data to those nodes. Four Storage Nodes are required to meet the zone's replication factor of four and the new shards require four nodes to satisfy the replication requirements:

```
kv-> plan deploy-sn -zn zn1 -host node04 -port 5000 -wait
Executed plan 7, waiting for completion...
Plan 7 ended successfully

kv-> plan deploy-sn -zn zn1 -host node05 -port 5000 -wait
Executed plan 8, waiting for completion...
Plan 8 ended successfully

kv-> plan deploy-sn -zn zn1 -host node06 -port 5000 -wait
Executed plan 9, waiting for completion...
Plan 9 ended successfully

kv-> plan deploy-sn -zn zn1 -host node07 -port 5000 -wait
Executed plan 10, waiting for completion...
Plan 10 ended successfully

kv-> pool join -name BostonPool -sn sn4
Added Storage Node(s) [sn4] to pool BostonPool
kv-> pool join -name BostonPool -sn sn5
```

```
Added Storage Node(s) [sn5] to pool BostonPool
kv-> pool join -name BostonPool -sn sn6
Added Storage Node(s) [sn6] to pool BostonPool
kv-> pool join -name BostonPool -sn sn7
Added Storage Node(s) [sn7] to pool BostonPool

kv-> topology clone -current -name newTopo
Created newTopo

kv-> topology redistribute -name newTopo -pool BostonPool
Redistributed: newTopo

kv-> plan deploy-topology -name newTopo -wait
Executed plan 11, waiting for completion...
Plan 11 ended successfully
```

The redistribute command incorporates the new Storage Node capacity that you added to the BostonPool, and creates new shards. The command also migrates partitions to the new shards. If the number of new shards is less than or equal to the current number of shards, the `topology redistribute` command fails.

> **Note:**
>
> Do not execute the `topology redistribute` command against a store with mixed shards. A mixed shard store has shards whose Replication Nodes are operating with different software versions of Oracle NoSQL Database.

The system goes through these steps when it is redistributing a topology candidate:

1. The topology redistribute command creates new Replication Nodes (RNs) for each shard, assigning the nodes to Storage Nodes according to the topology rules. While creating new RNs, the topology command might move existing RNs to different Storage Nodes, to best use available resources while complying with the topology rules.
2. The topology command distributes Partitions evenly among all shards. The partitions in over populated shards are moved to shards with the least number of partitions.

   You cannot specify which partitions the command moves.

Increase Replication Factor

You can use the `topology change-repfactor` command to increase the replication factor. Increasing the replication factor creates more copies of the data and improves read throughput and availability. More Replication Nodes are added to each shard so that it has the requisite number of nodes. The new Replication Nodes are populated from existing nodes in the shard. Since every shard in a zone has the same replication factor, and a large number of shards, this command may require a significant number of new Storage Nodes to succeed.

For additional information on how to identify your primary replication factor and to understand the implications of the factor value, see Replication Factor.

The following example increases the replication factor of the store to 4. The administrator deploys a new Storage Node and adds it to the Storage Node pool. The admin then clones the existing topology and transforms it to use a new replication factor of 4.

```
kv-> plan deploy-sn -zn zn1 -host node08 -port 5000 -wait
Executed plan 12, waiting for completion...
Plan 12 ended successfully

kv-> pool join -name BostonPool -sn sn8
Added Storage Node(s) [sn8] to pool BostonPool

kv-> topology clone -current -name repTopo
Created repTopo

kv-> topology change-repfactor -name repTopo -pool BostonPool -rf 4 -
zn zn1
Changed replication factor in repTopo

kv-> plan deploy-topology -name repTopo -wait
Executed plan 13, waiting for completion...
Plan 13 ended successfully
```

The `change-repfactor` command fails if either of the following occurs:

1. The new replication factor is less than or equal to the current replication factor.
2. The Storage Nodes specified by the storage node pool do not have enough capacity to host the required new Replication Nodes.

Balance a Non-Compliant Topology

Topologies must obey the rules described in Determining Your Store's Configuration. Changes to the physical characteristics of the store can cause the current store topology to violate those rules. For example, after performance tuning, you want to decrease the capacity of a Storage Node (SN). If that SN is already hosting the maximum permissible number of Replication Nodes, reducing its capacity will make the store non-compliant with the capacity rules. To decrease the capacity of an SN before using the `topology rebalance` command , use the `change-parameters` command for the storage node capacity. See plan change-parameters.

You can balance a non-compliant configuration using the `topology rebalance` command. This command requires a topology candidate name and a Storage Node pool name.

Before rebalancing your topology, use the `topology validate` command for any violations to the topology rules in your `repTopo` plan:

```
kv-> topology validate -name repTopo
Validation for topology candidate "repTopo":
4 warnings.
sn7 has 0 RepNodes and is under its capacity limit of 1
sn8 has 0 RepNodes and is under its capacity limit of 1
sn5 has 0 RepNodes and is under its capacity limit of 1
sn6 has 0 RepNodes and is under its capacity limit of 1
```

In this case, there are anticipated warnings, but you do not need improvements to the topology. However, if improvements are needed, then the `topology rebalance` command will move or create Replication Nodes, using the Storage Nodes in the BostonPool pool, to correct any violations. The command does not create additional shards under any circumstances. See Shard Capacity.

```
kv-> topology rebalance -name repTopo -pool BostonPool
Rebalanced: repTopo
```

If there are insufficient Storage Nodes, or if an insufficient storage directory size is allocated, the `topology rebalance` command may be unable to correct all violations. In that case, the command makes as much progress as possible, and warns of remaining issues.

Contracting a Topology

You can contract a topology by using the `topology contract` command. This command requires a topology candidate name and a Storage Node pool name. This command supports the removal of Storage Nodes and contracts the topology by relocating Replication Nodes, deleting shards, and migrating partitions.

> **Note:**
>
> Decreasing the replication factor is not currently supported. Also, Admin relocation is not supported. If an admin is present on a contracted Storage Node, the contraction operation will fail.

The following example contracts the topology by removing 3 Storage Nodes (sn2, sn5 and sn8). First, you clone the pool using the `pool clone command` and remove the Storage Nodes from the cloned pool using the `pool leave command`. Then, the topology is contracted and deployed using the contracted pool. Finally, the Storage Nodes can be removed using the `plan remove-sn` command. This command automatically stops Storage Nodes before removal.

```
# Clone the existing Storage Node pool as to be contractedPool
kv-> pool clone -name contractedPool -from AllStorageNodes
Cloned pool contractedPool
kv-> pool leave -name contractedPool -sn sn2
Removed Storage Node(s) [sn2] from pool contractedPool
kv-> pool leave -name contractedPool -sn sn5
Removed Storage Node(s) [sn5] from pool contractedPool
kv-> pool leave -name contractedPool -sn sn8
Removed Storage Node(s) [sn8] from pool contractedPool

# Generate a contracted candidate topology
kv-> topology clone -current -name contractedTopology
Created contractedTopology
kv-> topology contract -name contractedTopology -pool contractedPool
Contracted: contractedTopology

# Deploy the contracted candidate topology as the real topology.
kv-> plan deploy-topology -name contractedTopology -wait
```

```
Executed plan 16, waiting for completion...
Plan 16 ended successfully

# Remove to-be-deleted SNs
kv-> plan remove-sn -sn sn2 -wait
Executed plan 17, waiting for completion...
Plan 17 ended successfully
kv-> plan remove-sn -sn sn5 -wait
Executed plan 18, waiting for completion...
Plan 18 ended successfully
kv-> plan remove-sn -sn sn8 -wait
Executed plan 19, waiting for completion...
Plan 19 ended successfully
```

View the Topology Candidate

You can view details of the topology candidate or a deployed topology by using the `topology view` command. The command takes a topology name as an argument. With the topology view command, you can view all at once: the store name, number of partitions, shards, replication factor, host name and capacity in the specified topology.

For example:

```
kv-> topology view -name repTopo
store=mystore  numPartitions=300 sequence=315
  zn: id=zn1 name=Boston repFactor=4 type=PRIMARY

  sn=[sn1]  zn:[id=zn1 name=Boston] node01:5000 capacity=1
    [rg1-rn1]
  sn=[sn2]  zn:[id=zn1 name=Boston] node02:5000 capacity=1
    [rg1-rn2]
  sn=[sn3]  zn:[id=zn1 name=Boston] node03:5000 capacity=1
    [rg1-rn3]
  sn=[sn4]  zn:[id=zn1 name=Boston] node04:5000 capacity=1
    [rg1-rn4]
  sn=[sn5]  zn:[id=zn1 name=Boston] node05:5000 capacity=1
  sn=[sn6]  zn:[id=zn1 name=Boston] node06:5000 capacity=1
  sn=[sn7]  zn:[id=zn1 name=Boston] node07:5000 capacity=1
  sn=[sn8]  zn:[id=zn1 name=Boston] node08:5000 capacity=1

  shard=[rg1] num partitions=300
    [rg1-rn1] sn=sn1
    [rg1-rn2] sn=sn2
    [rg1-rn3] sn=sn3
    [rg1-rn4] sn=sn4
```

Validate the Topology Candidate

You can validate the topology candidate or a deployed topology by using the `topology validate` command. The topology validate command takes a topology name as an argument. If no topology is specified, the current topology is validated. Validation makes sure that the topology candidate obeys the topology rules described in Determining Your Store's Configuration. Validation generates "violations" and "notes".

Violations are issues that can cause problems and should be investigated.

Notes are informational and highlight configuration oddities that may be potential issues, but may be expected.

For example:

```
kv-> topology validate -name repTopo
Validation for topology candidate "repTopo":
4 warnings.
sn7 has 0 RepNodes and is under its capacity limit of 1
sn8 has 0 RepNodes and is under its capacity limit of 1
sn5 has 0 RepNodes and is under its capacity limit of 1
sn6 has 0 RepNodes and is under its capacity limit of 1
```

Preview the Topology Candidate

You should preview the changes that would be made for the specified topology candidate relative to a starting topology. You use the `topology preview` command to do this. This command takes the following arguments:

- **name**

  A string to identify the topology.

- **start <from topology>**

  If -start topology name is not specified, the current topology is used. This command should be used before deploying a new topology.

For example:

```
kv-> topology clone -current -name redTopo
Created redTopo
kv-> topology redistribute -name redTopo -pool BostonPool
Redistributed: redTopo
kv-> topology preview -name redTopo
Topology transformation from current deployed topology to redTopo:
Create 1 shard
Create 4 RNs
Migrate 150 partitions

shard rg2
  4 new RNs: rg2-rn1 rg2-rn2 rg2-rn3 rg2-rn4
  150 partition migrations
kv-> topology validate -name redTopo
Validation for topology candidate "redTopo":
No problems
```

Deploy the Topology Candidate

When your topology candidate is satisfactory, use the Admin service to generate and execute a plan that migrates the store to the new topology.

Deploy the topology candidate with the `plan deploy-topology` command, which takes a topology name as an argument.

While the plan is executing, you can monitor the plan's progress. You have several options:

- The plan can be interrupted then retried, or canceled.
- Other, limited plans may be executed while a transformation plan is in progress to deal with ongoing problems or failures.

By default, the `plan deploy-topology` command will not deploy a topology candidate if deployment would introduce new violations of the topology rules. You can override this behavior using the optional `-force` plan flag. Do not use the `-force` plan without consideration. Introducing a topology rule violation can have many adverse effects.

The next example shows the topology differences before and after plan deployment. The first `show topology` output lists four Storage Nodes running in Zone 1, with one shard (`rg1`) storing 300 partitions. Storage nodes `sn5 –sn8` are available.

After deploying the plan, the `show topology` output lists storage nodes `sn5 – sn8` as running. Another shard exists (`rg2`), and the partitions are split between the two shards, each with 150 partitions.

```
kv-> show topology
store=mystore  numPartitions=300 sequence=315
  zn: id=zn1 name=Boston repFactor=4 type=PRIMARY

  sn=[sn1]  zn=[id=zn1 name=Boston] node01:5000 capacity=1 RUNNING
    [rg1-rn1] RUNNING
           No performance info available
  sn=[sn2]  zn=[id=zn1 name=Boston] node02:5000 capacity=1 RUNNING
    [rg1-rn2] RUNNING
           No performance info available
  sn=[sn3]  zn=[id=zn1 name=Boston] node03:5000 capacity=1 RUNNING
    [rg1-rn3] RUNNING
           No performance info available
  sn=[sn4]  zn=[id=zn1 name=Boston] node04:5000 capacity=1 RUNNING
    [rg1-rn4] RUNNING
           No performance info available
  sn=[sn5]  zn=[id=zn1 name=Boston] node05:5000 capacity=1
  sn=[sn6]  zn=[id=zn1 name=Boston] node06:5000 capacity=1
  sn=[sn7]  zn=[id=zn1 name=Boston] node07:5000 capacity=1
  sn=[sn8]  zn=[id=zn1 name=Boston] node08:5000 capacity=1

  shard=[rg1] num partitions=300
    [rg1-rn1] sn=sn1
    [rg1-rn2] sn=sn2
    [rg1-rn3] sn=sn3
    [rg1-rn4] sn=sn4

kv-> plan deploy-topology -name redTopo -wait
Executed plan 14, waiting for completion...
Plan 14 ended successfully

kv-> show topology
store=mystore  numPartitions=300 sequence=470
  zn: id=zn1 name=Boston repFactor=4 type=PRIMARY
```

```
  sn=[sn1]  zn:[id=zn1 name=Boston] node01:5000 capacity=1 RUNNING
    [rg1-rn1] RUNNING
          No performance info available
  sn=[sn2]  zn:[id=zn1 name=Boston] node02:5000 capacity=1 RUNNING
    [rg1-rn2] RUNNING
          No performance info available
  sn=[sn3]  zn:[id=zn1 name=Boston] node03:5000 capacity=1 RUNNING
    [rg1-rn3] RUNNING
          No performance info available
  sn=[sn4]  zn:[id=zn1 name=Boston] node04:5000 capacity=1 RUNNING
    [rg1-rn4] RUNNING
          No performance info available
  sn=[sn5]  zn:[id=zn1 name=Boston] node05:5000 capacity=1 RUNNING
    [rg2-rn1] RUNNING
          No performance info available
  sn=[sn6]  zn:[id=zn1 name=Boston] node06:5000 capacity=1 RUNNING
    [rg2-rn2] RUNNING
          No performance info available
  sn=[sn7]  zn:[id=zn1 name=Boston] node07:5000 capacity=1 RUNNING
    [rg2-rn3] RUNNING
          No performance info available
  sn=[sn8]  zn:[id=zn1 name=Boston] node08:5000 capacity=1 RUNNING
    [rg2-rn4] RUNNING
          No performance info available

  shard=[rg1] num partitions=150
    [rg1-rn1] sn=sn1
    [rg1-rn2] sn=sn2
    [rg1-rn3] sn=sn3
    [rg1-rn4] sn=sn4
  shard=[rg2] num partitions=150
    [rg2-rn1] sn=sn5
    [rg2-rn2] sn=sn6
    [rg2-rn3] sn=sn7
    [rg2-rn4] sn=sn8
```

Verify the Store's Current Topology

You can verify the store's current topology by using the `verify` command. The verify command checks the current, deployed topology to make sure it adheres to the topology rules described in Determining Your Store's Configuration.

You should examine the new topology and decide if it is satisfactory. If it is not, you can apply more transformations, or start over with different parameters.

For example:

```
kv-> verify configuration
Verify: starting verification of store mystore based upon
    topology sequence #470
300 partitions and 8 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
See localhost:KVROOT/mystore/log/mystore_{0..N}.log for progress messages
Verify: Shard Status: healthy:2 writable-degraded:0 read-only:0 offline:0
Verify: Admin Status: healthy
```

```
Verify: Zone [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    RN Status: online:8 offline:0 maxDelayMillis:0 maxCatchupTimeSecs:0
Verify: == checking storage node sn1 ==
Verify: Storage Node [sn1] on node01:5000
    Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING
    Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin1]          Status: RUNNING,MASTER
Verify:         Rep Node [rg1-rn1]      Status: RUNNING,MASTER ...
Verify: == checking storage node sn2 ==
Verify: Storage Node [sn2] on node02:5000
    Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING
    Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg1-rn2]      Status: RUNNING,REPLICA ...
Verify: == checking storage node sn3 ==
Verify: Storage Node [sn3] on node03:5000
    Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING
    Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg1-rn3]      Status: RUNNING,REPLICA ...
Verify: == checking storage node sn4 ==
Verify: Storage Node [sn4] on node04:5000
    Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING
    Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg1-rn4]      Status: RUNNING,REPLICA ...
Verify: == checking storage node sn5 ==
Verify: Storage Node [sn5] on node05:5000
    Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING
    Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn1]      Status: RUNNING,MASTER ...
Verify: == checking storage node sn6 ==
Verify: Storage Node [sn6] on node06:5000
    Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING
    Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn2]      Status: RUNNING,REPLICA ...
Verify: == checking storage node sn7 ==
Verify: Storage Node [sn7] on node07:5000
    Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING
    Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn3]      Status: RUNNING,REPLICA ...
Verify: == checking storage node sn8 ==
Verify: Storage Node [sn8] on node08:5000
```

```
    Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING
    Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn4]      Status: RUNNING,REPLICA ...
Verification complete, no violations.
```

Deploying an Arbiter Node Enabled Topology

An `Arbiter Node` is a service that supports write availability when the store replication factor is two and a single Replication Node becomes unavailable. The role of an `Arbiter Node` is to participate in elections and respond to acknowledge requests if one of the two Replication Nodes in a shard becomes unavailable.

Arbiter Nodes are automatically configured in a topology if the store replication factor is two and a primary zone is configured to host `Arbiter Nodes`.

For example, suppose a store consists of a primary zone, "Manhattan" with two Storage Nodes deployed in the same shard. In this example, an Arbiter Node is deployed in the third Storage Node (capacity = 0) in order to provide write availability even if one of the two Replication Nodes in the shard becomes unavailable.

> **Note:**
>
> Durability.ReplicaAckPolicy must be set to SIMPLE_MAJORITY, so that writes can succeed if a Replication Node becomes unavailable in a shard. For more information on `ReplicaAckPolicy`, see this Javadoc page.

1. Create, start, and configure the store. Note that a Storage Node with capacity equal to zero is deployed, which will host the Arbiter Node.
   - Create the store:

     ```
     java -Xmx64m -Xms64m \
     -jar kv/lib/kvstore.jar makebootconfig \
     -root KVROOT \
     -host node01 \
     -port 8000 \
     -harange 8010,8020 \
     -capacity 1

     java -Xmx64m -Xms64m \
     -jar kv/lib/kvstore.jar makebootconfig \
     -root KVROOT \
     -host node02 \
     -port 9000 \
     -harange 9010,9020 \
     -capacity 1

     java -Xmx64m -Xms64m \
     -jar kv/lib/kvstore.jar makebootconfig \
     -root KVROOT \
     ```

```
-host node03 \
-port 10000 \
-harange 1000,10020 \
-capacity 0 \
```

- Create and copy the security directories:

```
java -Xmx64m -Xms64m \
-jar kv/lib/kvstore.jar
securityconfig \
config create -root KVROOT -kspwd password
Created files
KVROOT/security/security.xml
KVROOT/security/store.keys
KVROOT/security/store.trust
KVROOT/security/client.trust
KVROOT/security/client.security
KVROOT/security/store.passwd (Generated in CE version)
KVROOT/security/store.wallet/cwallet.sso (Generated in EE
version)

Created
scp -r KVROOT/security node02:KVROOT/
scp -r KVROOT/security node03:KVROOT/
```

- Start the store by running the following command on each Storage Node:

```
java -Xmx64m -Xms64m -jar KVHOME/lib/kvstore.jar \
start -root KVROOT &
```

2. Load the following script `conf.txt` to deploy the zone, admin and Storage Nodes. To host an Arbiter Node, the zone must be primary and should have the `-arbiters` flag set.

```
ssh node01
java -Xmx64m -Xms64m -jar KVHOME/lib/kvstore.jar runadmin \
-port 8000 -host node01 load -file conf.txt \
-security KVROOT/security/client.security
```

The file, `conf.txt`, would then contain content like this:

```
### Begin Script ###
plan deploy-zone -name "Manhattan" -type primary -arbiters -rf 2 -
wait
plan deploy-sn -zn zn1 -host node01 -port 8000 -wait
pool create -name SNs
pool join -name SNs -sn sn1
plan deploy-admin -sn sn1 -port 8001 -wait
plan deploy-sn -zn zn1 -host node02 -port 9000 -wait
pool join -name SNs -sn sn2
plan deploy-sn -zn zn1 -host node03 -port 10000 -wait
pool join -name SNs -sn sn3
### End Script ###
```

3. Create a topology, preview it, and then deploy it:

```
kv-> topology create -name arbTopo -pool SNs -partitions 300
Created: arbTopo


kv-> topology preview -name arbTopo
Topology transformation from current deployed topology to arbTopo:
Create 1 shard
Create 2 RNs
Create 300 partitions
Create 1 AN

shard rg1
  2 new RNs : rg1-rn1 rg1-rn2
  1 new AN : rg1-an1
  300 new partitions


kv-> plan deploy-topology -name arbTopo -wait
Executed plan 6, waiting for completion...
Plan 6 ended successfully
```

4. Verify that the Arbiter Node is running.

```
kv-> verify configuration
Verify: starting verification of store mystore
based upon topology sequence #308
300 partitions and 3 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
See node01:KVROOT/mystore/log/mystore_{0..N}.log
for progress messages
Verify: Shard Status: healthy:1 writable-degraded:0
                                              read-only:0 offline:0
Verify: Admin Status: healthy
Verify: Zone [name=Manhattan id=zn1 type=PRIMARY allowArbiters=true
masterAffinity=false]
RN Status: online:2 offline:0 maxDelayMillis:6 maxCatchupTimeSecs:0
Verify: == checking storage node sn1 ==
Verify: Storage Node [sn1] on node01:8000
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=true
masterAffinity=false]
Status: RUNNING
Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin1]          Status: RUNNING,MASTER
Verify:         Rep Node [rg1-rn1]
Status: RUNNING,MASTER sequenceNumber:635 haPort:8011 available storage
size:11 GB
Verify: == checking storage node sn2 ==
Verify: Storage Node [sn2] on node02:9000
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=true
masterAffinity=false]
Status: RUNNING
Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg1-rn2]
```

```
Status: RUNNING,REPLICA
sequenceNumber:635 haPort:9010 available storage size:12 GB
delayMillis:6 catchupTimeSecs:0
Verify: == checking storage node sn3 ==
Verify: Storage Node [sn3] on node03:10000
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=true
masterAffinity=false]
Status: RUNNING
Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Arb Node [rg1-an1]
Status: RUNNING,REPLICA sequenceNumber:0 haPort:node03:10010
...
```

5. Now suppose node02 is unreachable. Verify this by using `verify configuration`:

```
kv-> verify configuration
Verify: starting verification of store mystore
based upon topology sequence #308
300 partitions and 3 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
See node01:KVROOT/mystore/log/mystore_{0..N}.log
for progress messages
Verify: Shard Status: healthy:0 writable-degraded:1
                                                 read-only:0 offline:0
Verify: Admin Status: healthy
Verify:
      Zone [name=Manhattan id=zn1 type=PRIMARY allowArbiters=true
masterAffinity=false]
RN Status: online:1 offline:1
Verify: == checking storage node sn1 ==
Verify: Storage Node [sn1] on node01:8000
Zone:
    [name=Manhattan id=zn1 type=PRIMARY allowArbiters=true
masterAffinity=false]
 Status: RUNNING
 Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin1]          Status: RUNNING,MASTER
Verify:         Rep Node [rg1-rn1]
Status: RUNNING,MASTER sequenceNumber:901 haPort:8011 available
storage size:12 GB
Verify: == checking storage node sn2 ==
Verify:         sn2: ping() failed for sn2 : Unable to connect to
the storage node agent at host node02, port 9000, which may not be
running; nested exception is:
        java.rmi.ConnectException: Connection refused to
        host: node02; nested exception is:
        java.net.ConnectException: Connection refused
Verify: Storage Node [sn2] on node02:9000
Zone:
    [name=Manhattan id=zn1 type=PRIMARY allowArbiters=true
masterAffinity=false] UNREACHABLE
Verify:         rg1-rn2: ping() failed for rg1-rn2 : Unable to
connect
to the storage node agent at host node02, port 9000, which may not
be running; nested exception is:
```

```
        java.rmi.ConnectException: Connection refused to host: node02;
        nested exception is:
        java.net.ConnectException: Connection refused
Verify:         Rep Node [rg1-rn2]      Status: UNREACHABLE
Verify: == checking storage node sn3 ==
Verify: Storage Node [sn3] on node03:10000
 Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=true
masterAffinity=false]
 Status: RUNNING
Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Arb Node [rg1-an1]
Status: RUNNING,REPLICA sequenceNumber:901 haPort:node03:10010 available
storage size:16 GB delayMillis:? catchupTimeSecs:?
Verification complete, 3 violations, 0 notes found.
Verification violation: [rg1-rn2]
ping() failed for rg1-rn2 : Unable to connect to the storage node
agent at host node02, port 9000, which may not be running;
nested exception is:
        java.rmi.ConnectException: Connection refused to
        host: node02; nested exception is:
        java.net.ConnectException: Connection refused
Verification violation: [sn2]   ping() failed for sn2 : Unable to
connect to the storage node agent at host node02, port 9000, which
may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host: node02;
nested exception is:
        java.net.ConnectException: Connection refused
...
```

In this case the Arbiter Node supports write availability so you can still perform write operations while node02 is repaired or replaced. Once node02 is restored, any written data will be migrated.

6. Test that you can still write to the store with the help of the Arbiter Node. For example, run the script file `test.kvsql` (see below for test.kvsql) using the Oracle NoSQL Database Shell utility (see below example). To do this, use the `load` command in the Query Shell:

```
> java -jar KVHOME/lib/sql.jar -helper-hosts node01:8000 \
-store mystore -security USER/security/admin.security
kvsql-> load -file ./test.kvsql
Statement completed successfully.
Statement completed successfully.
Loaded 3 rows to users.
```

> **Note:**
>
> For the Enterprise Edition (EE) installation, make sure the kvstore-ee.jar is added in the classpath.

The following commands are collected in `test.kvsql`:

```
### Begin Script ###
load -file test.ddl
import -table users -file users.json
### End Script ###
```

Where the file `test.ddl` would contain content like this:

```
DROP TABLE IF EXISTS users;
CREATE TABLE users(id INTEGER, firstname STRING, lastname STRING,
age INTEGER, primary key (id));
```

And the file `users.json` would contain content like this:

```
{"id":1,"firstname":"Dean","lastname":"Morrison","age":51}
{"id":2,"firstname":"Idona","lastname":"Roman","age":36}
{"id":3,"firstname":"Bruno","lastname":"Nunez","age":49}
```

Backup and Recovery

Topics:

- Backing Up the Store
- Recovering the Store
- Recovering from Data Corruption
- Replacing a Failed Disk
- Replacing a Failed Storage Node
- Repairing a Failed Zone by Replacing Hardware

Backing Up the Store

To make backups of your KVStore, use the CLI `snapshot` command to copy nodes in the store. To maintain consistency, no topology changes should be in process when you create a snapshot. Restoring a snapshot relies on the system configuration having exactly the same topology that was in effect when you created the snapshot.

When you create a snapshot, it is stored in a subdirectory of the SN. But these snapshots don't become persistent backups unless they are copied to separate storage. It is your responsibility to copy each of the snapshots to another location, preferably on a different machine, for data safety.

Due to the distributed nature and scale of Oracle NoSQL Database, it is unlikely that a single machine has the resources to contain snapshots for the entire store. This document does not address where and how you should store your snapshots.

Taking a Snapshot

> **Note:**
>
> To avoid any snapshot from being inconsistent or unusable, do not take snapshots while any configuration (topological) changes are in process. At the time of the snapshot, use the `ping` command and save the output information that identifies Masters for later use during a load or restore. For more information, see Managing Snapshots.

To create a snapshot from the Admin CLI, use the `snapshot create` command:

```
kv-> snapshot create -name <snapshot name>
```

A snapshot consists of a set of hard links to data files in the current topology, specifically, all partition records within the same shard. The snapshot does not include partitions in independent shards. To minimize any potential inconsistencies, the snapshot utility performs its operations in parallel as much as possible.

To create a snapshot with a name of your choice, use `snapshot create -name <name>`.

```
kv-> snapshot create -name Thursday
Created snapshot named 110915-153514-Thursday on all 3 nodes
Successfully backup configurations on sn1, sn2, sn3
```

Snapshot Activities

Creating a snapshot of the Oracle NoSQL Database store performs these activities:

- Backs up the data files
- Backs up the configuration and environment files required for restore activities

To complete a full set of snapshot files, the `snapshot` command attempts to backup the storage node data files, configuration files, and adds other required files. Following is a description of the various files and directories the `snapshot` command creates or copies:

Creates a `snapshots` directory as a peer to the `env` directory. Each `snapshots` directory contains one subdirectory for each snapshot you create. That subdirectory contains the `*.jdb` files.
 	
The snapshot name subdirectory with a *date-time-name* prefix has the name you supply with the `-name` parameter. The *date-time* prefix consists of a 6-digit, year, month, day value in YYMMDD format, and a 6-digit hour, minute, seconds timestamp as HHMMSS.
	`kvroot/mystore/sn1/rg1-rn1/snapshots/170417-104506-snapshotName/*.jdb`
	`kvroot/mystore/sn1/rg1-rn1/env/*.jdb`
	`kvroot/mystore/sn1/admin1/snapshots/170417-104506-snapshotName/*.jdb`
	`kvroot/mystore/sn1/admin1/env/*.jdb`

The date and time values are separated from each other with a dash (-), and include a dash (-) suffix before the snapshot name.	
Copies the root `config.xml` file to the *date-time-name* directory.	`kvroot/config.xml > kvroot/snapshots/170417-104506-snapshotName/config.xml`
Creates a status file in the *date-time-name* subdirectory. The contents of this file, `snapshot.stat`, indicate whether creating a snapshot was successful. When you restore to a snapshot, the procedure first validates the status file contents, continuing only if the file contains the string `SNAPSHOT=COMPLETED`.	`kvroot/snapshots/170417-104506-snapshotName/snapshot.stat`
Creates a lock file in the *date-time-name* subdirectory. The lock file, `snapshot.lck`, is used to avoid concurrent modifications from different SN Admins within the same root directory.	`kvroot/snapshots/170417-104506-snapshotName/snapshot.lck`
Creates a subdirectory of the *date-time-name* subdirectory, `security`. This subdirectory has copies of security information copied from `kvroot/security`.	`kvroot/snapshots/170417-104506-snapshotName/security`
Copies the root security policy from `kvroot/security.policy`, to the *date-time-name* subdirectory.	`kvroot/snapshots/170417-104506-snapshotName/security.policy`
Copies the store security policy to *date-time-name* subdirectory, into another subdirectory, `mystore`.	`kvroot/snapshots/170417-104506-snapshotName/mystore/security.policy`
Copies the Storage Node configuration file, `config.xml`, from `kvroot/mystore/sn1/config.xml` to a corresponding SN subdirectory in the *date-time-name* directory.	`kvroot/snapshots/170417-104506-snapshotName/mystore/sn1/config.xml`

Copying a Snapshot

Keeping a snapshot in place for a short time so that it can be used to rollback the store after an upgrade is a reasonable thing to do. In such a scenario, it might be sufficient to delete the snapshot without copying it if the upgrade can be verified relatively quickly and the snapshot is no longer needed.

For ensuring data safety during disk or hardware failures, it is recommended that you convert these snapshots into persistent backups. Otherwise, if the machine suffers a disk or other hardware failure, or if store files are deleted or overwritten, the snapshot will be lost along with the live data for the store maintained on that machine.

To convert the snapshot into a persistent backup, the snapshot needs to be copied to another location on a different machine. Later, you can use the persistent backup to restore the store after a disk or hardware failure.

Deleting a Snapshot

To remove an existing snapshot, use `snapshot remove <name>`.

```
kv-> snapshot remove -name 110915-153514-Thursday
Removed snapshot 110915-153514-Thursday
```

To remove all snapshots currently stored in the store, use `snapshot remove -all`.

```
kv-> snapshot create -name Thursday
Created snapshot named 110915-153700-Thursday on all 3 nodes
kv-> snapshot create -name later
Created snapshot named 110915-153710-later on all 3 nodes
kv-> snapshot remove -all
Removed all snapshots
```

Managing Snapshots

When you create a snapshot, the utility collects data from every Replication Node in the system, including Masters and replicas. If the operation does not succeed for any one node in a shard, the entire snapshot fails.

When you are preparing to take the snapshot, you can use the `ping` command to identify which nodes are currently running as the Master. Each shard has a Master, identified by the `MASTER` keyword. For example, in the sample output, replication node `rg1-rn1`, running on Storage Node `sn1`, is the current Master:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar ping -port 5000 -host node01 \
-security USER/security/admin/security
Pinging components of store mystore based upon topology sequence #316
300 partitions and 3 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
Shard Status: healthy:3 writable-degraded:0 read-only:0 offline:0
Admin Status: healthy
Zone [name=Boston id=zn1 type=PRIMARY allowArbiters=false
```

```
masterAffinity=false]
RN Status: online:9 offline:0 maxDelayMillis:1 maxCatchupTimeSecs:0
Storage Node [sn1] on node01:5000
   Zone: [name=Boston id=zn1 type=PRIMARY]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
      Admin [admin1]          Status: RUNNING,MASTER
      Rep Node [rg1-rn1]      Status: RUNNING,REPLICA
        sequenceNumber:231 haPort:5011 available storage size:14 GB
delayMillis:1 catchupTimeSecs:0
      Rep Node [rg2-rn1]      Status: RUNNING,REPLICA
        sequenceNumber:231 haPort:5012 available storage size:12 GB
delayMillis:1 catchupTimeSecs:0
      Rep Node [rg3-rn1]      Status: RUNNING,MASTER
        sequenceNumber:227 haPort:5013 available storage size:13 GB
Storage Node [sn2] on node02:6000
   Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
      Rep Node [rg1-rn2]      Status: RUNNING,MASTER
        sequenceNumber:231 haPort:6010 available storage size:15 GB
      Rep Node [rg2-rn2]      Status: RUNNING,REPLICA
        sequenceNumber:231 haPort:6011 available storage size:18 GB
delayMillis:1 catchupTimeSecs:0
      Rep Node [rg3-rn2]      Status: RUNNING,REPLICA
        sequenceNumber:227 haPort:6012 available storage size:12 GB
delayMillis:1 catchupTimeSecs:0
Storage Node [sn3] on node03:7000
   Zone: [name=Boston id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
      Rep Node [rg1-rn3]      Status: RUNNING,REPLICA
        sequenceNumber:231 haPort:7010 available storage size:11 GB
delayMillis:1 catchupTimeSecs:0
      Rep Node [rg2-rn3]      Status: RUNNING,MASTER
        sequenceNumber:231 haPort:7011 available storage size:11 GB
      Rep Node [rg3-rn3]      Status: RUNNING,REPLICA
        sequenceNumber:227 haPort:7012 available storage size:10 GB
delayMillis:1 catchupTimeSecs:0
```

You should save the above information and associate it with the respective snapshot, for later use during a load or restore. If you decide to create an off-store copy of the snapshot, you should copy the snapshot data for only one of the nodes in each shard. If possible, copy the snapshot data taken from the node that was serving as the Master at the time the snapshot was taken.

> **Note:**
>
> Snapshots include the admin database, which may be required if the store needs to be restored from this snapshot.

Snapshot data for the local Storage Node is stored in a directory inside of the `KVROOT` directory. For each Storage Node in the store, you have a directory named:

```
KVROOT/<store>/<SN>/<resource>/snapshots/<snapshot_name>/files
```

where:

- <store> is the name of the store.
- <SN> is the name of the Storage Node.
- <resource> is the name of the resource running on the Storage Node. Typically, this is the name of a replication node.
- <snapshot_name> is the name of the snapshot.

Snapshot data consists of a number of files. For example:

```
 > ls /var/kvroot/mystore/sn1/rg1-rn1/snapshots/110915-153514-Thursday
00000000.jdb 00000002.jdb 00000004.jdb 00000006.jdb
00000001.jdb 00000003.jdb 00000005.jdb 00000007.jdb
```

> **Note:**
>
> To preserve storage, purge obsolete snapshots on a periodic basis.

Impact of Erasure with snapshots

Snapshot based backups create hard-links to original files. Until these backups are copied to their target location (complete off-store copy work) and the corresponding hard-links are removed (performing a snapshot remove command), erasure doesn't process obsolete data in those files. Erasure ignores files with hard-links to them.

Avoiding Disk Usage Violation

The storage engine does not consider the data consumed by snapshots when it collects information about disk space usage. Initially, the files in the snapshot are considered to be part of the live data of the store. Over time, though, as older files are cleaned and deleted, their presence in the snapshot causes the files to be retained and use the disk space that is not taken into account by the storage engine. It could cause a disk usage violation, in which case further writes to the store are disabled. To avoid this problem, users should delete snapshot files at regular intervals.

Recovering the Store

There are two ways to recover your store from a previously created snapshot:

1. Use a snapshot to create a store with any topology with the `Load` utility.
2. Restore a snapshot using the exact topology you were using when you created the snapshot.

This section describes and explains both ways to recover your store.

> **Note:**
>
> If you need to recover due to a hardware problem, such as a failed Storage Node, that qualifies as a topology change, so you must use the `Load` utility to recover. For information about replacing a failed Storage Node, see Replacing a Failed Storage Node.

Using the Load Program

You can use the `oracle.kv.util.Load` program to restore a store from a previously created snapshot. You can run this program directly, or you can access it using `kvstore.jar`, as shown in the examples in this section.

Using this tool lets you restore to any topology, not just the topology in effect when you created the snapshot.

This Load mechanism works by iterating through all records in a snapshot, putting each record into a target store as it proceeds through the snapshot. Use Load to populate a new, empty store. Do not use this with an existing store. Load only writes records if they do not already exist.

Note that to recover the store, you must load records from snapshot data captured for each shard in the store. For best results, you should load records using snapshot data captured from the replication nodes that were running as Master at the time the snapshot was taken. (If you have three shards in your store, then there are three Masters at any given time, and so you need to load data from three sets of snapshot data). To identify the Master, use ping at the time the snapshot was taken.

You should use snapshot data taken at the same point in time; do not, for example, use snapshot data for shard 1 that was taken on Monday, and snapshot data for shard 2 that was taken on Wednesday. Such actions will restore your store to an inconsistent state.

Also, the Load mechanism can only process data at the speed necessary to insert data into a new store. Because you probably have multiple shards in your store, you should restore your store from data taken from each shard. To do this, run multiple instances of the `Load` program in parallel, having each instance operate on data from different replication nodes.

The program's usage to load admin metadata is:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar load \
-store <storeName> -host <hostname> port <port>  \
-load-admin \
-source <admin-backup-dir> \
[-force] [-username <user>] \
[-security <security-file-path>]
```

The program's usage to load store data is:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar load [-verbose] \
```

```
-store <storeName> -host <hostname> \
-port <port> -source <shard-backup-dir> \
[, <shard-backup-dir>]* \
[-checkpoint <checkpoint-files-directory>] \
[-username <user>] [-security <security-file-path>]
```

where:

- `-load-admin` Loads the store metadata from the snapshot to the new store. In this case the `-source` directory must point to the environment directory of the admin node from the snapshot. The store must not be available for use by users at the time of this operation.

  > **Note:**
  >
  > This option should not be used on a store unless that store is being restored from scratch. If `-force` is specified in conjunction with `-load-admin`, any existing metadata in the store, including tables and security metadata, will be overwritten. For more information, see Load Program and Metadata.

- `-host <hostname>` identifies the host name of a node in your store.
- `-port <port>` identifies the registry port in use by the store's node.
- `-security <security-file-path>` identifies the security file used to specify properties for login.
- `-source <admin-backup-dir> | <shard-backup-dir> [,<shard-backup-dir>]*` `admin-backup-dir` specifies the admin snapshot directory containing the contents of the admin metadata that is to be loaded into the store.

  `Shard-backup-dir` specifies the backup directories that represent the contents of snapshots created using the snapshot commands described at Taking a Snapshot.
- `-store <storeName>` identifies the name of the store.
- `-username <user>` identifies the name of the user to login to the secured store.

For example, if a snapshot exists in `/var/backups/snapshots/110915-153828-later`, and a new store named "mystore" on host "host1" using registry port 5000, run the `Load` program on the host that has the `/var/backups/snapshots` directory:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar load \
-source /var/backups/snapshots/110915-153514-Thursday -store mystore \
-host host1 -port 5000 -security KVROOT/security/client.security
```

> **Note:**
>
> Before you load records into the new store, make sure that the store is deployed. For more information, see Configuring a single region data store.

Load Program and Metadata

You can use the Load program to restore a store with metadata (tables, security) from a previously created snapshot.

The following steps describe how to load from a snapshot with metadata to a newly created store:

1. Create, start and configure the new store (target). Do not make the store accessible to applications yet.
   - Create the new store:

     ```
     java -Xmx64m -Xms64m \
     -jar KVHOME/lib/kvstore.jar makebootconfig \
     -root KVROOT \
     -host NewHost -port 8000 \
     -harange 8010,8020 \
     -capacity 1
     ```

   - Create security directory:

     ```
     java -Xmx64m -Xms64m \
     -jar kv/lib/kvstore.jar securityconfig \
     config create
     -root KVROOT -kspwd password
     Created files
     KVROOT/security/security.xml
     KVROOT/security/store.keys
     KVROOT/security/store.trust
     KVROOT/security/client.trust
     KVROOT/security/client.security
     KVROOT/security/store.passwd (Generated in CE version)
     KVROOT/security/store.wallet/cwallet.sso (Generated in EE
     version)

     Created
     ```

   - Start the new store:

     ```
     java -Xmx64m -Xms64m \
     -jar KVHOME/lib/kvstore.jar start \
     -root KVROOT &
     ```

   - Configure the new store:

     ```
     java -Xmx64m -Xms64m \
     -jar KVHOME/lib/kvstore.jar runadmin \
     -port 8000 -host NewHost \
     -security KVROOT/security/client.security
     kv-> configure -name NewStore
     Store configured: NewStore
     ```

> **Note:**
>
> Loading security metadata requires the names of the source store and the target store to be the same, otherwise the security metadata cannot be used later.

2. Locate the snapshot directories for the source store. There should be one for the admin nodes plus one for each shard. For example in a 3x3 store there should be 4 snapshot directories used for the load. The load program must have direct file-based access to each snapshot directory loaded. In this case, the snapshot source directory is in `/var/kvroot/mystore/sn1/admin1/snapshots/110915-153514-Thursday`.
3. Load the store metadata using the `-load-admin` option. Host, port, and store refer to the target store. In this case the `-source` directory must point to the environment directory of the admin node from the snapshot.

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar load \
-source \
/var/kvroot/mystore/sn1/admin1/snapshots/110915-153514-Thursday \
-store NewStore -host NewHost -port 8000 \
-load-admin \
-security KVROOT/security/client.security
```

> **Note:**
>
> This command can be run more than once if something goes wrong, as long as the store is not accessible to applications.

4. Deploy the store. For more information, see Configuring a single region data store.
5. Once the topology is deployed, load the shard data for each shard. To do this, run the Load program in parallel, with each instance operating on data captured from different replication nodes. For example, suppose there is a snapshot of `OldStore` in `var/backups/snapshots/140827-144141-back`.

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar load \
-source var/backups/snapshots/140827-144141-back -store NewStore \
-host NewHost -port 8000 \
-security KVROOT/security/client.security
```

> **Note:**
>
> This step may take a long time or might need to be restarted. In order to significantly reduce retry time, the use of a status file is recommended.
>
> If the previous store has been configured with username and password, the program will prompt for username and password here.

6. The store is now ready for applications.

Restoring Directly from a Snapshot

You can restore a store directly from a snapshot. This mechanism is faster than using the `Load` program. However, you can restore from a snapshot only to the *exact same* topology as was in use when the snapshot was taken. This means that all ports and host names or IP addresses (depending on your configuration) must be exactly the same as when you took the snapshot.

To restore from a snapshot, complete these steps:

1. Run this command on each of the Storage Nodes (SNs) to shut down the store:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar stop -root $KVROOT
```

2. When each SN is stopped, run this command on each SN in the store to restore to the backup (using `-update-config true`):

```
> java -jar KVHOME/lib/kvstore.jar start -root /var/kvroot \
-restore-from-snapshot 170417-104506-mySnapshot -update-config true
```

3. To restore to the backup, but not override the existing configurations, run this command on each SN (with `-update-config false`):

```
> java -jar KVHOME/lib/kvstore.jar start -root /var/kvroot \
-restore-from-snapshot 170417-104506-mySnapshot -update-config
false
```

The *170417–104506–mySnapshot* value represents the directory name of the snapshot to restore.

> **Note:**
>
> This procedure recovers the store to the time you created the snapshot. If your store was active after snapshot creation, all modifications made since the last snapshot are lost.

Recovering from Data Corruption

Oracle NoSQL Database can automatically detect data corruption in the database store. When it detects data corruption, Oracle NoSQL Database automatically shuts down the associated Admin or Replication Nodes. Manual administrative action is then required before the nodes can be brought back online.

Detecting Data Corruption

Oracle NoSQL Database Admin or Replication Node processes will exit when they detect data corruption. This is caused by a background task which detects data corruption caused by a disk failure, or similar physical media or I/O subsystem problem. Typically, the corruption is detected because of a checksum error in a log entry in one of the data (*.jdb) files contained in an Admin or Replication Node database environment. A data corruption error generates output in the debug log similar to this:

```
2016-10-25 16:59:52.265 UTC SEVERE [rg1-rn1] Process exiting
com.sleepycat.je.EnvironmentFailureException: (JE 7.3.2)
rg1-rn1(-1):kvroot/mystore/sn1/rg1-rn1/env
com.sleepycat.je.log.ChecksumException:
Invalid log entry type: 102 lsn=0x0/0x0 bufPosition=5
bufRemaining=4091 LOG_CHECKSUM:
Checksum invalid on read, log is likely invalid. Environment is
invalid and must be closed
...
2016-10-25 16:59:52.270 UTC SEVERE [rg1-rn1] Exception creating
service rg1-rn1:
(JE 7.3.2) rg1-rn1(-1):kvroot/mystore/sn1/rg1-rn1/env
com.sleepycat.je.log.ChecksumException:
Invalid log entry type: 102 lsn=0x0/0x0 bufPosition=5
bufRemaining=4091 LOG_CHECKSUM:
Checksum invalid on read, log is likely invalid. Environment is
invalid and must be closed. (12.1.4.3.0): oracle.kv.FaultException:
(JE 7.3.2) rg1-rn1(-1):kvroot/mystore/sn1/rg1-rn1/env
com.sleepycat.je.log.ChecksumException: Invalid log entry type: 102
lsn=0x0/0x0 bufPosition=5 bufRemaining=4091 LOG_CHECKSUM: Checksum
invalid on read, log is likely invalid. Environment is invalid and
must be closed. (12.1.4.3.0)
Fault class name: com.sleepycat.je.EnvironmentFailureException
...
2016-10-25 16:59:52.272 UTC INFO [rg1-rn1] Service status changed
from STARTING to ERROR_NO_RESTART
```

The `EnvironmentFailureException` will cause the process to exit. Because the exception was caused by log corruption, the service status is set to `ERROR_NO_RESTART`, which means that the service will not restart automatically.

Data Corruption Recovery Procedure

If an Admin or Replication Node has been stopped due to data corruption, then manual administration intervention is required in order to restart the Node:

1. Optional: Archive the corrupted environment data files.

   If you want to send the corrupted environment to Oracle support for help in identifying the root cause of the failure, archive the corrupted environment data files. These are usually located at:

   ```
   <KVROOT>/<STORE_NAME>/<SNx>/<Adminx>/"
   ```

or

```
<KVROOT>/<STORE_NAME>/<SNx>/<rgx-rnx>"
```

However, if you used the plan change-storagedir CLI command to change the storage directory for your Replication Node, then you will find the environment in the location that you specified to that command.

You can use the show topology CLI command to display your store's topology. As part of this information, the storage directory for each of your Replication Nodes are identified.

2. Confirm that a non-corrupted version of the data is available.

   Before removing the files associated with the corrupted environment, confirm that another copy of the data is available either on another node or via a previously save snapshot. For a Replication Node, you must be using a Replication Factor greater than 1 and also have a properly operating Replication Node in the store in order for the data to reside elsewhere in the store. If you are using a RF=1, then you must have a previously saved snapshot in order to continue.

   If the problem is with an Admin Node, there must be to be another Admin available in the store that is operating properly.

   Use the ping or verify configuration commands to check if the available nodes are running properly and healthy.

3. Remove all the data files that reside in the corrupted environment.

   Once the data files associated with a corrupted environment have been saved elsewhere, and you have confirmed that another copy of the data is available, delete all the data files in the enviroment directory. Make sure you only delete the files associated with the Admin or Replication Node that has failed due to a corrupted environment error.

   ```
   # ls <KVROOT>/mystore/sn1/rg1-rn1/env
   00000000.jdb  00000001.jdb  00000002.jdb  je.config.csv
   je.info.0 je.lck  je.stat.csv

   # rm <KVROOT>/mystore/sn1/rg1-rn1/env/*.jdb
   ```

4. Perform recovery using either Network Restore, or from a backup. Be aware the recovery from a backup will not work to recover an Admin Node.

   - Recovery using Network Restore

     Network restore can be used to recover from data corruption if the corrupted node belongs to a replication group that has other replication nodes available. Network restore is automatic recovery task. After removing all of the database files in the corrupted environment, you only need to connect to CLI and restart the corrupted node.

     For a Replication Node:

     ```
     kv-> plan start-service -service rg1-rn1
     ```

For an Admin:

```
kv-> plan start-service -service rg1-rn1
```

- Recovery from a backup (RNs only)

  If the store does not have another member in the Replication Node's shard or if all of the nodes in the shard have failed due to data corruption, you will need to restore the node's environment from a previously created snapshot. See Recovering the Store for details.

  Note that to recover an Admin that has failed due to data corruption, you must have a working Admin somewhere in the store. Snapshots do not capture Admin data.

Replacing a Failed Disk

You can replace a disk that is either in the process of failing, or has already failed. Disk replacement procedures are necessary to keep the store running. These are the steps required to replace a failed disk to preserve data availability.

The following example deploys a KVStore to a set of three machines, each with 3 disks. Use the `storagedir` flag of the `makebootconfig` command to specify the storage location of the disks.

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar makebootconfig \
    -root /opt/ondb/var/kvroot \
    -port 5000  \
    -host node09
    -harange 5010,5020 \
    -num_cpus 0  \
    -memory_mb 0 \
    -capacity 3  \
    -admindir /disk1/ondb/admin -admindirsize 1_gb \
    -storagedir /disk1/ondb/data \
    -storagedir /disk2/ondb/data \
    -storagedir /disk3/ondb/data \
    -rnlogdir /disk1/ondb/rnlog01
```

With a boot configuration such as the previous example, the directory structure created and populated on each machine is as follows:

```
 - Machine 1 (SN1) -     - Machine 2 (SN2) -    - Machine 3 (SN3) -
/opt/ondb/var/kvroot   /opt/ondb/var/kvroot  /opt/ondb/var/kvroot
  /security            /security              /security
  /store-name           /store-name           /store-name
    /sn1                   /sn2                   /sn3
      config.xml             config.xml             config.xml

/disk1/ondb/admin         /disk1/ondb/admin        /disk1/ondb/admin
  /admin1                   /admin2                  /admin3
    /env                      /env                     /env

/disk1/ondb/data         /disk1/ondb/data        /disk1/ondb/data
  /rg1-rn1                  /rg1-rn2                /rg1-rn3
```

```
    /env                        /env                        /env

/disk2/ondb/data          /disk2/ondb/data          /disk2/ondb/data
  /rg2-rn1                  /rg2-rn2                  /rg2-rn3
    /env                        /env                        /env

/disk3/ondb/data          /disk3/ondb/data          /disk3/ondb/data
  /rg3-rn1                  /rg3-rn2                  /rg3-rn3
    /env                        /env                        /env

/disk1/ondb/rnlog01       /disk1/ondb/rnlog01         /disk1/ondb/rnlog01
  /log                      /log                        /log
```

In this case, configuration information and administrative data is stored in a location that is separate from all of the replication data. The replication data itself is stored by each distinct Replication Node service on separate, physical media as well. Storing data in this way provides failure isolation and will typically make disk replacement less complicated and time consuming. For information on how to deploy a store, see Configuring a single region data store.

To replace a failed disk:

1. Determine which disk has failed. To do this, you can use standard system monitoring and management mechanisms. In the previous example, suppose disk2 on Storage Node 3 fails and needs to be replaced.
2. Then given a directory structure, determine which Replication Node service to stop. With the structure described above, the store writes replicated data to disk2 on Storage Node 3, so `rg2-rn3` must be stopped before replacing the failed disk.
3. Use the `plan stop-service` command to stop the affected service (rg2-rn3) so that any attempts by the system to communicate with it are no longer made; resulting in a reduction in the amount of error output related to a failure you are already aware of.

   ```
   kv-> plan stop-service -service rg2-rn3
   ```

4. Remove the failed disk (disk2) using whatever procedure is dictated by the operating system, disk manufacturer, and/or hardware platform.
5. Install a new disk using any appropriate procedures.
6. Format the disk to have the same storage directory as before; in this case, `/disk2/ondb/var/kvroot`.
7. With the new disk in place, use the `plan start-service` command to start the `rg2-rn3` service.

   ```
   kv-> plan start-service -service rg2-rn3
   ```

> **Note:**
>
> Depending on the amount of data stored on the disk before it failed, recovering that data can take a considerable amount of time. Also, the system may encounter unexpected or additional network traffic and load while repopulating the new disk. If so, such events add even more time to completion.

Replacing a Failed Storage Node

You can replace a failed Storage Node, or one that is in the process of failing. Upgrading a healthy machine to another one with better specifications is also a common Storage Node replacement scenario. Generally, you should repair the underlying problem (be it hardware or software related) before proceeding with this procedure.

There are two ways to replace a failed Storage Node:

- A new, different Storage node
- An identical Storage Node

This section describes both replacement possibilities.

> **Note:**
>
> Replacing a Storage Node qualifies as a topology change. This means that if you want to restore your store from a snapshot taken before the Storage Node was replaced, you must use the `Load` program. See Using the Load Program for more information.

Using a New Storage Node

To replace a failed Storage Node by using a new, different Storage Node (node uses different host name, IP address, and port as the failed host):

1. If you are replacing hardware, bring it up and make sure it is ready for your production environment.
2. On the new, replacement node, create a "boot config" configuration file using the `makebootconfig` utility with the following commands. Enable the security configuration option in the new node. Do this on the hardware where your new Storage Node runs.

```
> mkdir -p KVROOT     (if it doesn't already exist)
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar makebootconfig -root KVROOT \
                                           -port 5000 \
                                           -host <hostname> \
                                           -harange 5010,5020 \
                                           -capacity 1 \
                                           -admindir /export/admin1 \
                                           -admindirsize 3_gb \
                                           -store-security enable \
```

```
                                                -storagedir /export/
data1 \
                                                -storagedirsize 1_tb \
                                                -rnlogdir /export/rnlog1
```

3. Create the security directory under KVROOT in your new node.

```
> cd KVROOT
> mkdir security
```

4. Copy the security directory from a healthy node to the failed node:

```
scp -r <sec dir> node02:KVROOT/security
```

5. Start the Oracle NoSQL Database software on the new node:

```
> nohup java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar start -root KVROOT &
```

6. Deploy the new Storage Node to the new node. To do this using the CLI:

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar runadmin \
-port <5000> -host <host> \
 -security security/client.security
kv-> plan deploy-sn -zn <id> -host <host> -port <5000> -wait
```

7. Add the new Storage Node to the Storage Node pool. (You created a Storage Node pool when you installed the store, and you added all your Storage Nodes to it, but it is otherwise not used in this version of the product.)

```
kv-> show pools
AllStorageNodes: sn1, sn2, sn3, sn4 ... sn25, sn26
BostonPool: sn1, sn2, sn3, sn4 ... sn25
kv-> pool join -name BostonPool -sn sn26
AllStorageNodes: sn1, sn2, sn3, sn4 ... sn25, sn26
BostonPool: sn1, sn2, sn3, sn4 ... sn25
```

8. Make sure the old Storage Node is not running. If the problem is with the hardware, then turn off the broken machine. You can also stop just the Storage Node software by:

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar stop -root KVROOT &
```

9. Migrate the services from one Storage Node to another. The syntax for this plan is:

```
kv-> plan migrate-sn -from <old SN ID> -to <new SN ID> -wait
```

Assuming that you are migrating from Storage Node 25 to 26, you would use:

```
kv-> plan migrate-sn -from sn25 -to sn26 -wait
```

10. The old Storage Node is shown in the topology and is reported as UNREACHABLE. The source SNA should be removed and its rootdir should be hosed out. Bringing up the old SNA will also bring up the old Replication Nodes and admins, which are no longer members of their replication groups. This should be harmless to the rest of the store, but it produces log error messages that might be misinterpreted as indicating a problem with the store. Use the `plan remove-sn` command to remove the old and unused Storage Node in your deployment.

    ```
    kv-> plan remove-sn sn sn25 -wait
    ```

11. Use the ping command to verify the migration to the new node is complete and all services are running well.

    ```
    > java -Xmx64m -Xms64m \
            -jar KVHOME/lib/kvstore.jar ping \
            -port <5000> -host <host> \
            -security security/client.security
    ```

> **Note:**
>
> Replacing a Storage Node qualifies as a topology change. This means that if you want to restore your store from a snapshot taken before the Storage Node was replaced, you must use the `Load` program. See Using the Load Program for more information.

Task for an Identical Node

To replace a failed Storage Node with an identical node, i.e. the target node uses the same host name, internet address, and port as the failed host.

1. Prerequisite information:
   a. The hostname and port number (registry port) of the machine in the cluster where the admin process is running (e.g “host1” and 5000).
   b. The ID of the Storage Node to replace (e.g. "sn1").

   > **Note:**
   >
   > The user can use the Admin CLI ping command to get the registry port and Storage Node Identifier of any failed Storage Node.

   c. Before starting the new Storage Node, the Storage Node to be replaced must be taken down. This can be done administratively or via failure.

> **Note:**
>
> The instructions below assume that the KVROOT in the target host is empty and has no valid data. When the new Storage Node Agent begins it starts the services that it hosts, which recovers their data from other hosts. The time taken for the recovery depends on the size of the shards involved and it happens in the background.

2. Create the configuration file of the failed host using the `generateconfig` command. The `generateconfig` command can be executed from any active host (machine) in the NoSQL cluster.

   The generateconfig's usage is:

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar generateconfig \
-host <hostname> -port <port> -sn <StorageNodeId> -target <zipfile>
\
-security  <path to security login file>
```

Parameter	Required	Description
`host`	Yes	The host name of the failed storage node for which the config file is generated.
`port`	Yes	The registry port of the failed storage node for which the config file is generated.
`sn`	Yes	Identifier of the failed storage node.
`target`	Yes	Full path of the zip file to be created.
`security`	No	The client security configuration file. This parameter is only required if your store is secure. A fully qualified path to a file containing security information can be specified.

   For more information on generateconfig command, See generateconfig

   For example:

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar generateconfig -host adminhost \
-port 13230 -sn sn1 -target /tmp/sn1.config.zip \
-security USER/security/admin.security
```

   The command above creates the target "`/tmp/sn1.config.zip`" . This is a zip file with the required configuration to re-create the failed Storage Node. The top-level directory in the newly created zip file (`sn1.config.zip`) is the store's KVROOT.

> **Note:**
>
> This assumes that you must have followed the steps as mentioned in Create users and configure security with remote access .

3. Restore the Storage Node configuration on the target host:
   a. Copy the zip file "`sn1.config.zip`" to the target host.
   b. Unzip the archive into your target host's `KVROOT` directory. That is, if `KVROOT` is `/opt/kvroot`, then do the following:

```
> cd /opt
> unzip <path-to-sn1.config.zip>
```

> **Note:**
>
> If `kvroot` already exists under `/opt` directory , remove all the contents in the `kvroot` directory before unzipping the config file.

4. Restart the Storage Node on the target host.

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar start -root KVROOT
```

> **Note:**
>
> The hostname, port number and internet address of the target host and the failed node are the same. So no changes have to be done in the Storage Node pool and the topology of the store.

Repairing a Failed Zone by Replacing Hardware

If all of the machines belonging to a `zone` fail, and quorum is maintained, you can replace them by using new, different Storage Nodes deployed to the same zone.

If a zone fails but quorum is lost, you can perform a failover instead. To do this, see Performing a Failover.

For example, suppose a store consists of three `zones`; `zn1`, deployed to the machines on the first floor of a physical data center, `zn2`, deployed to the machines on the second floor, and `zn3`, deployed to the third floor. Additionally, suppose that a fire destroyed all of the machines on the second floor, resulting in the failure of all of the associated Storage Nodes. In this case, you need to replace the machines in the `zn2` zone; which can be accomplished by doing the following:

1. Replace each individual Storage Node in the failed zone with new, different Storage Nodes belonging to same zone (zn2), although located in a new physical location. To do this, follow the instructions in Replacing a Failed Storage Node. Make sure to remove each old Storage Node after performing the replacement.

2. After replacing and then removing each of the targeted SNs, the zone to which those SNs belonged should now contain the new SNs.

Managing your kvstore

Topics:

- Increasing Storage Node Capacity
- Managing Storage Directory Sizes
- Managing Admin Directory Size
- Disabling Storage Node Agent Hosted Services
- Verifying the Store
- Erasing Data
- Setting Store Parameters
- Removing an Oracle NoSQL Database Deployment
- Modifying Storage Node HA Port Ranges
- Modifying Storage Node Service Port Ranges

Increasing Storage Node Capacity

You can increase the capacity of a Storage Node by adding additional hard disks. Adding hard disks to a Storage Node permits the placement of each Replication Node on its own disk, ensuring that the Replication Nodes on the SN are not competing for I/O resources. Specify the location of the storage directory on the new disk using the `storagedir` parameter.

> **Note:**
>
> When you specify a storage directory, Oracle strongly recommends you also specify the storage directory size using the `-storagedirsize` parameter. See Managing Storage Directory Sizes for details. The system uses the configured directory sizes to enforce disk usage. Be sure to specify a storage directory size for every storage node in the store.

The following example demonstrates deploying a new store and adding two more disks to a Storage Node, increasing the capacity from 1 to 3:

1. Create, start and configure the new store.
   - Create the new store:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar makebootconfig \
-root KVROOT \
-host node20 -port 5000 \
-harange 5010,5030 \
-capacity 1 \
```

```
-memory_mb 200 \
-storagedir /disk1/ondb/data
```

- Create and copy the security directory:

```
java -Xmx64m -Xms64m \
-jar kv/lib/kvstore.jar \
securityconfig config create -root KVROOT -kspwd password
Created files
KVROOT/security/security.xml
KVROOT/security/store.keys
KVROOT/security/store.trust
KVROOT/security/client.trust
KVROOT/security/client.security
KVROOT/security/store.passwd (Generated in CE version)
KVROOT/security/store.wallet/cwallet.sso (Generated in EE version)

Created
```

- Start the new store:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar start \
-root KVROOT &
```

- Configure the new store:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar runadmin \
-port 5000 -host node20 \
-security KVROOT/security/client.security

kv-> configure -name kvstore
Store configured: kvstore
```

2. Create a zone. Then create an administration process on a specific host:

```
kv-> plan deploy-zone -name Houston -rf 1 -wait
Executed plan 1, waiting for completion...
Plan 1 ended successfully
kv-> plan deploy-sn -znname "Houston" -port 5000 -wait -host node20
Executed plan 2, waiting for completion...
Plan 2 ended successfully


kv-> plan deploy-admin -sn sn1 -port 5001 -wait
Executed plan 3, waiting for completion...
Plan 3 ended successfully
```

3. Create the storage node pool. Then add the storage node to the pool:

```
kv-> pool create -name AllStorageNodes


kv-> pool join -name AllStorageNodes -sn sn1
```

4. Create a topology, preview it, and then deploy it:

```
kv-> topology create -name 1x1 -pool AllStorageNodes -partitions 120
Created: 1x1


kv-> topology preview -name 1x1
Topology transformation from current deployed topology to 1x1:
Create 1 shard
Create 1 RN
Create 120 partitions

shard rg1
  1 new RN : rg1-rn1
  120 new partitions


kv-> plan deploy-topology -name 1x1 -wait
Executed plan 4, waiting for completion...
Plan 4 ended successfully
```

5. Add two more disk drives to the Storage Node, mounted as `disk2` and `disk3`. Add the storage directories using the `plan change-storagedir` command. Be sure to add the Storage Directory size, such as `-storagedirsize "1 tb"`.

```
kv-> plan change-storagedir -sn sn1 -storagedir /disk2/ondb/data \
-storagedirsize "1 tb" -add -wait
Executed plan 5, waiting for completion...
Plan 5 ended successfully
kv-> plan change-storagedir -sn sn1 -storagedir /disk3/ondb/data \
-storagedirsize "1 tb" -add -wait
Executed plan 6, waiting for completion...
Plan 6 ended successfully
```

> **Note:**
>
> Because we specified storage directory sizes in the previous example, it is necessary to provide that information to your other nodes **if** you have not already done so. See Managing Storage Directory Sizes for more information.

6. Change the capacity equal to the total number of disks now available on the Storage Node (3).

```
kv-> plan change-parameters -service sn1 -wait -params capacity=3
Executed plan 7, waiting for completion...
Plan 7 ended successfully
```

> **Note:**
>
> You need to perform last two steps on all the Storage Nodes (in your cluster) to add the disk drives and increase the capacity of each Storage Node. In this case, it is a single node deployment, so the topology is now ready to be redistributed.

7. Redistribute your topology to expand the cluster in order to use the new capacity (3) of the Storage Node.

```
kv-> topology clone -current -name 3x1
Created 3x1


kv-> topology redistribute -name 3x1 -pool AllStorageNodes
Redistributed: 3x1


kv-> topology preview -name 3x1
Topology transformation from current deployed topology to 3x1:
Create 2 shards
Create 2 RNs
Migrate 80 partitions

shard rg2
  1 new RN : rg2-rn1
  40 partition migrations
shard rg3
  1 new RN : rg3-rn1
  40 partition migrations


kv-> plan deploy-topology -name 3x1 -wait
Executed plan 8, waiting for completion...
Plan 8 ended successfully
```

Managing Storage Directory Sizes

We strongly recommend that you always specify storage directory sizes for each Replication Node on every Storage Node in the store. Doing so sets disk threshold levels for each replication node, even when your store has hardware with varying disk capacities. This section describes this topic, and others.

Managing Disk Thresholds

It is very important to configure each storage directory with a specific amount of available disk space. The Oracle NoSQL Database uses the configured Storage Directory sizes to enforce disk space limits. Without configuring how much disk space is available, the store opportunistically uses all available space, less 5 GB free disk space. The system maintains 5 GB of free space to allow manual recovery if the Storage Node exceeds its configured disk limit. Be sure to monitor disk usage regularly using the statistics provided, as described in Monitoring Disk Usage.

Storage Nodes use their available disk space for two purposes:

- To store your data.
- To save reserved files.

Reserved files consist of data that has already been replicated to active replica nodes. The purpose of storing a copy of this data is to use for Replica Nodes that lose contact with the Master Node. Losing contact typically occurs because Replica nodes are shut down, or a network partition event occurs, or because another transient problem occurs. The Storage Node is primarily designed to consume the amount of disk space you assign it, and to use the remaining disk space to save the reserved files. Each Storage Node manages its available disk space, leaving 5 GB free for recovery purposes. Your intervention is typically not required in this disk management process, unless a storage node exceeds its available disk capacity.

> **Note:**
>
> If a Storage Node (SN) consumes more than what is assigned as *storagedirsize*, including leaving 5 GB of space free, the SN automatically attempts to free up disk space by deleting reserved files (**not** your data files), until more than 5 GB of space is available. If the Storage Node is unable to free up enough space, it suspends write operations to the node. Read operations continue as normal. Write operations resume automatically once the node obtains sufficient free disk space.

You can limit how much disk space the store consumes on a node by node basis, by explicitly specifying a storage directory size for each storage node, as described in Specifying Storage Directory Sizes. Storage nodes can then consume all of their configured disk space as needed, leaving free the required 5 GB. However, if you do **not** indicate a storage directory size, the Storage Node uses disk space until it consumes the disk, except for the required 5 GB for manual recovery.

Consider a storage node with a 200 GB disk. Without configuring a *storagedirsize* for that disk, the store keeps consuming up to 195 GB of disk space (leaving only the 5 GB for manual recovery). If your standard policy requires a minimum 20 GB available space on each disk, you must configure the storage node with a *storagedirsize* of 175 GB, leaving 20 GB available, and 5 GB for store recovery.

The most common reason a node's storage directory fills up is because of reserved files. If the Storage Node exceeds its disk threshold, it continues to delete the reserved files until the threshold is no longer exceeded.

Specifying Storage Directory Sizes

Use the makebootconfig `storagedirsize` parameter to specify Storage Node (SN) capacity when you initially install your store. See Configuring your data store installation and makebootconfig for details. Additionally, if your SN has the capacity to support more than one Replication Node, specify a storage directory location and storage directory size for each Replication Node.

To specify or change storage capacity after you have installed the store, use `plan change-storagedir`. When you use `plan change-storagedir` be sure to specify the `-storagedirsize` parameter to indicate how large the new storage directory is.

> **Note:**
>
> If you specify the `-storagedir` parameter, but not `-storagedirsize`, `makebootconfig` displays a warning. Always specify both parameters for control and tracking.

The value specified for the `storagedirsize` parameter must be a long, optionally followed by a unit string. Accepted unit strings are: KB, MB, GB, and TB, corresponding to 1024, 1024^2, 1024^3, 1024^4 respectively. Acceptable strings are case insensitive. Valid delimiters between the long value and the unit string are " ", "-", or "_". If you specify the delimiter as " ", your value should be enclosed in double quotes, For example "10 GB". If you have any other delimiter double quotes is not mandatory. For example 10_GB or 10-GB.

For example:

```
kv-> verbose
Verbose mode is now on
kv-> show topology
store=mystore  numPartitions=300 sequence=308
  zn: id=zn1 name=Manhattan repFactor=3 type=PRIMARY
allowArbiters=false

  sn=[sn1] zn:[id=zn1 name=Manhattan] node1:9000 capacity=1 RUNNING
    [rg1-rn1] RUNNING  /storage-dir/sn1 0
              No performance info available
  sn=[sn2] zn:[id=zn1 name=Manhattan] node2:9000 capacity=1 RUNNING
    [rg1-rn2] RUNNING  /storage-dir/sn2 0
            single-op avg latency=0.0 ms   multi-op avg latency=0.0 ms
  sn=[sn3] zn:[id=zn1 name=Manhattan] node3:9000 capacity=1 RUNNING
    [rg1-rn3] RUNNING  /storage-dir/sn3 0
              No performance info available

  shard=[rg1] num partitions=300
    [rg1-rn1] sn=sn1 haPort=node1:9010
    [rg1-rn2] sn=sn2 haPort=node2:9010
    [rg1-rn3] sn=sn3 haPort=node3:9010
    partitions=1-300

kv-> plan change-storagedir -sn sn1 -storagedir /storage-dir/sn1 \
-storagedirsize "200 gb" -add -wait
```

```
Executed plan 7, waiting for completion...
Plan 7 ended successfully
kv-> plan change-storagedir -sn sn2 -storagedir /storage-dir/sn2 \
-storagedirsize "300 gb" -add -wait
Executed plan 8, waiting for completion...
Plan 8 ended successfully
kv-> plan change-storagedir -sn sn3 -storagedir /storage-dir/sn3 \
-storagedirsize "400 gb" -add -wait
Executed plan 9, waiting for completion...
Plan 9 ended successfully
kv-> show topology
store=mystore  numPartitions=300 sequence=308
  zn: id=zn1 name=Manhattan repFactor=3 type=PRIMARY
allowArbiters=false

  sn=[sn1] zn:[id=zn1 name=Manhattan] node1:9000 capacity=1 RUNNING
    [rg1-rn1] RUNNING  /storage-dir/sn1 214748364800
              No performance info available
  sn=[sn2] zn:[id=zn1 name=Manhattan] node2:9000 capacity=1 RUNNING
    [rg1-rn2] RUNNING  /storage-dir/sn2 322122547200
            single-op avg latency=0.0 ms   multi-op avg latency=0.0 ms
  sn=[sn3] zn:[id=zn1 name=Manhattan] node3:9000 capacity=1 RUNNING
    [rg1-rn3] RUNNING  /storage-dir/sn3 429496729600
            single-op avg latency=0.0 ms   multi-op avg latency=0.0 ms

  shard=[rg1] num partitions=300
    [rg1-rn1] sn=sn1 haPort=node1:9010
    [rg1-rn2] sn=sn2 haPort=node2:9010
    [rg1-rn3] sn=sn3 haPort=node3:9010
   partitions=1-300
```

Note:

If any Storage Node stores its data in the root directory (not recommended), then instead of `plan change-storagedir`, set the `rootDirSize` parameter. For example:

```
kv-> plan change-parameters -service sn1 -params
rootDirSize=200_gb
```

Specifying Differing Disk Capacities

By default, Oracle NoSQL Database evenly distributes data across all the Storage Nodes in your store. No check is made in advance. The store expects all of the hardware in your storee to be homogenous, and so all Storage Nodes would have the same disk capacity.

However, more likely, you are running a store in an environment where some Storage Nodes have more disk capacity than others. In this case, you must specify appropriate disk capacity for each storage node. Oracle NoSQL Database will then place more data on higher capacity Storage Nodes. Be aware that specifying greater disk capacity

to a storage node can result in an increased workload. Storage Nodes with more capacity than others could then serve more read and/or write activity. Be sure to size your storage nodes accordingly to support additional workload, if any.

Monitoring Disk Usage

If a Storage Node exceeds its disk usage threshold value (*storagedirsize* - 5GB), then all write activity for that node is suspended until sufficient disk space is made available. The store makes disk space available by removing reserved files to satisfy the threshold requirement. No data files are removed. Read activity continues while reserved data is being removed.

To ensure that your Storage Node can continue to service write requests, monitor the `availableLogSize` JMX statistic. This represents the amount of space that can be used by write operations. This value is **not** necessarily representative of the amount of disk space currently in use, since quite a lot of disk space can, and is, used for reserved files, which are not included in the `availableLogSize` statistic.

Reserved files are data files that have already been replicated, but which are retained for replication to nodes that are out of contact with the master node. Because Oracle NoSQL Database liberally reserves files, all available storage will frequently be consumed by reserved data. However, reserved data is automatically deleted as necessary by the Storage Node to continue write operations. For this reason, monitoring the actual disk usage is not meaningful.

If `availableLogSize` reaches zero, writes are suspended for the Storage Node. Earlier, as `availableLogSize` approaches zero, the node has less and less space for reserved data files. The result is that the store becomes increasingly less resilient in the face of a prolonged but temporary node outage because there are increasingly fewer historical log files that the store can use to gracefully bring a node up to date once it is available again.

The following tables lists some other useful statistics about disk usage. These statistics are stored in the stats file, or you can monitor them using the JMX `oracle.kv.repnode.envmetric` type. (Xref)

Statistic	Description
`availableLogSize`	Disk space available (in bytes) for write operations. This value is calculated with consideration fo reserved data files, which are deleted automatically whenever space is required to perform write operations: `free space + reservedLogSize - protectedLogSize` In general, monitoring disk usage in the file system is not meaningful, because of the presence of reserved files that can be deleted automatically.
`activeLogSize`	Bytes used by all active data files: files required for basic operation.
`reservedLogSize`	Bytes used by all reserved data files: files that have been cleaned and can be deleted if they are not protected.
`protectedLogSize`	Bytes used by all protected data files: the subset of reserved files that are temporarily protected and cannot be deleted.
`ProtectedLogSizeMap`	A breakdown of protectedLogSize as a map of protecting entity name to protected size in bytes.

Statistic	Description
`TotalLogSize`	Total bytes used by data files on disk: activeLogSize + reservedLogSize.

The following list from part of some JMX output, shows an example of how you will see each statistic. All of these statistic names have a `Cleaning_` prefix, indicating that they may be in the log cleaning statistics group (for garbage collection):

```
.
.
.
"Cleaning_nRepeatIteratorReads": 0,
"Cleaning_nLNsExpired": 0,
"Cleaning_nCleanerRuns": 0,
"Cleaning_nBINDeltasDead": 0,
"Cleaning_nCleanerDisksReads": 0,
"Cleaning_protectedLogSizeMap": "",
"Cleaning_nCleanerDeletions": 0,
"Cleaning_nCleanerEntriesRead": 0,
"Cleaning_availableLogSize": 48942137344,
"Cleaning_nLNsDead": 0,
"Cleaning_nINsObsolete": 0,
"Cleaning_activeLogSize": 112716,
"Cleaning_nINsDead": 0,
"Cleaning_nINsMigrated": 0,
"Cleaning_totalLogSize": 112716,
"Cleaning_nBINDeltasCleaned": 0,
"Cleaning_nLNsObsolete": 0,
"Cleaning_nLNsCleaned": 0,
"Cleaning_nLNQueueHits": 0,
"Cleaning_reservedLogSize": 0,
"Cleaning_protectedLogSize": 0,
"Cleaning_nClusterLNsProcessed": 0,
"Node Compression_processedBins": 0,
.
.
.
```

You can tell if writes have been suspended for a Storage Node using the `ping` command from the CLI. In the following sample output, the Shard Status shows `read-only:1`. This indicates that one of the Storage Nodes is in read-only mode. The likeliest reason for that is that it has exceeded its disk threshold.

```
kv-> ping
Pinging components of store istore based upon topology sequence #11
3 partitions and 3 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
Shard Status: healthy:0 writable-degraded:0 read-only:1 offline:0
Admin Status: healthy
Zone [name=dc1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online:1 offline:2
```

```
Storage Node [sn1] on sn1.example.com:5000 Zone: [name=dc1
id=zn1 type=PRIMARY allowArbiters=false masterAffinity=false] Status: RUNNING
Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Admin [admin1] Status: RUNNING,MASTER
Rep Node [rg1-rn1] Status: RUNNING,MASTER (non-authoritative)
sequenceNumber:39,177,477 haPort:5011 available storage size:6 GB
Storage Node [sn2] on sn2.example.com:5000 Zone:
[name=dc1 id=zn1 type=PRIMARY allowArbiters=false masterAffinity=false]
Status: RUNNING
Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Rep Node [rg1-rn2] Status: RUNNING,UNKNOWN sequenceNumber:39,176,478
haPort:5010 available storage size:NOT AVAILABLE delayMillis:?
catchupTimeSecs:?
Storage Node [sn3] on sn3.example.com:5000 Zone: [name=dc1
id=zn1 type=PRIMARY allowArbiters=false masterAffinity=false] Status: RUNNING
Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Rep Node [rg1-rn3] Status: RUNNING,UNKNOWN sequenceNumber:39,166,804
haPort:5010 available storage size:NOT AVAILABLE delayMillis:?
catchupTimeSecs:?
```

For information on JMX monitoring the store, see Java Management Extensions (JMX) Notifications.

Handling Disk Limit Exception

If a Storage Node exceeds its disk usage threshold value (`<storagedirsize> - 5 GB`), then the store suspends all write activities on that node, until sufficient data is removed to satisfy the threshold requirement. In such a situation, there are two ways to bring the store back to read and write availability, without deleting user data.

- Increasing `storagedirsize` on one or more Replication Nodes if there is available disk space
- Expanding the store by adding a new shard

If there is enough space left on the disk or if the complete disk size is not set as the size of `storagedirsize`, you can bring back the write availability (without any additional need of the hardware) by simply increasing the storage directory size for one or more Replication Nodes.

If there is not enough space left on disk or if the complete disk size is set as the size of the storage directory, then you should follow the store expansion procedure, where you will need additional hardware to increase the number of shards by one.

Note:

If you are following the store expansion procedure, it is important to check the performance files to see if the cleaner is working well, by monitoring the `minUtilization` statistics. If the `minUtilization` statistics is less than 30%, it may mean that the cleaner is not keeping up. In this case it is not possible to perform store expansion.
Store expansion can only be performed if the `minUtilization` statistics percentage is not less than 30%.

For example:

```
2018-03-07 16:07:12.499 UTC INFO [rg1-rn1] JE: Clean file
0x2b:
predicted min util is below minUtilization, current util min:
39 max: 39,
predicted util min: 39 max: 39, chose file with util min: 30
max: 30 avg: 30

2018-03-07 16:07:04.029 UTC INFO [rg1-rn2] JE: Clean file
0x27:
predicted min util is below minUtilization, current util min:
39 max: 39,
predicted util min: 39 max: 39, chose file with util min: 30
max: 30 avg: 30

2018-03-07 16:05:44.960 UTC INFO [rg1-rn3] JE: Clean file
0x27:
predicted min util is below minUtilization, current util min:
39 max: 39,
predicted util min: 39 max: 39, chose file with util min: 30
max: 30 avg: 30
```

Increasing Storage Directory Size

To increase the storage directory size in one or more Replication Nodes, open the CLI and execute the following commands:

1. Disable write operations on the store or on the failed shard.

   ```
   plan enable-requests -request-type READONLY \
   {-shard <shardId[,shardId]*> | -store}
   ```

   Here, `-request-type READONLY` is the option which disables write operations on a shard. You can disable write operations on one or more shards by using the `-shard` option, or on the entire store by using the `–store` option.

> **Note:**
>
> Though Replication Nodes are already in non-write availability mode whenever they hit an out of disk limit exception, it is important to disable user write operations explicitly. Disabling the user write operations ensures that the Replication Nodes are brought back up in the correct manner.

2. Execute the `PING` command to analyze the state of one or more Replication Nodes.

   ```
   kv-> ping
   ```

   Usually, when Replication Nodes hit an out of disk limit exception, Replica Replication Nodes are in the `RUNNING, UNKNOWN` state, and Master Replication Nodes are in the `RUNNING, MASTER (non-authoritative)` state.

3. To display the current, deployed topology, execute the `show topology –verbose` command. Make note of the current storage directory size allocated to each Replication Node.

   ```
   show topology -verbose [-zn] [-rn] [-an] [-sn] [-store] [-status] [-json]
   ```

4. To ensure that other Replication nodes in the store do not hit a disk limit exception while you increase the `storagedirsize`, reduce the JE free disk space on all Replication Nodes to 2 GB or 3 GB. You can use the `-all-rns` option to reduce the JE free disk space on all Replication Nodes at once, or the `-service -rgx-rgy` option to reduce the free disk space on a specific Replication Node.

   ```
   kv-> plan change-parameters [-all-rns|-service -rgx-rgy] \
   -params "configProperties=je.freeDisk=XXX"
   ```

   After executing this command with either option, the system will stop the Replication Nodes, update parameters, and restart Replication Nodes with the JE free disk space parameter you specify.

5. To increase the storage directory size on one or more Replication Nodes.

   ```
   kv-> plan change-storagedir -wait -sn snX \
   -storagedir <storagedirpath> -add -storagedirsize X_GB
   ```

   Here `snX` is the Storage Node whose directory size you want to increase, and X is the new storage size in GB.

6. After the `plan change-parameters` command executes successfully, verify the new `storagedirsize` value is assigned to one or more Replication Nodes in the store.

   ```
   show topology -verbose [-zn] [-rn] [-an] [-sn] [-store] [-status] [-json]
   ```

7. Lastly, reset the JE free disk space back to 5 GB. Also, enable write operations back on the store or a specific shard.

   ```
   kv-> plan change-parameters [-all-rns|-service -rgx-rgy] \
   -params "configProperties=je.freeDisk=5368709120"
   ```

```
kv-> plan enable-requests -request-type ALL {-shard
<shardId[,shardId]*> | -store}
```

The `-request-type ALL` option re-enables write operations on the store or on a specific shard.

Example

Let us consider a store with 1x3 topology, hitting a disk limit exception. Perform the following steps to increase the storage directory size of all Replication Nodes in the store from 16 GB to 25 GB.

1. Stop the write operations on the store level:

```
kv-> plan enable-requests -request-type READONLY -store;
```

2. Ping the store to analyze the state of one or more Replication Nodes.

```
kv-> ping
Pinging components of store istore based upon topology sequence #11
3 partitions and 3 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
Shard Status: healthy:0 writable-degraded:0 read-only:1 offline:0
total:1
Admin Status: healthy
Zone [name=dc1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online:1 offline:2   Storage Node [sn1] on node21:port1
Zone: [name=dc1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id:
a72484b8b33c
        Admin [admin1]          Status: RUNNING,MASTER
        Rep Node [rg1-rn1]      Status: RUNNING,MASTER (non-
authoritative)
        sequenceNumber:27,447,667 haPort:5011 available storage
size:12 GB
Storage Node [sn2] on node22:port1
Zone: [name=dc1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id:
a72484b8b33c
        Rep Node [rg1-rn2]      Status: RUNNING,UNKNOWN
        sequenceNumber:27,447,667 haPort:5010 available storage
size:10 GB delayMillis:? catchupTimeSecs:?
Storage Node [sn3] on node23:port1
Zone: [name=dc1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id:
a72484b8b33c
        Rep Node [rg1-rn3]      Status: RUNNING,UNKNOWN
        sequenceNumber:27,447,667 haPort:5010 available storage
size:9 GB delayMillis:? catchupTimeSecs:?
```

The example shows that the Replication Nodes are in `RUNNING, UNKNOWN` state and Master Replication Node is in `RUNNING, MASTER(non-authoritative)` state.

3. View the current, deployed topology.

```
kv-> show topology -verbose
store=istore  numPartitions=3 sequence=11
  zn: id=zn1 name=dc1 repFactor=3 type=PRIMARY allowArbiters=false \
  masterAffinity=false

  sn=[sn1] zn:[id=zn1 name=dc1] node21:port1 capacity=1 RUNNING
    [rg1-rn1] RUNNING  /scratch/kvroot 16 GB
                  single-op avg latency=36.866146 ms   multi-op avg
latency=0.0 ms
    [rg1-rn1] RUNNING  /scratch/kvroot 16 GB
                  single-op avg latency=36.866146 ms   multi-op avg
latency=0.0 ms
  sn=[sn2] zn:[id=zn1 name=dc1] node22:port1 capacity=1 RUNNING
    [rg1-rn2] RUNNING  /scratch/kvroot 16 GB
                  single-op avg latency=0.0 ms   multi-op avg latency=0.0
ms
    [rg1-rn2] RUNNING  /scratch/kvroot 16 GB
                  single-op avg latency=0.0 ms   multi-op avg latency=0.0
ms
  sn=[sn3] zn:[id=zn1 name=dc1] node23:port1 capacity=1 RUNNING
    [rg1-rn3] RUNNING  /scratch/kvroot 16 GB
                  single-op avg latency=0.0 ms   multi-op avg latency=0.0
ms
    [rg1-rn3] RUNNING  /scratch/kvroot 16 GB
                  single-op avg latency=0.0 ms   multi-op avg latency=0.0
ms

  numShards=1
  shard=[rg1] num partitions=3
    [rg1-rn1] sn=sn1 haPort=node21:port2
    [rg1-rn2] sn=sn2 haPort=node22:port3
    [rg1-rn3] sn=sn3 haPort=node23:port3
    partitions=1-3
```

You see that 16 GB of disk space is assigned as the storage directory size for each Replication Node.

4. Reduce the JE free disk space from 5 GB to 2 GB for all Replication Nodes in the store.

```
kv-> plan change-parameters -all-rns -params \
"configProperties=je.freeDisk=2147483648";
Started plan 70. Use show plan -id 70 to check status.
To wait for completion, use plan wait -id 70
```

5. For each Replication Node, increase the storage directory size to 25 GB.

```
kv-> plan change-storagedir -wait -sn sn1 -storagedir /scratch/kvroot \
-add -storagedirsize 25_GB -wait
```

```
Executed plan 72, waiting for completion...
Plan 72 ended successfully


kv-> plan change-storagedir -wait -sn sn2 -storagedir /scratch/
kvroot \
-add -storagedirsize 25_GB -wait
Executed plan 73, waiting for completion...
Plan 73 ended successfully


kv-> plan change-storagedir -wait -sn sn3 -storagedir /scratch/
kvroot \
-add -storagedirsize 25_GB -wait
Executed plan 74, waiting for completion...
Plan 74 ended successfully
```

6. View the topology again to verify that the new value is assigned to `storagedirsize`.

```
kv-> show topology -verbose
store=istore  numPartitions=3 sequence=11
  zn: id=zn1 name=dc1 repFactor=3 type=PRIMARY allowArbiters=false \
masterAffinity=false

  sn=[sn1] zn:[id=zn1 name=dc1] node21:port1 capacity=1 RUNNING
    [rg1-rn1] RUNNING  /scratch/kvroot 25 GB
                  single-op avg latency=0.0 ms   multi-op avg
latency=0.0 ms
  sn=[sn2] zn:[id=zn1 name=dc1] node22:port1 capacity=1 RUNNING
    [rg1-rn2] RUNNING  /scratch/kvroot 25 GB
                  single-op avg latency=552.51996 ms   multi-op avg
latency=0.0 ms
  sn=[sn3] zn:[id=zn1 name=dc1] node23:port1 capacity=1 RUNNING
    [rg1-rn3] RUNNING  /scratch/kvroot 25 GB
                  single-op avg latency=14.317171 ms   multi-op avg
latency=0.0 ms

  numShards=1
  shard=[rg1] num partitions=3
    [rg1-rn1] sn=sn1 haPort=node21:port2
    [rg1-rn2] sn=sn2 haPort=node22:port3
    [rg1-rn3] sn=sn3 haPort=node23:port3
    partitions=1-3
```

The example now shows that 25 GB is assigned as the storage directory size for each Replication Node.

7. Reset the JE free disk space to 5 GB and enable write operations back on the store.

```
kv-> plan change-parameters [-all-rns|-service -rgx-rgy] \
-params "configProperties=je.freeDisk=5368709120"
```

```
kv-> plan enable-requests -request-type READONLY -store;
```

Adding a New Shard

Apart from increasing the storage directory size, you can also handle disk limit exceptions by adding a new shard and expanding your store.

The following example demonstrates adding three new Storage Nodes (Storage Nodes 21, 22, and 23) and deploying the new store to recover from disk limit exception:

1. Disable write operations on the store.

   ```
   kv-> plan enable-requests -request-type READONLY -store;
   ```

   Here, `-request-type READONLY` disables write operations on a store and allows only read operations.

2. Reduce the JE free disk space to 2 GB on all nodes and increase the `je.cleaner.minUtilization` configuration parameter from 40 (the default in a KVStore) to 60.

   ```
   kv-> plan change-parameters -all-rns \
   -params "configProperties=je.cleaner.minUtilization 60; \
   je.freeDisk 2147483648";
   ```

   Executing this command creates more free space for store expansion. Replication Nodes will be stopped, parameters will be updated, and the Replication Nodes will be restarted with the new parameters.

3. Create, start, and configure the new nodes for expanding the store.

   - Create the new node. Run the `makebookconfig` utility to configure each Storage Node in the store:

     ```
     java -Xmx256m -Xms256m -jar KVHOME/kvstore.jar makebootconfig \
     -root sn1/KVROOT \
     -store-security none -capacity 1 \
     -port port1 -host node21 \
     -harange 5010,5020 \
     -storagedir /scratch/sn1/u01 -storagedirsize 20-Gb
     ```

     ```
     java -Xmx256m -Xms256m -jar KVHOME/kvstore.jar makebootconfig \
     -root sn2/KVROOT \
     -store-security none -capacity 1 \
     -port port1 -host node22 \
     -harange 5010,5020 \
     -storagedir /scratch/sn2/u01 -storagedirsize 20-Gb
     ```

     ```
     java -Xmx256m -Xms256m -jar KVHOME/kvstore.jar makebootconfig \
     -root sn3/KVROOT \
     -store-security none -capacity 1 \
     -port port1 -host node23 \
     ```

```
-harange 5010,5020 \
-storagedir /scratch/sn3/u01 -storagedirsize 20-Gb
```

- Restart the Storage Node Agent (SNA) on each of the Oracle NoSQL Database nodes using the `start` utility:

```
kv-> nohup java -Xmx256m -Xms256m -jar \
KVHOME/lib/kvstore.jar start -root KVROOT &
```

- Configure the new store:

```
java -Xmx256m -Xms256m -jar KVHOME/lib/kvstore.jar runadmin \
-port port1 -host node21
```

```
java -Xmx256m -Xms256m -jar KVHOME/lib/kvstore.jar runadmin \
-port port1 -host node22
```

```
java -Xmx256m -Xms256m -jar KVHOME/lib/kvstore.jar runadmin \
-port port1 -host node23
```

4. Redistribute the store according to its new configuration.

```
kv-> java -Xmx256m -Xms256m -jar KVHOME/lib/kvstore.jar runadmin \
-port port1 -host host1
```

```
kv-> plan deploy-sn -zn zn1 -host node21 -port port1 -wait
Executed plan 7, waiting for completion...
Plan 7 ended successfully
kv-> plan deploy-sn -zn zn1 -host node22 -port port1 -wait
Executed plan 8, waiting for completion...
Plan 8 ended successfully
kv-> plan deploy-sn -zn zn1 -host node23 -port port1 -wait
Executed plan 9, waiting for completion...
Plan 9 ended successfully
Plan 11 ended successfully
```

```
kv-> pool join -name ExamplePool -sn sn4
Added Storage Node(s) [sn4] to pool ExamplePool
kv-> pool join -name ExamplePool -sn sn5
Added Storage Node(s) [sn5] to pool ExamplePool
```

```
kv-> pool join -name ExamplePool -sn sn6
Added Storage Node(s) [sn6] to pool ExamplePool


kv-> topology clone -current -name newTopo
Created newTopo


kv-> topology redistribute -name newTopo -pool ExamplePool
Redistributed: newTopo


kv-> plan deploy-topology -name newTopo -wait
Executed plan 11, waiting for completion...
```

5. Restore the Replication Nodes to its original configuration.

```
plan change-parameters -all-rns \
-params "configProperties=je.cleaner.minUtilization 40; \
je.freeDisk 5368709120";
```

6. Enable write operations back on the store.

```
kv-> plan enable-requests -request-type ALL -store;
```

Here, `–request-type ALL` enables both read and write operations on the store.

Managing Admin Directory Size

You should specify a sufficient directory size for the Admin database when you initially install your store, using the `makebootconfig admindirsize` parameter. If you do not specify a value, the system allocates a default of 3 GB as the size of the Admin directory. See Configuring your data store installation and makebootconfig for details.

Specify the value for the `-admindirsize` parameter as a long, optionally followed by a unit string. Accepted unit strings are: KB, MB, and GB, corresponding to 1024, 1024^2, and 1024^3 respectively. Acceptable strings are case insensitive. Valid delimiters between the long value and the unit string are " ", "-", or "_". If you specify the delimiter as " ", your value should be enclosed in double quotes, For example "10 GB". If you have any other delimiter double quotes is not mandatory. For example 10_GB or 10-GB.

Also, if the admin directory fills up its allotted storage space with reserved files, see Managing Disk Thresholds for more information.

If the Admin completely uses up its storage space, it will not be able to start. This condition is unlikely to occur, but in the event that your Admin cannot start, you should check its available disk space. If the directory is full, then you should increase the available disk space to the Admin. For the Admin to completely fill its storage space with actual data files, the store would have to be configured in some unexpected way — such as with an extraordinarily large number of tables, or have been allotted a very small Admin directory size.

The procedure that you use to change an Admin's allocated disk space differs depending on whether the Admin is in working condition.

Admin is Working

To increase or decrease the Admin's disk space when the Admin is functional, use the CLI to execute the following plan:

```
plan change-parameters -all-admins -params \
"configProperties=je.maxDisk=<size>"
```

where <size> is the desired storage size in bytes.

Admin is not Working

To increase or decrease the Admin's disk space when the Admin is *not* functional:

1. Set the value of `je.maxDisk` to the desired value in `config.xml` for all Admins manually:
   a. For each Storage Node that is hosting an Admin, locate the `config.xml` file in the Storage Node's root directory:

      ```
      <kvroot dir>/<store name>/<SN name>/config.xml
      ```

      and edit it as follows.
   b. Locate the admin section of the `config.xml` file. This is the section that begins with:

      ```
      <component name="ADMIN-NAME" type="adminParams" validate="true">
          ...
      </component>
      ```

   c. Add the following line to the admin section of each `config.xml` file:

      ```
      <propertyname="configProperties" value="je.maxDisk=<size>"
      type="STRING"/>
      ```

      where <size> is the desired storage size in bytes for your Admin.
2. Stop/start these Storage Nodes one by one, using the following commands:

   ```
   java -Xmx64m -Xms64m \
   -jar kvstore.jar stop -root <root dir> \
   -config <config file name>

   java -Xmx64m -Xms64m \
   -jar kvstore.jar start -root <root dir> \
   -config <config file name>
   ```

3. Wait for the status of these Storage Nodes to change to RUNNING. You can use the ping command to get the Storage Node status:

```
java -Xmx64m -Xms64m \
-jar kvstore.jar runadmin -host <host name> -port <port> ping
```

4. If any Admins are unreachable (you cannot get a response using the ping command), start them from the CLI using the following command:

```
kv-> plan start-service -service <ADMIN_NAME> -wait
```

5. Once all the Admins are running, execute the following command using the CLI:

```
plan change-parameters -all-admins -params \
"configProperties=je.maxDisk=<size>"
```

   where <size> is the desired storage size in bytes for your Admin. This value should match the value you provided in the `config.xml` file.

Disabling Storage Node Agent Hosted Services

To disable all services associated with a stopped SNA use the `-disable-services` flag. This helps isolate failed services to avoid hard rollbacks during a failover. Also, in this way, the configuration can be updated during recovery after a failover. The usage is:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar {start | stop | restart}
[-disable-services] [-verbose]
-root KVROOT [-config <bootstrapFileName>]
```

where:

- `start -disable-services`

  Starts an Oracle NoSQL Database Storage Node Agent with all of its hosted services disabled. If the SNA is already running, the command will fail.

- `stop -disable-services`

  Stops an Oracle NoSQL Database Storage Node Agent, marking all of its services disabled so that they will not start when starting up the SNA in the future or until the services are reenabled.

- `restart -disable-services`

  Restarts an Oracle NoSQL Database Storage Node Agent with all of its hosted services disabled.

Verifying the Store

Use the Admin CLI verify command to complete these tasks:

- Perform general troubleshooting of the store.

The `verify` command inspects all store components. It also checks whether all store services are available. For the available store services, the command also checks for any version or metadata mismatches.

- Check the status of a long-running plan

  Some plans require many steps and may take some time to execute. The administrator can verify plans to check on the plan progress. For example, you can verify a `plan deploy-sn` command while it is running against many Storage Nodes. The `verify` command can report at each iteration to confirm that additional nodes have been created and come online.

  For more about managing plans, see Using Plans.

- Get additional information to help diagnose a plan in an ERROR state.

You verify your store using the `verify` command in the CLI. The command requires no parameters, and runs in verbose mode, by default. For example:

```
kv-> verify configuration
Verify: starting verification of store MetroArea based upon
topology sequence #117
100 partitions and 6 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
See node01:Data/virtualroot/datacenter1/kvroot/MetroArea/
                                          log/MetroArea_{0..N}.log for
                                          progress messages
Verify: Shard Status: healthy:2 writable-degraded:0
                                            read-only:0 offline:0
Verify: Admin Status: healthy
Verify: Zone [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
   RN Status: online:2 offline: 0 maxDelayMillis:1 maxCatchupTimeSecs:0
Verify: Zone [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
   RN Status: online:2 offline: 0 maxDelayMillis:1 maxCatchupTimeSecs:0
Verify: Zone [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]
   RN Status: online:2 offline: 0
Verify: == checking storage node sn1 ==
Verify: Storage Node [sn1] on node01:5000
   Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin1]          Status: RUNNING,MASTER
Verify:         Rep Node [rg1-rn2]      Status: RUNNING,REPLICA
   sequenceNumber:127 haPort:5011 available storage size:14 GB
delayMillis:1 catchupTimeSecs:0
Verify: == checking storage node sn2 ==
Verify: Storage Node [sn2] on node02:6000
   Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn2]      Status: RUNNING,REPLICA
   sequenceNumber:127 haPort:6010 available storage size:24 GB
```

```
delayMillis:1 catchupTimeSecs:0
Verify: == checking storage node sn3 ==
Verify: Storage Node [sn3] on node03:7000
   Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin2]          Status: RUNNING,REPLICA
Verify:         Rep Node [rg1-rn3]      Status: RUNNING,REPLICA
   sequenceNumber:127 haPort:7011 available storage size:22 GB delayMillis:1
catchupTimeSecs:0
Verify: == checking storage node sn4 ==
Verify: Storage Node [sn4] on node04:8000
   Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn3]      Status: RUNNING,REPLICA
   sequenceNumber:127 haPort:8010 available storage size:24 GB delayMillis:1
catchupTimeSecs:0
Verify: == checking storage node sn5 ==
Verify: Storage Node [sn5] on node05:9000
   Zone: [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin3]          Status: RUNNING,REPLICA
Verify:         Rep Node [rg1-rn1]      Status: RUNNING,MASTER
   sequenceNumber:127 haPort:9011 available storage size:18 GB
Verify: == checking storage node sn6 ==
Verify: Storage Node [sn6] on node06:10000
   Zone: [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn1]      Status: RUNNING,MASTER
   sequenceNumber:127 haPort:10010 available storage size:16 GB

Verification complete, no violations.
```

Use the optional `-silent` mode to show only problems or completion.

```
kv-> verify configuration -silent
Verify: starting verification of store MetroArea based upon
topology sequence #117
100 partitions and 6 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
See node01:Data/virtualroot/datacenter1/kvroot/MetroArea/
                            log/MetroArea_{0..N}.log for progress messages
Verification complete, no violations.
```

The `verify` command clearly reports any problems with the store. For example, if a Storage Node is unavailable, using `-silent` mode displays that problem as follows:

```
kv-> verify configuration -silent
Verify: starting verification of store MetroArea based upon
topology sequence #117
100 partitions and 6 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
See node01:Data/virtualroot/datacenter1/kvroot/MetroArea/
                            log/MetroArea_{0..N}.log for progress
messages
Verification complete, 2 violations, 0 notes found.
Verification violation: [rg2-rn2]       ping() failed for rg2-rn2 :
Unable to connect to the storage node agent at host node02, port 6000,
which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host: node02;
        nested exception is:
        java.net.ConnectException: Connection refused
Verification violation: [sn2]   ping() failed for sn2 : Unable to
connect
        to the storage node agent at host node02, port 6000,
        which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host: node02;
        nested exception is:
        java.net.ConnectException: Connection refused
```

Using the default mode (verbose), `verify configuration` shows the same problem as follows:

```
kv-> verify configuration
Verify: starting verification of store MetroArea based upon
topology sequence #117
100 partitions and 6 storage nodes
Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
See node01:Data/virtualroot/datacenter1/kvroot/MetroArea/
                            log/MetroArea_{0..N}.log for progress
messages
Verify: Shard Status: healthy:1 writable-degraded:1
                                                 read-only:0 offline:0
Verify: Admin Status: healthy
Verify: Zone [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
   RN Status: online:1 offline: 1 maxDelayMillis:1 maxCatchupTimeSecs:0
Verify: Zone [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
   RN Status: online:2 offline: 0 maxDelayMillis:1 maxCatchupTimeSecs:0
Verify: Zone [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]
   RN Status: online:2 offline: 0
Verify: == checking storage node sn1 ==
Verify: Storage Node [sn1] on node01:5000
   Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
   Status: RUNNING
```

```
   Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin1]          Status: RUNNING,MASTER
Verify:         Rep Node [rg1-rn2]      Status: RUNNING,REPLICA
   sequenceNumber:127 haPort:5011 available storage size:18 GB delayMillis:1
catchupTimeSecs:0
Verify: == checking storage node sn2 ==
Verify:         sn2: ping() failed for sn2 :
Unable to connect to the storage node agent at host node02, port 6000,
which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host: node02;
        nested exception is:
        java.net.ConnectException: Connection refused
Verify: Storage Node [sn2] on node02:6000
   Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
   UNREACHABLE
Verify:         rg2-rn2: ping() failed for rg2-rn2 :
Unable to connect to the storage node agent at host node02, port 6000,
which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host: node02;
        nested exception is:
        java.net.ConnectException: Connection refused
Verify:         Rep Node [rg2-rn2]      Status: UNREACHABLE
Verify: == checking storage node sn3 ==
Verify: Storage Node [sn3] on node03:7000
  Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
  Status: RUNNING
  Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin2]          Status: RUNNING,REPLICA
Verify:         Rep Node [rg1-rn3]      Status: RUNNING,REPLICA
  sequenceNumber:127 haPort:7011 available storage size:12 GB delayMillis:1
catchupTimeSecs:0
Verify: == checking storage node sn4 ==
Verify: Storage Node [sn4] on node04:8000
  Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
  Status: RUNNING
  Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn3]      Status: RUNNING,REPLICA
  sequenceNumber:127 haPort:8010 available storage size:11 GB delayMillis:0
catchupTimeSecs:0
Verify: == checking storage node sn5 ==
Verify: Storage Node [sn5] on node05:9000
  Zone: [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]
  Status: RUNNING
  Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Admin [admin3]          Status: RUNNING,REPLICA
Verify:         Rep Node [rg1-rn1]      Status: RUNNING,MASTER
  sequenceNumber:127 haPort:9011 available storage size:14 GB
Verify: == checking storage node sn6 ==
Verify: Storage Node [sn6] on node06:10000
  Zone: [name=Queens id=zn3 type=PRIMARY allowArbiters=false
masterAffinity=false]
```

```
  Status: RUNNING
  Ver: 18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Verify:         Rep Node [rg2-rn1]      Status: RUNNING,MASTER
  sequenceNumber:127 haPort:10010 available storage size:16 GB

Verification complete, 2 violations, 0 notes found.
Verification violation: [rg2-rn2]       ping() failed for rg2-rn2 :
Unable to connect to the storage node agent at host node02, port 6000,
which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host: node02;
        nested exception is:
        java.net.ConnectException: Connection refused
Verification violation: [sn2]   ping() failed for sn2 :
Unable to connectto the storage node agent at host node02, port 6000,
which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host: node02;
        nested exception is:
        java.net.ConnectException: Connection refused
```

Note:

The verify output is only displayed in the shell after the command is complete. Use `tail`, or `grep` the Oracle NoSQL Database log file to get a sense of how the verification is progressing. Look for the string `Verify`. For example:

```
grep Verify /KVRT1/mystore/log/mystore_0.log
```

Erasing Data

The Oracle NoSQL Database has a built-in erasure feature that can be run in the background to erase user data after it has become obsolete (i.e. deleted data, expired data, older versions of updated records). The goal of this background thread is to erase the data with very little impact on application performance. The background thread takes a best effort approach. If a replication node is down, no erasure will occur until it comes back up.

Erasure happens over a user defined erasure time period (its cycle) and it is recommended that the cycle time be long enough for all the erasure work to complete. Any work not completed will be carried over to the next cycle. Data is erased from the tables and corresponding indexes. The Replication Node parameters: "enableErasure", and "erasurePeriod" control the behavior of erasure. For more information on setting these parameters, see Setting Store Wide Policy Parameters.

Setting Store Parameters

The three Oracle NoSQL Database service types, Admin, Storage Node, and Replication Node, have configuration parameters. You can modify some parameters

after deploying the service. Use the following Admin CLI command to see the parameter values that you can change:

```
show parameters -service <>
```

You identify an Admin, Storage Node, or Replication service using a valid string. The `show parameters -service` command displays service parameters and state for any of the three services. Use the optional `-policy` flag to show global policy parameters.

Changing Parameters

All of the CLI commands used for creating parameter-changing plans share a similar syntax:

```
plan change-parameters -service <id>...
```

All such commands can have multiple `ParameterName=NewValue` assignment arguments on the same command line. If NewValue contains spaces, then the entire assignment argument must be quoted within double quote marks. For example, to change the Admin parameter collectorPollPeriod, you would issue the command:

```
kv-> plan change-parameters -all-admins -params \
    "collectorPollPeriod=20 SECONDS">
```

If your configProperties for all Replication Nodes is set to:

```
"configProperties=je.cleaner.minUtilization=40;">
```

And you want to add new settings for configProperties, you would issue the following command:

```
kv-> plan change-parameters -all-rns -params \
     "configProperties=je.cleaner.minUtilization=40;\
     je.env.runVerifier=false;">
```

If for some reason, different Replication Nodes have different configProperties parameter values, then the change-parameters command will need to be tailored for each Replication Node.

The following commands are used to change service parameters:

- `plan change-parameters -service <shardId-nodeId> -params [assignments]`

  This command is used to change the parameters of a single Replication Node, which must be identified using the shard and node numbers. The `shardId-nodeId` identifier must be given as a single argument with one embedded hyphen and no spaces. The `shardId` identifier is represented by `rgX`, where `X` refers to the shard number.

- `plan change-parameters -all-rns -params [assignments]`

  This command is used to change the parameters of all Replication Nodes in a store. No Replication Node identifier is needed in this case.

- `plan change-parameters -service <storageNodeId> -params [assignments]`

This command is used to change the parameters of a single Storage Node instance. The storageNodeId is a simple integer.

- `plan change-parameters -all-admins -params [assignments]`

  This command is used to change Admin parameters. Because each instance of Admin is part of the same replicated service, all instances of the Admin are changed at the same time, so no Admin identifier is needed in this command.

  If an Admin parameter change requires the restarting of the Admin service, KVAdmin loses its connection to the server. Under normal circumstances, KVAdmin automatically reconnects after a brief pause, when the next command is given. At this point the plan is in the `INTERRUPTED` state, and must be completed manually by issuing the `plan execute` command.

- `plan change-parameters -security <id>`

  This command is used to change security parameters. The parameters are applied implicitly and uniformly across all SNs, RNs and Admins.

In all cases, you can choose to create a plan and execute it; or to create the plan and execute it in separate steps by using the `-noexecute` option of the plan command.

Setting Store Wide Policy Parameters

Most admin, Storage Node, and replication node parameters are assigned to default values when a store is deployed. It can be inconvenient to adjust them after deployment, so Oracle NoSQL Database provides a way to set the defaults that are used during deployment. These defaults are called store-wide Policy parameters.

You can set policy parameters in the CLI by using this command:

```
change-policy -params [name=value]
```

The parameters to change follow the `-params` flag and are separated by spaces. Parameter values with embedded spaces must be separated by spaces. Parameter values with embedded spaces must be quoted. For example: name = "value with spaces". If the optional `dry-run` flag is specified, the new parameters are returned without changing them.

Admin Parameters

You can set the following parameters for the Admin service:

- `collectorPollPeriod=<Long TimeUnit>`

  Sets the Monitor subsystem's delay for polling the various services for status updates. This value defaults to "20" seconds. Units are supplied as a string in the `change-parameters` command, for example: -params collectorPollPeriod="2 MINUTES"

- `loggingConfigProps=<String>`

  Property settings for the Logging subsystem in the Admin process. Its format is `property=value;property=value...`. Standard java.util.logging properties can be set by this parameter.

- `eventExpiryAge=<Long TimeUnit>`

You can use this parameter to adjust how long the Admin stores critical event history. The default value is "30 DAYS".

- `configProperties=<String>`

  This is an omnibus string of property settings for the underlying BDB JE subsystem. Its format is `property=value;property=value....`

- `javaMiscParams=<String>` [deprecated]

  This parameter should ONLY be used to set flags for which there are no service parameters.

  This parameter is **deprecated**. You are encouraged not to use it, and instead use the `javaAdminParamsOverride` parameter.

- `javaAdminParamsOverride=<String>`

  This is an omnibus string that is added to the command line when the Admin process is started. This parameter is intended for specifying miscellaneous JVM properties that cannot be specified using other Admin parameters. If the string is not a valid sequence of tokens for the JVM command line, the Admin process fails to start.

  No default value is provided for this parameter.

Changing Admin JVM Memory Parameters

Admin processes can run out of memory. One of the most likely reasons is that the default memory setting was insufficient for the Admin services to represent all of the metadata associated with the store. Metadata includes information about tables, security information about users and roles, and information about incomplete plans. Stores with large amounts of metadata may need to increase the memory setting for Admin services if the activity logs show that Admin services are failing with `OutOfMemoryError`. This topic describes increasing the memory setting of the `javaAdminParamsOverride`.

The system continues to use the old `javaMiscParams` setting to specify the initial JVM memory settings for the admin. The system does not use `javaAdminParamsOverride` since it is reserved for use if you want to override the default settings.

To change the `javaAdminParamsOverride` requires a comprehensive all or nothing change. You cannot change individual parameters within the set. To change any setting, declare them all in the `plan change-parameters` command, described next.

First determine the basic information about all Admin services using the `show admins` command. You get the output as shown below.

```
show admins
admin1: Storage Node sn1 storageDir=/home/opc/nosql/kvroot type=PRIMARY
(connected RUNNING,MASTER)
admin2: Storage Node sn2 storageDir=/home/opc/nosql/kvroot type=PRIMARY
(RUNNING,REPLICA)
admin3: Storage Node sn3 storageDir=/home/opc/nosql/kvroot type=PRIMARY
(RUNNING,REPLICA)
```

> **Note:**
>
> The above output is just an example. Here the replication factor RF=3 for the primary nodes. Your output will reflect your topology.

To determine the current settings of `javaAdminParamsOverride` and `configProperties`, enter the Admin CLI `show parameters -service admin` command as follows:

```
kv-> show parameters -service admin1
adminId=1
adminLogFileCount=20
adminLogFileLimit=5242880
adminMountPoint=/home/opc/nosql/kvroot
collectEnvStats=true
collectorPollPeriod=20 SECONDS
disabled=false
eventExpiryAge=30 DAYS
hideUserData=true
javaAdminParamsOverride=
loggingConfigProps=com.sleepycat.je.util.FileHandler.level=OFF
maxEvents=10000
storageNodeId=1
```

In this example, the `javaAdminParamsOverride` parameters that specify the Admin JVM memory shows the default value.

```
javaAdminParamsOverride=
```

> **Note:**
>
> The value may or may not have a default value. The value in your setup could also be different.

To increase Admin JVM memory when Admins are operational, use the `plan change-parameters` command from the Admin CLI, as follows:

```
kv-> plan change-parameters -wait -all-admins  -params \
javaAdminParamsOverride="-Xms2048m -Xmx2048m
```

> **Note:**
>
> It is recommended that you apply the modification to all of the admins by using the `all-admins` option. However, you can modify the admin parameters of every primary admin individually, but only if you have three or more primary admins. When you have only two admins, you must use the `all-admins` parameter; otherwise, the restart needed during this operation will cause the admins to lose the quorum.

Specifying these new values changes the Java heap size from the default values to 2 GB for both as shown below.

```
kv-> show parameters -service admin1
adminId=1
adminLogFileCount=20
adminLogFileLimit=5242880
adminMountPoint=/home/opc/nosql/kvroot
collectEnvStats=true
collectorPollPeriod=20 SECONDS
disabled=false
eventExpiryAge=30 DAYS
hideUserData=true
javaAdminParamsOverride=-Xms2048m -Xmx2048m
loggingConfigProps=com.sleepycat.je.util.FileHandler.level=OFF
maxEvents=10000
storageNodeId=1
```

Make sure that you locate the existing `javaAdminParamsOverride` from the Admin CLI as shown above, and update the individual entries. The `javaAdminParamsOverride` setting must represent all desired flags, not just new ones, so be sure to include any previously existing flag values that you want to retain.

If the Admin loses quorum, then you must use the Admin CLI `repair-admin-quorum` command.

Storage Node Parameters

You can set the following Storage Node parameters:

- `capacity=<Integer>`

  Sets the number of Replication Nodes that this Storage Node can host. This value informs decisions about where to place new Replication Nodes. The default value is `1`. You can set the capacity level to greater than `1` if the Storage Node has sufficient disk, CPU, and memory resources to support multiple Replication Nodes.

  Setting the Storage Node capacity to 0 indicates that the Storage Node can be used to host Arbiter Nodes. The pool of Storage Nodes in a zone configured to host Arbiter Nodes is used for Arbiter Node allocation. See Deploying an Arbiter Node Enabled Topology.

- `jvmOverheadPercent=<Integer>`

Sets the percentage of Java heap size, for additional memory used by JVM overhead. Default value: 25. In standard memory allocation, 85% of the SN's memory is for Java heap and JVM overhead: 68% for Java heap (`rnHeapPercent`), 25% (`jvmOverheadPercent`) * 68 (`rnHeapPercent`) = 17% for JVM overhead, and 68% + 17% = 85%.

- `memoryMB=<Integer>`

  Sets the amount of memory (in megabytes) available on this Storage Node. The default value is `0`, which indicates that the amount of memory is `unknown`. The store determines the amount of memory automatically as the total amount of RAM available on the machine.

  You should not need to change this parameter. If the machine has other applications running on it, reserve some memory for those applications, and set the `memoryMB` parameter value with a memory allowance for application needs. Having other applications running on a Storage Node is not a recommended configuration.

- `mgmtClass=<String>`

  The name of the class that provides the Management Agent implementation. See Standardized Monitoring Interfaces . The port cannot be a privileged port number (<1024).

- `numCPUs=<Integer>`

  Sets the number of CPUs known to be available on this Storage Node. Default value: 1.

- `rnHeapMaxMB=<Integer>`

  Sets a hard limit for the maximum size of the Replication Node's Java VM heap. The default value is `0`, which means the VM-specific limit is used. The default is roughly 32 GB, which represents the largest heap size that can make use of compressed object references.

  Do not set this value to greater than 32 GB. Doing so can adversely impact your Replication Node's performance.

  Settings larger than the maximum size that supports compressed object references will maintain the default limit unless the size is large enough that the heap can reference a larger number of objects given the increased memory requirements for uncompressed object references. Using larger heap sizes is not recommended.

- `rnHeapPercent=<Integer>`

  Sets the percentage of a Storage Node's memory reserved for heap space for all RN processes that the SN hosts. Default value: `68`.

- `rootDirPath=<path>`

  The path to the Storage Node's root directory.

- `rootDirSize=<Long Unit_String>`

  Sets the storage size of the root directory. However, no run-time checks are performed to verify that the actual directory size is greater than or equal to the size you specify. Use this setting for heterogeneous installation environments where some Storage Nodes have more disk capacity than others. Then, use this parameter only for those Storage Nodes that store data in the root directory (not recommended).

The value that you specify for this parameter must be a long, followed optionally by a unit string. Accepted unit strings are: KB, MB, GB, and TB, corresponding to 1024, 1024^2, 1024^3, 1024^4, respectively. Acceptable strings are case insensitive. Valid delimiters between the long value and the unit string are " ", "-", or "_".

> **Note:**
>
> The `rootDirSize` parameter is intended for backward compatibility with older installations that were created without specifying the `-storagedir` parameter. We strongly recommend not storing data in your root directory. See Managing Storage Directory Sizes. However, if you do specify a `-rootDirPath` parameter, you must also specify `-rootDirSize`. If you are trying to change parameter settings (`plan change-parameters`), and do not specify both parameters, a warning is displayed.

Do not use the `rootDir` parameter if a Storage Nodes uses some other directory (such as you can specify using `plan change-storagedir`).

The following store-wide parameter settings apply to statistics files and performance files, as well as the service debug logs across all Storage Nodes, Replication Nodes, Admins, and Arbiters. The associated Storage Node Agent must be restarted to reflect any changes in the settings.

- `serviceLogFileCount=<Integer>`

  Sets the number of log files kept by this Storage Node, and for all the Replication Nodes it hosts. This default value is `20`. Limiting the number of log files controls the amount of disk space devoted to logging history. If the value is less than 1, then it is converted to 1.

- `serviceLogFileLimit=<Integer>`

  Limits the size of each log file. After reaching this size, the logging subsystem starts a new log file. This setting applies to the Storage Node and to all Replication Nodes that it hosts. The default value is `2,000,000` bytes. The limit specifies an approximate maximum amount of bytes written to any one file. If the value is lesser than or equal to 0, then there is no limit to the size of the service log files.

- `serviceLogFileCompression=<Boolean>`

  Enables the compression of the log files to store significantly more logging output in the same amount of disk space. By default, the compression is disabled. You can enable log file compression by setting the parameter to `true`.

  When you enable compression, the `adminLogFileLimit` and `serviceLogFileLimit` parameters are auto-adjusted to retain larger log files than the specified size. With the default value of the maximum file count, the actual size limit is approximately five times larger than the specified limit.

  > **Note:**
  >
  > The size of the log files may temporarily exceed the defined limits in certain cases.

- `servicePortRange=<String>`

Sets the range of ports used for communication among administrative services running on a Storage Node and its managed services. This parameter is optional. By default the services use anonymous ports. The format of the value string is "*startPort,endPort*."

The range needs to be large enough to accommodate the Storage Node, all the Replication Nodes (as defined by the capacity parameter), Admin and Arbiter services hosted on the machine, and JMX, if enabled. The number of ports required also depends on whether the system is configured for security, which is the default. For a non-secure system, the Storage Node consumes 1 port (shared with the port assigned separately for the Registry Service, if it overlaps the service port range), and each Replication Node consumes 1 port in the range. An Admin, if configured, consumes 1 port. Arbiters consume 1 port each. If JMX is enabled, that consumes 1 additional port. On a secure system, two additional ports are required for the Storage Node, and two for the Admin. As a general rule, we recommend that you specify a range significantly larger than the minimum. More available ports allows for increases in Storage Node capacity, or network problems that can render ports temporarily unavailable.

The ports that you specify in the `servicePortRange` should not overlap with the Admin port or with `haPortRange`. The service port range can include the registry port, so the registry and Storage Node share a port.

For deploying a secure Oracle NoSQL Database, use the following formula to estimate the port range size number, adding an additional port for each Storage Node, Replication Node or the Admin (if configured):

```
 3 (Storage Nodes) +
capacity (the number of Replication Nodes) +
Arbiters (the number of Arbiter Nodes) +
3 (if the Storage Node is hosting an admin) +
1 (if the Storage node is running JMX)
```

For more information on configuring Oracle NoSQL Database securely, see *Security Guide*.

For a non-secure system, use the following formula to estimate the port range size number:

```
1 (Storage Node) +
capacity (the number of Replication Nodes) +
Arbiters (the number of Arbiter Nodes) +
1 (if the Storage Node is hosting an admin) +
1 (if the Storage Node is running JMX)
```

For example, if a Storage Node has capacity 1, is hosting an Admin process, and neither Arbiters nor JMX are in use, the range size must be at least 3. You can increase the range size beyond this minimum, for safety and Storage Node expansion. Then, if you expand the Storage Node, you will not need to make changes to this parameter. If capacity is `2`, the range size must be greater than or equal to 4.

Replication Node Parameters

The following parameters can be set for Replication Nodes:

- `cacheSize=<Long>`

  Sets the cache size in the underlying BDB JE subsystem. The units are bytes. The size is limited by the java heap size, which in turn is limited by the amount of memory available on the machine. You should only ever change this low level parameter under explicit directions from Oracle support.

- `collectEnvStats=<Boolean>`

  If true, then the underlying BDB JE subsystem dumps statistics into the .stat file. This information is useful for tuning JE performance. Oracle Support may request these statistics to aid in tuning or to investigate a problem.

- `configProperties=<String>`

  Contains property settings for the underlying BDB JE subsystem. Its format is `property=value;property=value....`

- `enableErasure=<Boolean>`

  If true, then erasure is enabled for the underlying storage system. Erasure periodically wipes the obsolete data (i.e. delete data, older versions of updated records, expired data) from the storage layer by zeroing out the corresponding records. Erasure can be enabled or disabled without restarting the database. Erasure is enabled by default.

  Default value is true.

- `erasurePeriod=<Long Timeunit>`

  The duration for one complete erasure pass over the entire data set (the cycle time). Erasure is throttled based on this value, to minimize its impact on performance. It is recommended that erasure period be set to less than half of the duration one expects the obsoleted data to stay around. In other words, we recommend two erasure cycles to remove the obsoleted data. For example, if we intend to remove all obsolete data in 30 days, then erasure period can be set to 14 days.

  Default value is "6 DAYS".

- `javaMiscParams=<String>` [deprecated]

  This parameter should ONLY be used to set flags for which there are no service parameters.

  The `javaMiscParams` parameter is **deprecated**. You are encouraged not to use it, and instead use the `javaRnParamsOverride` parameter.

- `javaRnParamsOverride=<String>`

  A string that is added to the command line when the Replication Node process is started. This parameters is intended for specifying miscellaneous JVM properties that cannot be specified using other RN parameters. If the string is not a valid sequence of tokens for the JVM command line, the Admin process fails to start.

  No default value is provided for this parameter.

  It is recommended that to specify the heap sizes for Replication Nodes you use Storage Node's memoryMB and other JVM parameters. For more information about these parameters see, Storage Node Parameters.

- `latencyCeiling=<Integer>`

  If the Replication Node's average latency exceeds this number of milliseconds, it is considered an "alertable" event. If JMX monitoring is enabled, the event also causes an appropriate notification to be sent.

- `loggingConfigProps=<String>`

  Contains property settings for the Logging subsystem. The format of this string is like that of configProperties, above. Standard java.util.logging properties can be set by this parameter.

- `maxTrackedLatency=<Long TimeUnit>`

  The highest latency that is included in the calculation of latency percentiles.

- `rnCachePercent=<Integer>`

  The portion of an RN's memory set aside for the JE environment cache.

- `rnStatisticsEnabled=<Boolean>`

  If true, then the Replication Nodes gather key distribution statistics.

- `rnStatisticsGatherInterval=<Long TimeUnit>`

  The time interval at which Replication Nodes should gather distribution statistics.

- `rnStatisticsTTL=<Long DaysOrHours>`

  Specifies the duration for which the key distribution statistics should be retained in the system tables. The duration specified must be in days or hours. By default, these statistics are retained for 60 days.

- `rnStatisticsIncludeStorageSize=<Boolean>`

  If true, then the information on storage sizes are included when gathering key distribution statistics.

- `throughputFloor=<Integer>`

  Similar to latencyCeiling, throughputFloor sets a lower bound on Replication Node throughput. Lower throughput reports are considered alertable. This value is given in operations per second.

Arbiter Node Parameters

The following parameters can be set for Arbiter Nodes:

`javaAnParamsOverride=<String>`

A string that is added to the command line when the Arbiter Node process is started. This parameters is intended for specifying miscellaneous JVM properties. If the string is not a valid sequence of tokens for the JVM command line, the Admin process fails to start.

No default value is provided for this parameter.

It is recommended that to specify the heap sizes for Arbiter Nodes you use Storage Node's memoryMB and other JVM parameters. For more information about these parameters see, Storage Node Parameters.

Global Parameters

The following store-wide non-security parameters can be implicitly and uniformly set across all Storage Nodes, Replication Nodes and Admins:

- `collectorInterval =<Long TimeUnit>`

Sets the collection period for latency statistics at each component. This value defaults to 20 seconds. Values like average interval latencies and throughput are averaged over this period of time.

The following store-wide parameters can be set for the debug log files:

- `adminLogFileCount=<Integer>`

  Sets the number of log files that are kept. This value defaults to 20. It is used to control the amount of disk space devoted to logging history. If the value is less than 1, then it is converted to 1.

- `adminLogFileLimit=<Integer>`

  Limits the size of log files. After reaching this limit, the logging subsystem switches to a new log file. This value defaults to 4,000,000 bytes. The limit specifies an approximate maximum amount to write (in bytes) to any one file. If the value is lesser than or equal to 0, then there is no limit to the size of the log files.

Security Parameters

The following store-wide security parameters can be implicitly and uniformly set across all Storage Nodes, Replication Nodes and Admins:

- `accountErrorLockoutThresholdCount=<Integer>`

  Number of invalid login attempts for a user account from a particular host address over the tracking period needed to trigger an automatic account lockout for a host. The default value is 10 attempts.

- `accountErrorLockoutThresholdInterval=<Long TimeUnit>`

  Specifies the time period over which login error counts are tracked for account lockout monitoring. The default value is 10 minutes.

- `accountErrorLockoutTimeout=<Long TimeUnit>`

  Time duration for which an account will be locked out once a lockout has been triggered. The default value is 30 minutes.

- `loginCacheTimeout=<Long TimeUnit>`

  Time duration for which KVStore components cache login information locally to avoid the need to query other servers for login validation on every request. The default value is 5 minutes.

- `sessionExtendAllowed=<Boolean>`

  Indicates whether session extensions should be granted. Default value is true.

- `sessionTimeout=<Long TimeUnit>`

  Specifies the length of time for which a login session is valid, unless extended. The default value is 24 hours.

The following password security parameters can be set:

Parameter Name	Value Range and Type	Description
passwordAllowedSpecial	Sub set or full set of #$ %&'()*+,-./:; <=>?@[]^_`{|} (string)~	Lists the allowed special characters.

Parameter Name	Value Range and Type	Description
passwordComplexityCheck	[true|false] (boolean)	Whether to enable the password complexity checking. The default value is true.
passwordMaxLength	1 - 2048 (integer)	The maximum length of a password. The default value is 256.
passwordMinDigit	0 - 2048 (integer)	The minimum required number of numeric digits. The default value is 2.
passwordMinLength	1 - 2048 (integer)	The Minimum length of a password. The default value is 9.
passwordMinLower	0 - 2048 (integer)	The minimum required number of lower case letters. The default value is 2.
passwordMinSpecial	0 - 2048 (integer)	The minimum required number of special characters. The default value is 2.
passwordMinUpper	0 - 2048 (integer)	The minimum required number of upper case letters. The default value is 2.
passwordNotStoreName	[true|false] (boolean)	If true, password should not be the same as current store name, nor is it the store name spelled backwards or with the numbers 1–100 appended. The default value is true.
passwordNotUserName	[true|false] (boolean)	If true, password should not be the same as current user name, nor is it the user name spelled backwards or with the numbers 1-100 appended. The default value is true.
passwordProhibited	list of strings separated by comma (string)	Simple list of words that are not allowed to be used as a password. The default reserved words are: oracle,password,user,nosql.
passwordRemember	0 - 256 (integer)	The maximum number of passwords to be remembered that are not allowed to be reused when setting a new password. The default value is 3.

For more information on top-level, transport, and password security parameters see the *Security Guide*.

Admin Restart

Changes to the following Oracle NoSQL Database parameters will result in a Admin restart by the Storage Node Agent:

Admin parameters:

- adminHttpPort

- adminLogFileCount
- adminLogFileLimit
- configProperties
- javaAdminParamsOverride
- javaMiscParams (**deprecated**)
- loggingConfigProps

For example:

```
kv-> plan change-parameters -all-admins
-params adminLogFileCount=10
Started plan 14. Use show plan -id 14 to check status.
        To wait for completion, use plan wait -id 14
kv-> show plan -id 14
Plan Change Admin Params (14)
Owner: null
State:                 INTERRUPTED
Attempt number:        1
Started:               2013-08-26 20:12:06 UTC
Ended:                 2013-08-26 20:12:06 UTC
Total tasks:           4
 Successful:           1
 Interrupted:          1
 Not started:          2
Tasks not started
   Task StartAdmin start admin1
   Task WaitForAdminState waits for Admin admin1 to reach RUNNING state
kv-> plan execute -id 14
Started plan 14. Use show plan -id 14 to check status.
        To wait for completion, use plan wait -id 14
kv-> show plan -id 14
Plan Change Admin Params (14)
State:                 SUCCEEDED
Attempt number:        1
Started:               2013-08-26 20:20:18 UTC
Ended:                 2013-08-26 20:20:18 UTC
Total tasks:           2
 Successful:           2
```

> **Note:**
>
> When you change a parameter that requires an Admin restart using the `plan change-parameters` command, the plan ends in an `INTERRUPTED` state. To transition it to a `SUCCESSFUL` state, re-issue the plan a second time using the `plan execute -id <id>` command.

Replication Node Restart

The Storage Node Agent must be restarted to reflect any changes in the setting of the following parameters.

Storage Node parameters:

- serviceLogFileCount
- serviceLogFileLimit
- serviceLogFileCompression
- servicePortRange

Replication Node parameters:

- configProperties
- javaMiscParams (**deprecated**)
- javaRnParamsOverride
- loggingConfigProps

Removing an Oracle NoSQL Database Deployment

There are no scripts or tools available to completely remove an Oracle NoSQL Database installation from your hardware. However, the procedure is simple. On each node (machine) comprising your store:

1. Shut down the Storage Node:

   ```
   java -Xmx64m -Xms64m \
   -jar KVHOME/lib/kvstore.jar stop -root KVROOT
   ```

   Note that if an Admin process is running on the machine, this command also stops that process.

2. Physically remove the entire contents of KVROOT:

   ```
   > rm -rf KVROOT
   ```

3. Empty the contents of all the storage directories configured for the KVStore. For example, if you configured three storage directories using the `makebootconfig` utility, you must clean up all the three storage directories.

   ```
   cd /disk1
   rm -rf *
   ```

Once you have performed this procedure on every machine comprising your store, you have completely removed the Oracle NoSQL Database deployment from your hardware.

Modifying Storage Node HA Port Ranges

When you initially configured your installation, you defined a range of ports for the nodes to use when communicating between themselves. (You did this in Installation Configuration Parameters.) This range of ports is called the *HA port range*, where *HA* is an acronym for High Availability, and indicates your store's replication factor.

If you inadvertently used invalid values for the HA Port Range, you cannot deploy a Replication Node (RN) or a secondary Administration process (Admin) on any Storage Node. You will discover the problem when you first attempt to deploy a store with a Replication node. Following are indications that the Replication Node did not come up on the Storage Node:

- The Admin logs include an error that the Replication Node is in the `ERROR_RESTARTING` state. After a number of retries, the warning error changes to `ERROR_NO_RESTART`. You can find the Replication Node state in the `ping` command output.
- The plan enters an `ERROR` state. Using the CLI's `show plan <planID>` command to get more history details includes an error message like this:

  ```
  Attempt 1
          state: ERROR
          start time: 10-03-11 22:06:12
          end time: 10-03-11 22:08:12
          DeployOneRepNode of rg1-rn3 on sn3/farley:5200 [RUNNING]
          failed. ....  Failed to attach to RepNodeService for rg1-rn3,
          see log, /KVRT3/<storename>/log/rg1-rn3*.log, on host
          farley for more information.
  ```

- The critical events mechanism, accessible through the Admin CLI `show events` command, includes an alert containing the same error information from the plan history.
- The store's runtime or boot logs for the Storage Node and/or Admin shows a port specific error message, such as:

  ```
  [rg1-rn3] Process exiting
  java.lang.IllegalArgumentException: Port number 1 is invalid because
  the port must be outside the range of "well known" ports
  ```

You can address incorrect HA port ranges in a configuration by completing the following steps. Steps that require you to execute them on the physical node hosting the Oracle NoSQL Database Storage Node, begin with the directive *On the Storage Node*. You can execute other steps from any node that can access the Admin CLI.

1. Using the Admin CLI, cancel the `plan deploy-sn` or `plan deploy-admin` command that includes invalid HA Port Range values.
2. On the Storage Node, kill the existing, incorrectly configured `StorageNodeAgentImpl` process and all of its Managed Processes. You can distinguish managed processes from other processes because they have the parameter `-root <KVROOT>`.
3. On the Storage Node, remove all files from the KVROOT directory.
4. On the Storage Node, recreate the storage node bootstrap configuration file in the KVROOT directory. For directions, see Installation Configuration Parameters.
5. On the Storage Node, restart the storage node using this Java command:

   ```
   java -Xmx64m -Xms64m
   -jar KVHOME/lib/kvstore.jar restart
   ```

6. Using the Admin CLI, you can now create and execute a `deploy-sn` or `deploy-admin` plan, using the same parameters as the initial plan, but with the correct HA range.

Modifying Storage Node Service Port Ranges

This section explains how to modify your Storage Node service port ranges after an initial configuration and deployment.

When you initially configure your installation, you specify a range of ports that your Storage Node's Replication Nodes and Admin services use. These ports are collectively called the *service port ranges*. Configuring them at installation time was optional. If you did not configure them, the configuration scripts automatically selected a range of ports for you.

The process of modifying your service port range depends on whether the Storage Node has already been deployed. You can determine whether a Storage Node has been deployed by using the Command Line Interface (CLI) to run the `show topology` command. (See show topology for details). The `show topology` command lists the Storage Node, along with the host and port if it has been deployed.

Storage Node Not Deployed

Use this process to modify your Service Port Ranges if the Storage Node has been configured but not deployed.

Execute the following steps on the Storage Node host:

1. Kill the existing Storage Node process. You can find the ID of this process by using:

   ```
   ps -af | grep -e "kvstore.jar.*start.*<KVROOT>"
   ```

   Kill the process using:

   ```
   kill <storage node id>
   ```

2. Remove all the files from the <KVROOT> directory.

   ```
   rm -rf <KVROOT>/*
   ```

3. Recreate the Storage Node bootstrap configuration file with the updated service port ranges, being sure to specify the `-servicerange` parameter. For example:

   ```
   java -Xmx64m -Xms64m \
   -jar <KVHOME>/lib/kvstore.jar makebootconfig -root <KVROOT> \
   -port <port> -host <host> -harange <harange> \
   -servicerange <startPort, endPort>
   ```

   See makebootconfig for details on using this utility.

4. Restart the Storage Node:

   ```
   java -Xmx64m -Xms64m -jar <KVHOME>/lib/kvstore.jar restart
   ```

You can proceed to deploy the Storage Node using the Admin CLI. It will use the newly specified service port range.

Storage Node Deployed

Use this process to modify your Service Port Ranges if the Storage Node has been deployed.

1. Using the Admin CLI, modify the service port range using the `plan change-parameters` command. Specify `servicePortRange` while you do. For example:

   ```
   plan change-parameters -service <id> \
   -params servicePortRange=<startPort,endPort>
   ```

   `servicePortRange` is described in Storage Node Parameters.

2. Restart the Storage Node process and its services. The Replication Nodes and any admin services for the Storage Node can be stopped in an orderly fashion using the CLI. Use the `show topology` command (show topology) to list all the services associated with the Storage Node.

   Stop each of these services using the `plan stop-service` command. See plan stop-service for details on this command. Note that when you stop a service, you must use the services ID, which you can find from the output of the `show topology` command. Keep track of these IDs because you will need them when you restart the Storage Node.

   Repeat until all services for the Storage Node have been stopped.

3. Kill the existing Storage Node process. You can find the ID of this process by going to the Storage Node host and issuing:

   ```
   ps -af | grep -e "kvstore.jar.*start.*<KVROOT>"
   ```

   Kill the process using:

   ```
   kill <storage node id>
   ```

   > **Note:**
   >
   > Avoid killing all Replication Nodes in your store at the same time, as doing so will result in unexpected errors.

4. Restart the Storage Node by going to the Storage Node host and issuing:

   ```
   java -Xmx64m -Xms64m -jar <KVHOME>/lib/kvstore.jar restart
   ```

5. Restart the Storage Node services by using `plan start-service` for each service on the Storage Node. See plan start-service for details.

6. When the Storage Node is restarted and all its Replication Nodes and any admin services are running, the services will be using the updated service port range. You can check by first locating the process ID of the Storage Node services using this command:

   ```
   ps -af | grep -e "ManageService.*<KVROOT>"
   ```

and then check the ports the services are listening to by using this command:

```
netstat -tlpn | grep <id>
```

One of the listening ports is the service port and it should be within the new range.

Availability, Failover and Switchover

Topics:

- Availability and Failover
- Replication Overview
- Loss of a Read-Only Replica Node
- Loss of a Read/Write Master
- Unplanned Network Partitions
- Failover and Switchover Operations
- Zone Failover
- Durability Summary
- Consistency Summary

Availability and Failover

Oracle NoSQL Database is a data storage product with enormous scalability and performance benefits. Additionally, Oracle NoSQL Database offers excellent *availability* mechanisms. These mechanisms are designed to provide your applications access to data contained in the store in the event of localized hardware and network failures.

This document describes the mechanisms Oracle NoSQL Database uses to ensure your data remains available, along with the various failover algorithms that Oracle NoSQL Database employs. In addition, this document describes application design patterns you can use to best make use of Oracle NoSQL Database's availability mechanisms. In some cases, tradeoffs exist between ensuring data is highly available, and achieving optimal performance. This document explores these tradeoffs.

The intended audience for this document includes system architects, engineers, and others who want to understand the concepts and issues surrounding data availability when using Oracle NoSQL Database. In addition, software engineers responsible for writing code that interacts with an Oracle NoSQL Database store should also read this document.

We recommend that you read and get familiar with the following contents before continuing.

- Developers Guide

  This document introduces terms and concepts you need to know before reading this document.

- Durability Guarantees in the *Java Direct Driver Developer's Guide*

  This section includes concepts that lead to issues surrounding write availability.

- Consistency Guarantees in the *Java Direct Driver Developer's Guide*

  This section includes concepts that lead to issues surrounding read availability.

Replication Overview

To ensure data durability and availability, Oracle NoSQL Database uses a single-master replication strategy. Using a single machine to perform write operations, Oracle NoSQL Database then broadcasts those operations to multiple read-only replicas.

The *Concepts Guide* describes a shard as a collection of replication nodes, associated with a single master node and multiple replicas. Your store contains multiple shards, and your data is spread evenly across all of the shards that your store uses.

When you perform a write operation in your store, Oracle NoSQL Database completes the write operation on the master node in use by the shard containing your data. The master node performs this write according to whatever durability guarantees are in place at the time. If you set a strong durability guarantee, the master requires the participation of some or all of the replicas in the shard to complete the write operation.

If the master node of the shard becomes unavailable for any reason, the replica nodes in primary zones hold an election to determine which of the remaining replication nodes should take over as the master node. The replication node with the most up-to-date data wins the election.

The election is decided based on a simple majority vote. This means that a majority of the nodes in the shard in primary zones must be available to participate in the election to select a new master.

Loss of a Read-Only Replica Node

A common fail over case is losing a replica node due to a problem with the machine upon which it is running. This loss can be due to something as common as a hard drive failure.

In this case, the only shard that is affected is the one using the replica. By default, the effect on the shard is reduced read throughput capacity. The shard itself is capable of continuing normal operations. However, losing a single Replication Node reduces its capacity to service read requests by whatever read throughput a single host machine offers your store. Whether you detect this reduction in read throughput capacity depends on how heavy a read load your shard is experiencing. The shard could have a low enough read load that losing the replica results in a minor performance reduction.

Such a small performance reduction assumes that a single host machine contains only one Replication Node. If you configure your store so that multiple Replication Nodes run on a single host, then the loss of throughput capacity increases accordingly. It is likely that the loss of a machine running multiple Replication Nodes will affect the throughput capacity of more than one shard, because it is unlikely that all the Replication Nodes on that machine will belong to the same shard. Again, whether you notice any performance reduction from the loss of the Storage Node depends on how heavy a read load the individual affected shards are experiencing.

In this scenario, with one exception, the shard will continue servicing write requests, and may be able to do so with no changes to its write throughput capacity. The master itself is not affected, so it can continue performing writes and replicating them to the remaining replicas in the shard. There can be reduced write throughput capacity if:

- there is such a heavy read load on the shard that the loss of one replica saturates the remaining replica(s); and
- the master requires an acknowledgement before finishing a write commit.

In this scenario, write performance capacity can be reduced either because the master is continually waiting for the replica to acknowledge commits, or because the master itself is expending resources responding to read requests. In either case, you may see degraded write throughput, but the level of degradation depends on how heavy the read/write load actually is on the shard. Again, it is possible that you will never detect any write throughput reduction, because the write load on the shard is low.

In addition, the loss of a single read-only replica can cause all write operations at that shard to fail with a `DurabilityException` exception. This happens if you are using a durability guarantee that requires acknowledgements from all replicas in the shard in primary zones. In this case, writes at that shard will fail until either that replica is brought back online, or you place a less strict durability guarantee into use.

Using durability guarantees that require acknowledgements from all replicas in primary zones offer you the strongest data durability possible (by making certain that your writes are replicated to every machine in a shard). At the same time, they have the potential to lose write capabilities for an entire shard from a single hardware failure. Consequently, be sure to balance your durability requirements against your availability requirements, and configure your store and related code accordingly.

Loss of a Read/Write Master

If you lose a host machine containing a shard's master, the shard will be incapable of responding to write requests, momentarily. The lack of write request response is so brief that it may not be detected by your client code. Only the shard containing the master is affected by this outage. All other shards continue to perform as normal.

In this case, the shard's replicas in primary zones will quickly notice the master is missing and call for an election. Typically this will occur within a few milliseconds after losing the master.

The replica nodes will conduct an election, and the replica in a primary zone with the most up-to-date set of data will be elected master. To be elected master requires a simple majority vote from the other machines in the shard hosting nodes in primary zones. Keep in mind that this simple majority requirement has implications if many machines are lost from your store.

Once a new master is elected, the shard will continue operations, reducing its read throughput capacity by one machine. As with the loss of a single replica (see the previous section), all write operations can continue as long as your durability guarantee does not require acknowledgements from all replicas in primary zones.

Your client code will not notice the missing master if the new master is elected and services the write request within the timeout value used for the write operation. However, we recommend that your production code include ways to guard against timeout problems. In the event of a timeout, your code should include a decision policy about what to do next. For example, your policy could:

- Retry the write operation immediately,
- Retry the write operation after a defined wait,
- Abandon the write operation entirely.

Unplanned Network Partitions

A shard can be split into two, non-communicating networks. Such an event can occur when a piece of network hardware, such as a router, fails in some way that divides the shard. The store's response to such an event depends on how the network partition divides the shard's Replication Nodes as in these three cases:

A single Replication Node is isolated from the rest of the shard. If the Replication Node is a read-only replica, the shard continues operating as normal, but without the read throughput capacity caused by the loss of a single machine. See Loss of a Read-Only Replica Node for more details.

A single Replication Node becomes isolated from the rest of the shard. If the Replication Node is a master, the shard handles the event in the same way as if it had lost a master. The shard holds an election to select a new master and then continues operating as normal. See Loss of a Read/Write Master for further information.

The new network partition divides the shard into two or more groups of machines. In this case, there will be at least one *minority node partition.* A minority node partition contains less than a majority of the Replication Nodes in the shard. There could also be a *majority node partition.* A majority node partition has the majority of nodes in the shard —. However, a majority node partition is not a given, especially if the new network partition creates more than two sets of Replication Nodes.

How failover is handled in this scenario depends on whether a majority node partition does exist, and if the master exists in that partition. There are also other issues to consider, such as the durability and consistency policies that were in use at the time the new network partition was created.

Master is in the Majority Node Partition

Suppose the shard is divided into two partitions. Partition A contains a simple majority of the Replication Nodes in primary zones, including the master. Partition B has the remaining nodes.

- Partition A continues to service read and write requests as normal, but with a reduced read throughput from the loss of however many Replication Nodes are in Partition B. A caveat in this situation is what durability policy is in use at the time. If Partition A does not have enough replicas from primary zones to meet the durability policy requirements, it could be prevented from servicing write requests. If the durability policy requires a simple majority, or less, of replicas, then the shard will be able to service write requests.
- Partition B continues to service read requests as normal, but with increasingly stale data. Depending on the consistency guarantee in place, Partition B might cease to service read requests. If a version-based consistency is in use, then Partition B will probably encounter `ConsistencyException` exceptions soon after the network partition occurs, due to its inability to obtain version tokens from the master. Similarly, if a time-based consistency policy is in use, then `ConsistencyException` exceptions will occur as soon as the replica lags too far behind the master, from which it is no longer receiving write updates. By default, a consistency guarantee is not required to service read requests. So unless you explicitly create and use a consistency policy, Partition B can continue to service read requests through the entire network outage.

  Partition B will attempt to elect a new master, but will be unable to do so because it does not contain the simple majority of Replication Nodes required to hold an election.

Further, if the partition is such that your client code can reach Partition A but not Partition B, then the shard will continue to service read and write requests as normal, but with a reduced read capacity.

However, if the partition is such that your client code can read Partition B but not Partition A, then the shard will be unable to service any write requests. This is because Partition A contains the master, and Partition B does not include enough Replication Nodes to elect a new master.

Master is in the Minority Node Partition

Suppose the shard is divided into two partitions. Partition A contains a simple majority of the Replication Nodes from primary zones, but NOT the master. Partition B has the remaining nodes, including the master.

Assuming both partitions are network accessible by your client code, then:

- Partition A will notice that it no longer has a master. Because Partition A has at least a simple majority of the Replication Nodes in primary zones, it will be able to elect a new master. It will do this quickly, and the shard will continue operations as normal.

  Whether Partition A can service write requests is determined by the durability policy in use. As long as the durability policy requires a simple majority, or less, of replicas, then the shard is able to service write requests.

- Partition B will continue to operate as normal, believing that it has a valid master. However, the only way Partition B can service write requests is if the durability policy in use requires no participation from the shard's replicas. If a majority of nodes in primary zones must acknowledge the write operation, or if all nodes in primary zones must acknowledge the write, then the partitions will be unable to service writes because not enough nodes are available to satisfy the durability policy.

  If durability NONE is in use, then for the period of time that it takes to resolve the network partition, the shard will operate with two masters. When the partition is resolved, the shard will recognize the problem and correct it. Because Partition A held a valid election, writes performed there will be kept. *Any writes performed in Partition B will be discarded.* The old master in Partition B will be demoted to a simple replica, and the replicas in Partition B will all be synced with the new master.

  > **Note:**
  >
  > Because of the potential for loss of data in this scenario, Oracle *strongly* recommends that you do NOT use durability NONE. The only time you should use that durability setting is if you want to absolutely maximize write throughput, and do not care if you lose the data.

Further, if the partition is such that your client code can reach Partition A but not Partition B, then the shard will continue to service read and write requests as normal, but only after an election is held, and then with a reduced read capacity.

However, if the partition is such that your client code can read Partition B but not Partition A, then the shard will be unable to service write requests at all, unless you use the weakest durability policy available. This is because Partition B does not

include enough Replication Nodes to satisfy anything other than the weakest available durability policy.

No Majority Node Partition

Suppose the shard is divided into multiple partitions, and no partition contains a majority of the Replication Nodes in the shard. In this case, the shard's partitions can service read requests, so long as the consistency policy in use for the read supports it. If the read requires tight consistency with the master, and the master is not available to ensure the consistency can be met, then the read will fail.

The partition containing the master can service write requests only if you are using the weakest available durability policy, in which no acknowledgements from replicas are required. If acknowledgements are required, then there will not be enough replicas to satisfy the durability policy and no write operations can occur.

Once the network partition is resolved, the shard will elect a new master, synchronize all replicas with it, and continue operations as normal.

Failover and Switchover Operations

Optimal use of available physical datacenters is achieved by deploying your store across multiple zones. This provides fault isolation as each zone has a copy of your complete store, including a copy of all the shards. With this configuration, when a zone fails, write availability is automatically reestablished as long as quorum is maintained.

> **Note:**
>
> To achieve other levels of fault isolation, best practices for data center design should be applied. For example, site location, building selection, floor layout, mechanical design, electrical system design, modularity, etc.

However, if quorum is lost, manual procedures such as failovers can be used instead to recover from zone failures. For more information on quorum, see *Concepts Guide*.

A failover is typically performed when the primary zone fails or has become unreachable and one of the secondary zones is transitioned to take over the primary role. Failover can also be performed to reduce the quorum to the available primary zones. Failover may or may not result in data loss.

Switchovers can be used after performing a failover (to restore the original configuration) or for planned maintenance.

A switchover is typically a role reversal between a primary zone and one of the secondary zones of the store. A switchover can also be performed to convert one or more zones to another type for maintenance purposes. Switchover requires quorum and guarantees no data loss. It is typically done for planned maintenance of the primary system.

In this chapter, we explain how failover and switchover operations are performed.

> **Note:**
>
> Arbiters are not currently supported during failover and switchover operations.

Repairing a Failed Zone

If a zone fails but quorum is maintained, you have the option to repair the failed zone with new hardware by following the procedure described in Repairing a Failed Zone by Replacing Hardware.

Another option is to convert the failed zone to a secondary zone. In some cases, this approach can improve the high availability characteristics of the store by reducing the quorum requirements.

For example, suppose a store consists of two primary zones: zone 1 with a replication factor of three and zone 2, with a replication factor of two. Additionally, suppose zone 2 fails. In this case, quorum is maintained because you would have 3 out of the 5 replicas, but any additional failure would result in a loss of quorum.

Converting zone 2 to a secondary zone would reduce the primary replication factor to 3, meaning that each shard could tolerate an additional failure.

You should determine if switching zone types would actually improve availability. If so, then decide if it is worth doing in the current circumstances.

Need for an Admin node in the secondary zone:

Having admins in a secondary zone is very useful to support failure recovery. For example, if a store has primary and secondary zones, and all of the primary zones are lost, the administrator can use the repair-admin-quorum and plan failover commands to resume operations by converting the secondary zone to a primary zone. But these operations can occur only if an Admin node is available. For this reason, stores with secondary zones should include Admins in the secondary zones.

The recommendation is to deploy the same number of admins as the replication factor for the zone. For example if you have primary and secondary zone with a replication factor of 3, then each zone should be configured with three admins. If a zone failure occurs and no admins remain available, the failover procedures cannot be used. To avoid this situation you need to configure as many admins as the replication factor for the zone.

Performing a Failover

If quorum is maintained, you do not need to do anything because the store is still performing normally.

In situations where a zone fails but quorum is lost, your only option is to perform a failover.

For example, suppose a store consists of two zones, "Manhattan" and "JerseyCity", each deployed in its own physical data center.

> **Note:**
>
> This example uses a store with a replication factor of three. In this case, each zone is also configured with three admins.

Additionally, suppose that the "Manhattan" zone fails, resulting in the failure of all of the associated Storage Nodes and a loss of quorum. In this case, if the host hardware of "Manhattan" was irreparably damaged or the problem will take too long to repair you may choose to initiate a failover.

The following steps walk you through the process of verifying failures, isolating Storage Nodes, and reducing admin quorum to perform a failover operation. This process allows service to be continued in the event of a zone failure.

1. Connect to the store. To do this, connect to an admin running in the JerseyCity zone:

   ```
   java -Xmx64m -Xms64m -jar KVHOME/lib/kvstore.jar \
   runadmin -host jersey1 -port 6000 \
   -security USER/security/admin.security
   ```

   > **Note:**
   >
   > This assumes that you must have followed the steps as mentioned in Create users and configure security with remote access .

2. Use the `verify configuration` command to confirm the failures. The output confirms the Storage Node Agents in the Manhattan zone are unavailable.

   ```
   kv-> verify configuration
   Connected to Admin in read-only mode
   Verify: starting verification of store mystore based upon topology
   sequence #115
   100 partitions and 6 storage nodes
   Time: 2022-06-09 07:15:23 UTC   Version: 21.3.10
   See jersey1:/kvroot/mystore/log/mystore_{0..N}.log for progress messages
   Verify: Shard Status: healthy: 0 writable-degraded: 0 read-only: 1
   offline: 0 total: 1
   Verify: Admin Status: read-only
   Verify: Zone [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
   masterAffinity=false] RN Status: online: 0 read-only: 0 offline: 3
   Verify: Zone [name=JerseyCity id=zn2 type=SECONDARY allowArbiters=false
   masterAffinity=false]   RN Status: online: 0 read-only: 3 offline: 0
   Verify: == checking storage node sn1 ==
   Verify:  sn1: ping() failed for sn1 :
   Unable to connect to the storage node agent at host nyc1,
   port 5000, which may not be running; nested exception is:
       java.rmi.ConnectException: Connection refused to host:
       nyc1; nested exception is:
       java.net.ConnectException: Connection refused
   Verify: Storage Node [sn1] on nyc1:5000
   ```

```
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
  UNREACHABLE
Verify:  admin1: ping() failed for admin1 :
Unable to connect to the storage node agent at host nyc1,
port 5000, which may not be running; nested exception is:
    java.rmi.ConnectException: Connection refused to host:
    nyc1; nested exception is:
    java.net.ConnectException: Connection refused
Verify:  Admin [admin1]      Status: UNREACHABLE
Verify:  rg1-rn1: ping() failed for rg1-rn1 :
Unable to connect to the storage node agent at host nyc1,
port 5000, which may not be running; nested exception is:
    java.rmi.ConnectException: Connection refused to host:
    nyc1; nested exception is:
    java.net.ConnectException: Connection refused
Verify:  Rep Node [rg1-rn1]  Status: UNREACHABLE
Verify: == checking storage node sn2 ==
Verify:  sn2: ping() failed for sn2:
Unable to connect to the storage node agent at host nyc1,
port 5100, which may not be running; nested exception is:
    java.rmi.ConnectException: Connection refused to host:
    nyc1; nested exception is:
    java.net.ConnectException: Connection refused
Verify: Storage Node [sn2] on nyc1:5100
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
  UNREACHABLE
Verify:  admin2: ping() failed for admin2:
Unable to connect to the storage node agent at host nyc1,
port 5100, which may not be running; nested exception is:
    java.rmi.ConnectException: Connection refused to host:
    nyc1; nested exception is:
    java.net.ConnectException: Connection refused
Verify:   Admin [admin2]      Status: UNREACHABLE
Verify:   rg1-rn2: ping() failed for rg1-rn2 :
Unable to connect to the storage node agent at host nyc1,
port 5100, which may not be running; nested exception is:
    java.rmi.ConnectException: Connection refused to host:
    nyc1; nested exception is:
    java.net.ConnectException: Connection refused
Verify: Rep Node [rg1-rn2]  Status: UNREACHABLE
Verify: == checking storage node sn3 ==
Verify:  sn3: ping() failed for sn3:
Unable to connect to the storage node agent at host nyc1,
port 5200, which may not be running; nested exception is:
    java.rmi.ConnectException: Connection refused to host:
    nyc1; nested exception is:
    java.net.ConnectException: Connection refused
Verify: Storage Node [sn3] on nyc1:5200
Zone: [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
  UNREACHABLE
Verify:  admin3: ping() failed for admin3:
Unable to connect to the storage node agent at host nyc1,
```

```
port 5200, which may not be running; nested exception is:
    java.rmi.ConnectException: Connection refused to host:
    nyc1; nested exception is:
    java.net.ConnectException: Connection refused
Verify:  Admin [admin3]      Status: UNREACHABLE
Verify: rg1-rn3: ping() failed for rg1-rn3 :
Unable to connect to the storage node agent at host nyc1,
port 5200, which may not be running; nested exception is:
    java.rmi.ConnectException: Connection refused to host:
    nyc1; nested exception is:
    java.net.ConnectException: Connection refused
Verify: Rep Node [rg1-rn3]  Status: UNREACHABLE
Verify: == checking storage node sn4 ==
Verify: Storage Node [sn4] on jersey1:6000
Zone: [name=JerseyCity id=zn2 type=SECONDARY allowArbiters=false
masterAffinity=false]
Ver: 21.3.10 2021-12-21 21:24:59 UTC  Build id: 78bbc4cb976b Edition:
Enterprise isMasterBalanced: true   serviceStartTime: 2022-06-09 07:05:44
UTC
Verify:  Admin [admin4]
Status: RUNNING,MASTER (non-authoritative)
Verify:  Rep Node [rg1-rn4]
Status: RUNNING,MASTER (non-authoritative) sequenceNumber:217 haPort:6003
available storage size:12 GB
Verify: == checking storage node sn5 ==
Verify: Storage Node [sn5] on jersey1:6100
Zone: [name=JerseyCity id=zn2 type=SECONDARY allowArbiters=false
masterAffinity=false]
Ver: 21.3.10 2021-12-21 21:24:59 UTC  Build id: 78bbc4cb976b Edition:
Enterprise isMasterBalanced: true   serviceStartTime: 2022-06-09 07:05:44
UTC
Verify:  Admin [admin5]
Status: RUNNING,MASTER (non-authoritative)
Verify:  Rep Node [rg1-rn5]
Status: RUNNING,MASTER (non-authoritative) sequenceNumber:217 haPort:6003
available storage size:12 GB
Verify: == checking storage node sn6 ==
Verify: Storage Node [sn6] on jersey1:6200
Zone: [name=JerseyCity id=zn2 type=SECONDARY allowArbiters=false
masterAffinity=false]
Ver: 21.3.10 2021-12-21 21:24:59 UTC  Build id: 78bbc4cb976b Edition:
Enterprise isMasterBalanced: true   serviceStartTime: 2022-06-09 07:05:44
UTC
Verify:  Admin [admin6]
Status: RUNNING,MASTER (non-authoritative)
Verify:  Rep Node [rg1-rn6]
Status: RUNNING,MASTER (non-authoritative) sequenceNumber:217 haPort:6003
available storage size:12 GB
Verification complete, 9 violations, 0 notes found.
Verification violation: [admin1]        ping() failed for admin1 : Unable
to connect to the storage node agent at host nyc1, port 5000 , which may
not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host: nyc1;
nested exception is:
        java.net.ConnectException: Connection refused (Connection refused)
```

```
Verification violation: [admin2]        ping() failed for admin2 :
Unable to connect to the storage node agent  at host nyc1, port
5100, which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host:
nyc1; nested exception is:
        java.net.ConnectException: Connection refused (Connection
refused)
Verification violation: [admin3]        ping() failed for admin3 :
Unable to connect to the storage node  at host nyc1, port 5200,
which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host:
nyc1; nested exception is:
        java.net.ConnectException: Connection refused (Connection
refused)
Verification violation: [rg1-rn1]       ping() failed for rg1-rn1 :
Unable to connect to the storage node agent at host nyc1, port
5000, which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host:
nyc1; nested exception is:
        java.net.ConnectException: Connection refused (Connection
refused)
Verification violation: [rg1-rn2]       ping() failed for rg1-rn2 :
Unable to connect to the storage node agent at host nyc1, port
5100, which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host:
nyc1; nested exception is:
        java.net.ConnectException: Connection refused (Connection
refused)
Verification violation: [rg1-rn3]       ping() failed for rg1-rn3 :
Unable to connect to the storage node agent at host nyc1, port
5200, which may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host:
nyc1; nested exception is:
        java.net.ConnectException: Connection refused (Connection
refused)
Verification violation: [sn1]   ping() failed for sn1 : Unable to
connect to the storage node agent at host nyc1, port 5000, which
may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host:nyc1;
nested exception is:
        java.net.ConnectException: Connection refused (Connection
refused)
Verification violation: [sn2]   ping() failed for sn2 : Unable to
connect to the storage node agent at host nyc1, port 5100, which
may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host:
nyc1; nested exception is:
        java.net.ConnectException: Connection refused (Connection
refused)
Verification violation: [sn3]   ping() failed for sn3 : Unable to
connect to the storage node agent at host nyc1, port 5200, which
may not be running; nested exception is:
        java.rmi.ConnectException: Connection refused to host:nyc1;
nested exception is:
```

```
        java.net.ConnectException: Connection refused (Connection refused)
```

In this case, the Storage Node Agent at host nyc1 is confirmed unavailable.

3. To prevent a hard rollback and data loss, isolate failed nodes (Manhattan) from the rest of the system. Make sure all failed nodes are prevented from rejoining the store until their configurations have been updated.

   To do this, you can:

   - Disconnect the network physically or use a firewall.
   - Modify the start-up sequence on failed nodes to prevent SNAs from starting.

4. To make changes to the store, you first need to reduce admin quorum. To do this, use the `repair-admin-quorum` command, specifying the available primary zone:

```
kv-> repair-admin-quorum -znname JerseyCity
Connected to admin in read-only mode
Repaired admin quorum using admins: [admin4, admin5, admin6]
```

   Now you can perform administrative procedures using the remaining admin service with the temporarily reduced quorum.

5. Use the `plan failover` command to update the configuration of the store with the available zones.

```
kv-> plan failover -znname \
JerseyCity -type primary \
-znname Manhattan -type offline-secondary -wait
Executing plan 8, waiting for completion...
Plan 8 ended successfully
```

   The `plan failover` command fails if it is executed while other plans are still running. You should cancel or interrupt the plans, before executing this plan.

   For example, suppose the `topology redistribute` is in progress. If you run the `plan failover` command, it will fail. For it to succeed, you need to first cancel or interrupt the `topology redistribute` command.

   To do this, first use the `show plans` command to learn the plan ID of the topology redistribute command. In this case, 9. Then, cancel the `topology redistribute` command using the `plan cancel` command:

```
kv-> plan cancel -id 9
```

   After performing the failover, confirm that the zone type of Manhattan has been changed to secondary using the `ping` command.

```
kv-> ping
Pinging components of store mystore based upon topology sequence #208
100 partitions and 6 storage nodes
Time: 2022-06-09 07:33:51 UTC   Version: 21.3.10
Shard Status: healthy:0 writable-degraded:1 read-only:0 offline:0
Admin Status: writable-degraded
Zone [name=Manhattan id=zn1 type=SECONDARY allowArbiters=false
```

```
masterAffinity=false]
RN Status: online:0 offline:3
Zone [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online:3 offline:0
Storage Node [sn1] on nyc1:5000
Zone: [name=Manhattan id=zn1 type=SECONDARY allowArbiters=false
masterAffinity=false]
UNREACHABLE
      Admin [admin1]          Status: UNREACHABLE
      Rep Node [rg1-rn1]      Status: UNREACHABLE
Storage Node [sn2] on nyc1:5100
Zone: [name=Manhattan id=zn1 type=SECONDARY allowArbiters=false
masterAffinity=false]
UNREACHABLE
      Admin [admin2]          Status: UNREACHABLE
      Rep Node [rg1-rn2]      Status: UNREACHABLE
Storage Node [sn3] on nyc1:5200
Zone: [name=Manhattan id=zn1 type=SECONDARY allowArbiters=false
masterAffinity=false]
UNREACHABLE
      Admin [admin3]          Status: UNREACHABLE
      Rep Node [rg1-rn3]      Status: UNREACHABLE
Storage Node [sn4] on jersey1:6000
Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING
Ver: 21.3.10 2021-12-21 21:24:59 UTC  Build id: 78bbc4cb976b
      Admin [admin4]          Status: RUNNING,REPLICA
      Rep Node [rg1-rn4]
Status: RUNNING,REPLICA sequenceNumber:427 haPort:6011 available
storage
Storage Node [sn5] on jersey1:6100
Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING
Ver: 21.3.10 2021-12-21 21:24:59 UTC  Build id: 78bbc4cb976b
      Admin [admin5]          Status: RUNNING,REPLICA
      Rep Node [rg1-rn5]
Status: RUNNING,REPLICA sequenceNumber:427 haPort:6011 available
storage
Storage Node [sn6] on jersey1:6200
Zone: [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING
Ver: 21.3.10 2021-12-21 21:24:59 UTC  Build id: 78bbc4cb976b
     Admin [admin6]          Status: RUNNING,MASTER
     Rep Node [rg1-rn6]
Status: RUNNING,MASTER sequenceNumber:427 haPort:6011 available
storage
```

The failover operation is now complete. Write availability in the store is reestablished using zone 2 as the only available primary zone. Zone 1 is offline. Any data that was not propagated from zone 1 prior to the failure will be lost.

> **Note:**
>
> In this case, the store has only a single working copy of its data, so single node failures in the surviving zone will prevent read and write access, and, if the failure is a permanent one, may produce permanent data loss.

If the problems that led to the failover have been corrected and the original data from the previously failed nodes (Manhattan) is still available, you can return the old nodes to service by performing a switchover. To do this, see the next section.

Performing a Switchover

To continue from the example of the previous section, after performing the failover, you can return the old nodes to service by performing the following switchover procedure:

1. After the failed zones are repaired, restart all the Storage Nodes of the failed zones without starting any services (avoids hard rollback):

   ```
   java -Xmx64m -Xms64m \
   -jar KVHOME/lib/kvstore.jar restart -disable-services \
   -root nyc1/KVROOT &
   ```

   > **Note:**
   >
   > When performing planned maintenance, there is no need to isolate nodes or disable services prior to bringing nodes back online.

2. Reestablish network connectivity or reenable the standard startup sequence of the previously failed zones.
3. Repair the topology so that the topology for the newly restarted Storage Nodes can be updated with changes made by the failover.

   ```
   java -Xmx64m -Xms64m -jar KVHOME/lib/kvstore.jar runadmin \
   -host jersey1 -port 5000 \
   -security USER/security/admin.security

   kv-> plan repair-topology -wait
   Executed plan 10, waiting for completion...
   Plan 10 ended successfully
   ```

   > **Note:**
   >
   > This assumes that you must have followed the steps as mentioned in Create users and configure security with remote access .

> **Note:**
>
> This command will also restart services on the previously failed nodes.

Use the `verify configuration` command to confirm that there are no configuration problems.

4. Run the ping command. The "maxCatchupTimeSecs" value will be used for the -timeout flag of the `await-consistency` command.

   Use the timeout flag to specify an estimate of how long the switchover will take. For example, if the nodes have been offline for a long time it might take many hours for them to catch up so that they can be converted back to primary nodes.

```
kv-> ping
Pinging components of store mystore based upon topology sequence
#117
100 partitions and 6 storage nodes
Time: 2022-06-09 07:39:18 UTC   Version: 21.3.10
Shard Status: healthy: 1 writable-degraded: 0 read-only: 0 offline:
0 total: 1
Admin Status: healthy
Zone [name=Manhattan id=zn1 type=SECONDARY allowArbiters=false
masterAffinity=false]
RN Status: online: 3 read-only: 0 offline: 0 maxDelayMillis: 3
maxCatchupTimeSecs: 0
Zone [name=JerseyCity id=zn2 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online: 3 read-only: 0 offline: 0 maxDelayMillis: 4
maxCatchupTimeSecs: 0
Storage Node [sn1] on nyc1: 5000    Zone: name=Manhattan id=zn1
type=SECONDARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise isMasterBalanced: true
serviceStartTime: 2022-06-09 07:36:01 UTC
Admin [admin1]  Status: RUNNING,REPLICA serviceStartTime:
2022-06-09 07:38:14 UTC
stateChangeTime: 2022-06-09 07:38:14 UTC  availableStorageSize: 2 GB
Rep Node [rg1-rn1] Status: RUNNING,REPLICA sequenceNumber: 2,672
haPort: 5111
availableStorageSize: 273 GB storageType: HD serviceStartTime:
2022-06-09 07:37:14 UTC
stateChangeTime: 2022-06-09 07:37:20 UTC delayMillis: 0
catchupTimeSecs: 0
Storage Node [sn2] on nyc1: 5100    Zone: [name=Manhattan id=zn1
type=SECONDARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise  isMasterBalanced:
true
serviceStartTime: 2022-06-09 07:36:25 UTC
Admin [admin2]  Status: RUNNING,REPLICA serviceStartTime:
```

```
2022-06-09 07:38:34 UTC
stateChangeTime: 2022-06-09 07:38:33 UTC availableStorageSize: 2 GB
Rep Node [rg1-rn2]  Status: RUNNING,REPLICA sequenceNumber: 2,672 haPort:
5211
availableStorageSize: 273 GB storageType: HD serviceStartTime: 2022-06-09
07:37:28 UTC
stateChangeTime: 2022-06-09 07:37:33 UTC delayMillis: 0 catchupTimeSecs: 0
Storage Node [sn3] on nyc1: 5200    Zone: [name=Manhattan id=zn1
type=SECONDARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise isMasterBalanced: true
serviceStartTime: 2022-06-09 07:36:35 UTC
Admin [admin3]  Status: RUNNING,REPLICA serviceStartTime: 2022-06-09
07:38:56 UTC
stateChangeTime: 2022-06-09 07:38:56 UTC  availableStorageSize: 2 GB
Rep Node [rg1-rn3] Status: RUNNING,REPLICA sequenceNumber: 2,672 haPort:
5311
availableStorageSize: 273 GB storageType: HD serviceStartTime: 2022-06-09
07:37:43 UTC
stateChangeTime: 2022-06-09 07:37:49 UTC delayMillis: 3 catchupTimeSecs: 0
Storage Node [sn4] on jersey1: 6000  Zone: [name=JerseyCity id=zn2
type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise isMasterBalanced: true
serviceStartTime: 2022-06-09 07:05:44 UTC
Admin [admin4]  Status: RUNNING,REPLICA serviceStartTime: 2022-06-09
07:36:49 UTC
stateChangeTime: 2022-06-09 07:36:47 UTC availableStorageSize: 2 GB
Rep Node [rg1-rn4]  Status: RUNNING,REPLICA sequenceNumber: 2,672 haPort:
5411
availableStorageSize: 273 GB storageType: HD serviceStartTime: 2022-06-09
07:36:36 UTC
stateChangeTime: 2022-06-09 07:36:59 UTC delayMillis: 4 catchupTimeSecs: 0
Storage Node [sn5] on jersey1: 6100    Zone: [name=JerseyCity id=zn2
type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise  isMasterBalanced: true
serviceStartTime: 2022-06-09 07:05:54 UTC
Admin [admin5] Status: RUNNING,REPLICA serviceStartTime: 2022-06-09
07:36:49 UTC
stateChangeTime: 2022-06-09 07:36:48 UTC  availableStorageSize: 2 GB
Rep Node [rg1-rn5]  Status: RUNNING,REPLICA sequenceNumber: 2,672 haPort:
5511
availableStorageSize: 273 GB storageType: HD serviceStartTime: 2022-06-09
07:36:36 UTC
stateChangeTime: 2022-06-09 07:36:59 UTC delayMillis: 0 catchupTimeSecs: 0
Storage Node [sn6] on jersey1: 6200    Zone: [name=JerseyCity id=zn2
type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise  isMasterBalanced: true
serviceStartTime: 2022-06-09 07:06:03 UTC
```

```
Admin [admin6]  Status: RUNNING,MASTER  serviceStartTime:
2022-06-09 07:36:55 UTC
stateChangeTime: 2022-06-09 07:36:46 UTC  availableStorageSize: 2 GB
Rep Node [rg1-rn6] Status: RUNNING,MASTER sequenceNumber: 2,672
haPort: 5611
availableStorageSize: 273 GB storageType: HD  serviceStartTime:
2022-06-09 07:36:36 UTC
stateChangeTime: 2022-06-09 07:36:57 UTC
```

In this case, 1800 seconds (30 minutes) is the value to be used.

5. Use the `await-consistency` command to specify the wait time (1800 seconds) used for the secondary zones to catch up with their masters.

   The system will only wait five minutes for nodes to catch up when attempting to change a zone's type. If the nodes do not catch up in that amount of time, the plan will fail.

   If the nodes will take more than five minutes to catch up, you should run the `await-consistency` command, specifying a longer wait time using the -timeout flag. In this case, the wait time (1800 seconds) is used:

   ```
   kv-> await-consistent -timeout 1800 -znname Manhattan
   The specified zone is consistent
   ```

   By default, nodes need to have a delay of no more than 1 second to be considered caught up. You can change this value by specifying the -replica-delay-threshold flag. You should do this if network delays prevent the nodes from catching up within 1 second of their masters.

   > **Note:**
   >
   > If you do not want the switchover to wait for the nodes to catch up, you can use the -no-replica-delay threshold flag. In that case, nodes will be converted to primary nodes even if they are behind. You should evaluate whether this risk is worth taking.

6. Perform the switchover to convert the previously failed zone back to a primary zone , and the formerly secondary zone back to its earlier state.

   ```
   kv-> topology clone -current -name newTopo
   kv-> topology change-zone-type -name newTopo \
   -znname Manhattan -type primary
   Changed zone type of zn1 to PRIMARY in newTopo

   kv-> topology change-zone-type -name newTopo \
   -znname JerseyCity -type secondary
   Changed zone type of zn2 to SECONDARY in newTop

   kv-> plan deploy-topology -name newTopo -wait
   Executed plan 11, waiting for completion...
   Plan 11 ended successfully
   ```

Confirm the zone type change of the Manhattan zone to PRIMARY by running the ping command.

```
kv-> ping
Pinging components of store mystore based upon topology sequence #117
100 partitions and 6 storage nodes
Time: 2022-06-09 07:39:18 UTC   Version: 21.3.10
Shard Status: healthy: 1 writable-degraded: 0 read-only: 0 offline: 0
total: 1
Admin Status: healthy
Zone [name=Manhattan id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online: 3 read-only: 0 offline: 0 maxDelayMillis: 3
maxCatchupTimeSecs: 0
Zone [name=JerseyCity id=zn2 type=SECONDARY allowArbiters=false
masterAffinity=false]
RN Status: online: 3 read-only: 0 offline: 0 maxDelayMillis: 4
maxCatchupTimeSecs: 0
Storage Node [sn1] on nyc1: 5000    Zone: name=Manhattan id=zn1
type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise isMasterBalanced: true
serviceStartTime: 2022-06-09 07:36:01 UTC
Admin [admin1]  Status: RUNNING,MASTER serviceStartTime: 2022-06-09
07:38:14 UTC
stateChangeTime: 2022-06-09 07:38:14 UTC  availableStorageSize: 2 GB
Rep Node [rg1-rn1]  Status: RUNNING,MASTER sequenceNumber: 2,672 haPort:
5111
availableStorageSize: 273 GB storageType: HD serviceStartTime: 2022-06-09
07:37:14 UTC
stateChangeTime: 2022-06-09 07:37:20 UTC delayMillis: 0 catchupTimeSecs: 0
Storage Node [sn2] on nyc1: 5100  Zone: [name=Manhattan id=zn1
type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING  Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise  isMasterBalanced: true
serviceStartTime: 2022-06-09 07:36:25 UTC
Admin [admin2]   Status: RUNNING,REPLICA serviceStartTime: 2022-06-09
07:38:34 UTC
stateChangeTime: 2022-06-09 07:38:33 UTC availableStorageSize: 2 GB
Rep Node [rg1-rn2]  Status: RUNNING,REPLICA sequenceNumber: 2,672 haPort:
5211
availableStorageSize: 273 GB storageType: HD serviceStartTime: 2022-06-09
07:37:28 UTC
stateChangeTime: 2022-06-09 07:37:33 UTC delayMillis: 0 catchupTimeSecs: 0
Storage Node [sn3] on nyc1: 5200    Zone: [name=Manhattan id=zn1
type=PRIMARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise    isMasterBalanced: true
 serviceStartTime: 2022-06-09 07:36:35 UTC
Admin [admin3]        Status: RUNNING,REPLICA serviceStartTime:
2022-06-09 07:38:56 UTC
stateChangeTime: 2022-06-09 07:38:56 UTC  availableStorageSize: 2 GB
```

```
Rep Node [rg1-rn3     Status: RUNNING,REPLICA sequenceNumber: 2,672
haPort: 5311
availableStorageSize: 273 GB storageType: HD serviceStartTime:
2022-06-09 07:37:43 UTC
stateChangeTime: 2022-06-09 07:37:49 UTC delayMillis: 3
catchupTimeSecs: 0
Storage Node [sn4] on jersey1: 6000    Zone: [name=JerseyCity
id=zn2 type=SECONDARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise isMasterBalanced:
true
serviceStartTime: 2022-06-09 07:05:44 UTC
Admin [admin4]  Status: RUNNING,REPLICA serviceStartTime:
2022-06-09 07:36:49 UTC
stateChangeTime: 2022-06-09 07:36:47 UTC availableStorageSize: 2 GB
Rep Node [rg1-rn4]  Status: RUNNING,REPLICA sequenceNumber: 2,672
haPort: 5411
availableStorageSize: 273 GB storageType: HD serviceStartTime:
2022-06-09 07:36:36 UTC
stateChangeTime: 2022-06-09 07:36:59 UTC delayMillis: 4
catchupTimeSecs: 0
Storage Node [sn5] on jersey1: 6100 Zone: [name=JerseyCity id=zn2
type=SECONDARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise isMasterBalanced:
true
serviceStartTime: 2022-06-09 07:05:54 UTC
Admin [admin5]    Status: RUNNING,REPLICA serviceStartTime:
2022-06-09 07:36:49 UTC
stateChangeTime: 2022-06-09 07:36:48 UTC  availableStorageSize: 2 GB
Rep Node [rg1-rn5]   Status: RUNNING,REPLICA sequenceNumber: 2,672
haPort: 5511
availableStorageSize: 273 GB storageType: HD serviceStartTime:
2022-06-09 07:36:36 UTC
stateChangeTime: 2022-06-09 07:36:59 UTC delayMillis: 0
catchupTimeSecs: 0
Storage Node [sn6] on jersey1: 6200 Zone: [name=JerseyCity id=zn2
type=SECONDARY
allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 21.3.10 2021-12-21 21:24:59 UTC
Build id: 78bbc4cb976b Edition: Enterprise  isMasterBalanced:
true
serviceStartTime: 2022-06-09 07:06:03 UTC
Admin [admin6]   Status: RUNNING,REPLICA  serviceStartTime:
2022-06-09 07:36:55 UTC
stateChangeTime: 2022-06-09 07:36:46 UTC  availableStorageSize: 2 GB
Rep Node [rg1-rn6]  Status: RUNNING,REPLICA sequenceNumber: 2,672
haPort: 5611
availableStorageSize: 273 GB storageType: HD  serviceStartTime:
2022-06-09 07:36:36 UTC
stateChangeTime: 2022-06-09 07:36:57 UTC
```

Zone Failover

Zones allow you to spread your data store across various physical installation locations. The different locations can be anything from different physical buildings near each other, to different racks in the same building. The basic goal of spreading your store across locations is to guard against large-scale infrastructure disruptions, such as power outages or major storm damage, by placing the nodes in your store physically as far apart as possible.

Oracle NoSQL Database provides support for two kinds of zones. *Primary* zones contain nodes which can serve as masters or replicas. Zones are created as primary zones by default. *Secondary* zones contain nodes which can serve only as replicas. Secondary zones can be used to make a copy of the data available at a distant location, or to maintain an extra copy of the data to increase redundancy or read capacity.

Both types of zones require high throughput network connections to transmit the replication data required to keep replicas up-to-date. Failing to provide sufficient network capacity will result in nodes in poorly connected zones falling farther and farther behind. Locations connected by low throughput network connections are not suitable for use with zones.

For primary zones, in addition to a high throughput network, the network connections with other primary zones should provide highly reliable and low latency communication. These capabilities make it possible to perform master elections for quick master failovers, and to provide acknowledgments to meet write request timeout requirements. Primary zones are not, therefore, suitable for use with an unreliable or slow wide area network.

For secondary zones, the nodes do not participate in master elections or acknowledgments. For this reason, the system can tolerate reduced reliability or increased latency for connections between secondary and primary zones. The network connections still need to provide sufficient throughput to support replication, and must provide sufficient reliability that temporary interruptions do not interfere with network throughput.

If you deploy your store across multiple zones, then Oracle NoSQL Database tries to physically place at least one Replication Node from each shard in each zone. Whether Oracle NoSQL Database can do this depends on the number of shards in use in your store, the number of zones, the number of Replication Nodes, and the number of physical machines available in each zone. Still, Oracle NoSQL Database makes a best-effort to spread Replication Nodes across available zones. Doing so guards against losing entire shards should the zone become unavailable for any reason.

All of the failover descriptions covered here apply to zones. Failover works across zones in the same way as it does if all nodes are contained within a single zone. Zones offer you the ability for your data to remain available in the event of a large outage. However, read and write capability for any given shard is still gated by whether the remaining zone(s) constitute a majority node partition, and the durability and consistency policies in use for your store activities.

Durability Summary

This document has described how durability guarantees affect a shard's write availability in the event of hardware or network failures. In summary:

- A durability guarantee that requires no acknowledgements from the shard's replicas gives you the best chance that the shard can continue servicing write requests in the event of an outage. However, this durability guarantee can also result in the shard operating with

two masters, which leads to data loss once hardware problems are resolved. This is *not* a recommended configuration.

- A durability guarantee requiring a simple majority of primary zone replicas to acknowledge the write operation guards against two masters accidently operating at one time. However, it also means that the shard will be incapable of servicing write requests if more than a majority of the replicas are offline due to a hardware failure.
- A durability guarantee requiring all primary zone replicas to acknowledge the write operation guards against any possibility of data loss. However, it also means that the shard will be unable to service write requests if even one of the replicas is unavailable for any reason.

Consistency Summary

In most cases, replicas can continue to service read requests as long as the underlying hardware remains functional. In its default configuration, there is nothing that stops a replica from doing this, even if it is the only node running after some catastrophic failure.

However, is is possible for a replica to stop servicing read requests following a network failure, if the consistency policy requires either version information, or disallows stale data relative to the master. Whether this happens depends on how your Replication Nodes are exactly partitioned as a result of the failure, and how long it takes to establish a new master. The replica's ability to service read requests is also determined by the consistency policy in use for each request. If the read requires tight consistency with the master, and the master is not available to ensure the consistency can be met, then the read will fail.

5
Tools and Utilities

The articles in this section describe about the plugins and extensions available for developing NoSQL applications in the Oracle NoSQL Database from external integrated development environments (IDEs) or code editors.

Using Oracle NoSQL Database Migrator

Learn about Oracle NoSQL Database Migrator and how to use it for data migration.

Oracle NoSQL Database Migrator is a tool that enables you to migrate Oracle NoSQL tables from one data source to another. This tool can operate on tables in Oracle NoSQL Database Cloud Service and Oracle NoSQL Database on-premises and AWS S3. The Migrator tool supports several different data formats and physical media types. Supported data formats are JSON, Parquet, MongoDB-formatted JSON, DynamoDB-formatted JSON, and CSV files. Supported physical media types are files, OCI Object Storage, Oracle NoSQL Database on-premises, Oracle NoSQL Database Cloud Service, and AWS S3.

Topics:

- Overview
- Workflow for Oracle NoSQL Database Migrator
- Supported Sources and Sinks
- Use Case Demonstrations
- Troubleshooting the Oracle NoSQL Database Migrator

Overview

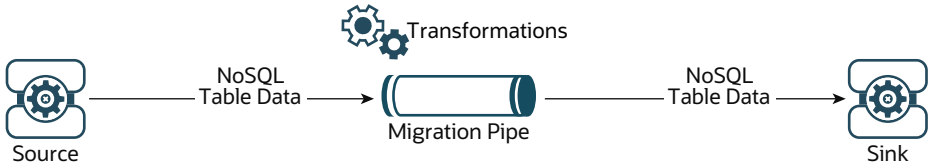
Oracle NoSQL Database Migrator lets you move Oracle NoSQL tables from one data source to another, such as Oracle NoSQL Database on-premises or cloud or even a simple JSON file.

There can be many situations that require you to migrate NoSQL tables *from* or *to* an Oracle NoSQL Database. For instance, a team of developers enhancing a NoSQL Database application may want to test their updated code in the local Oracle NoSQL Database Cloud Service (NDCS) instance using cloudsim. To verify all the possible test cases, they must set up the test data similar to the actual data. To do this, they must copy the NoSQL tables from the production environment to their local NDCS instance, the cloudsim environment. In another situation, NoSQL developers may need to move their application data from on-premise to the cloud and vice-versa, either for development or testing.

In all such cases and many more, you can use Oracle NoSQL Database Migrator to move your NoSQL tables from one data source to another, such as Oracle NoSQL Database on-premise or cloud or even a simple JSON file. You can also copy NoSQL tables from a MongoDB-formatted JSON input file, DynamoDB-formatted JSON input file (either stored in AWS S3 source or from files), or a CSV file into your NoSQL Database on-premises or cloud.

As depicted in the following figure, the NoSQL Database Migrator utility acts as a connector or pipe between the data source and the target (referred to as the sink). In essence, this utility exports data from the selected source and imports that data into the sink. This tool is table-oriented, that is, you can move the data only at the table level. A single migration task operates on a single table and supports migration of table data from source to sink in various data formats.

Oracle NoSQL Database Migrator is designed such that it can support additional sources and sinks in the future. For a list of sources and sinks supported by Oracle NoSQL Database Migrator as of the current release, see Supported Sources and Sinks.



Terminology used with Oracle NoSQL Database Migrator

Learn about the different terms used in the above diagram, in detail.

- **Source:** An entity from where the NoSQL tables are exported for migration. Some examples of sources are Oracle NoSQL Database on-premise or cloud, JSON file, MongoDB-formatted JSON file, DynamoDB-formatted JSON file, and CSV files.
- **Sink:** An entity that imports the NoSQL tables from NoSQL Database Migrator. Some examples for sinks are Oracle NoSQL Database on-premise or cloud and JSON file.

The NoSQL Database Migrator tool supports different types of sources and sinks (that is physical media or repositories of data) and data formats (that is how the data is represented in the source or sink). Supported data formats are JSON, Parquet, MongoDB-formatted JSON, DynamoDB-formatted JSON, and CSV files. Supported source and sink types are files, OCI Object Storage, Oracle NoSQL Database on-premise, and Oracle NoSQL Database Cloud Service.

- **Migration Pipe:** The data from a source will be transferred to the sink by NoSQL Database Migrator. This can be visualized as a Migration Pipe.
- **Transformations:** You can add rules to modify the NoSQL table data in the migration pipe. These rules are called Transformations. Oracle NoSQL Database Migrator allows data transformations at the top-level fields or columns only. It does not let you transform the data in the nested fields. Some examples of permitted transformations are:
  - Drop or ignore one or more columns,
  - Rename one or more columns, or
  - Aggregate several columns into a single field, typically a JSON field.
- **Configuration File :** A configuration file is where you define all the parameters required for the migration activity in a JSON format. Later, you pass this

configuration file as a single parameter to the `runMigrator` command from the CLI. A typical configuration file format looks like as shown below.

```
{
 "source": {
   "type" : <source type>,
   //source-configuration for type. See  Source Configuration Templates .
 },
 "sink": {
   "type" : <sink type>,
   //sink-configuration for type. See  Sink Configuration Templates .
 },
 "transforms" : {
   //transforms configuration. See  Transformation Configuration
Templates .
 },
 "migratorVersion" : "<migrator version>",
 "abortOnError" : <true|false>
}
```

Group	Parameters	Mandatory (Y/N)	Purpose	Supported Values
`source`	`type`	Y	Represents the source from which to migrate the data. The source provides data and metadata (if any) for migration.	To know the `type` value for each source, see Supported Sources and Sinks.
`source`	source-configuration for type	Y	Defines the configuration for the source. These configuration parameters are specific to the type of source selected above.	See Source Configuration Templates . for the complete list of configuration parameters for each source type.
`sink`	`type`	Y	Represents the sink to which to migrate the data. The sink is the target or destination for the migration.	To know the `type` value for each source, see Supported Sources and Sinks.
`sink`	sink-configuration for type	Y	Defines the configuration for the sink. These configuration parameters are specific to the type of sink selected above.	See Sink Configuration Templates for the complete list of configuration parameters for each sink type.

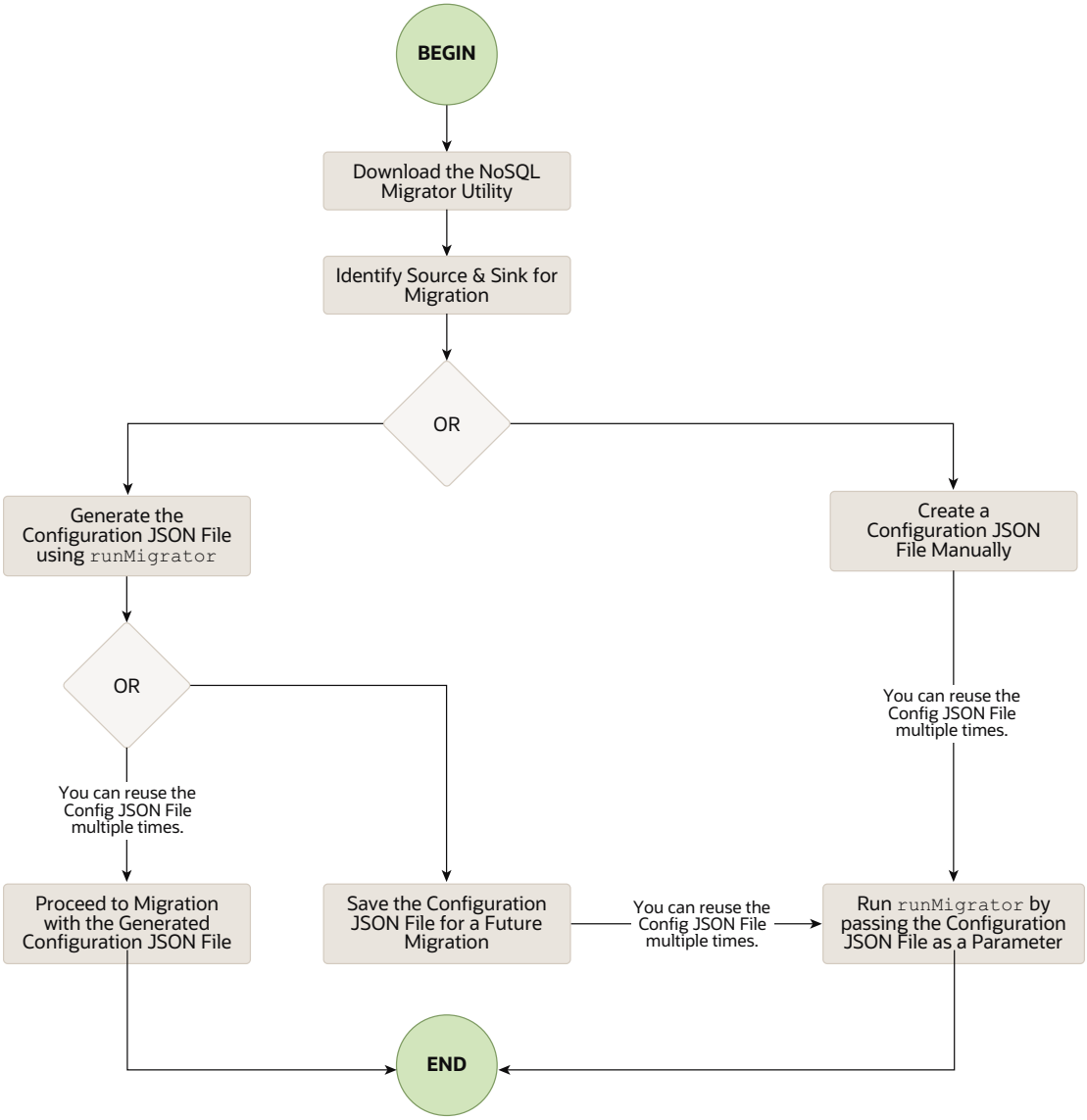
Group	Parameters	Mandatory (Y/N)	Purpose	Supported Values
`transforms`	transforms configuration	N	Defines the transformations to be applied to the data in the migration pipe.	See Transformation Configuration Templates for the complete list of transformations supported by the NoSQL Data Migrator.
-	`migratorVersion`	N	Version of the NoSQL Data Migrator	-
-	`abortOnError`	N	Specifies whether to stop the migration activity in case of any error or not. The default value is *true* indicating that the migration stops whenever it encounters a migration error. If you set this value to *false*, the migration continues even in case of failed records or other migration errors. The failed records and migration errors will be logged as WARNINGs on the CLI terminal.	true, false

> **Note:**
>
> As JSON file is case-sensitive, all the parameters defined in the configuration file are case-sensitive unless specified otherwise.

Workflow for Oracle NoSQL Database Migrator

Learn about the various steps involved in using the Oracle NoSQL Database Migrator utility for migrating your NoSQL data.

The high level flow of tasks involved in using NoSQL Database Migrator is depicted in the below figure.



Download the NoSQL Data Migrator Utility

The Oracle NoSQL Database Migrator utility is available for download from the Oracle NoSQL Downloads page. Once you download and unzip it on your machine, you can access the `runMigrator` command from the command line interface.

> **Note:**
>
> Oracle NoSQL Database Migrator utility requires Java 11 or higher versions to run.

Identify the Source and Sink

Before using the migrator, you must identify the data source and sink. For instance, if you want to migrate a NoSQL table from Oracle NoSQL Database on-premise to a JSON formatted file, your source will be Oracle NoSQL Database and sink will be JSON file. Ensure that the identified source and sink are supported by the Oracle NoSQL Database Migrator by

referring to Supported Sources and Sinks. This is also an appropriate phase to decide the schema for your NoSQL table in the target or sink, and create them.

- **Identify Sink Table Schema:** If the sink is Oracle NoSQL Database on-premise or cloud, you must identify the schema for the sink table and ensure that the source data matches with the target schema. If required, use transformations to map the source data to the sink table.
  - **Default Schema:** NoSQL Database Migrator provides an option to create a table with the default schema without the need to predefine the schema for the table. This is useful primarily when loading JSON source files into Oracle NoSQL Database.
    If the source is a MongoDB-formatted JSON file, the default schema for the table will be as follows:

    ```
    CREATE TABLE IF NOT EXISTS <tablename>(ID STRING, DOCUMENT
    JSON,PRIMARY KEY(SHARD(ID))
    ```

    Where:

    — tablename = value provided for the table attribute in the configuration.

    — ID = _id value from each document of the mongoDB exported JSON source file.

    — DOCUMENT = For each document in the mongoDB exported file, the contents excluding the _id field are aggregated into the DOCUMENT column.

    If the source is a DynamoDB-formatted JSON file, the default schema for the table will be as follows:

    ```
    CREATE TABLE IF NOT EXISTS <TABLE_NAME>(DDBPartitionKey_name
    DDBPartitionKey_type,
    [DDBSortKey_name DDBSortKey_type],DOCUMENT JSON,
    PRIMARY KEY(SHARD(DDBPartitionKey_name),[DDBSortKey_name]))
    ```

    Where:

    — TABLE_NAME = value provided for the sink table in the configuration

    — DDBPartitionKey_name = value provided for the partition key in the configuration

    — DDBPartitionKey_type = value provided for the data type of the partition key in the configuration

    — DDBSortKey_name = value provided for the sort key in the configuration if any

    — DDBSortKey_type = value provided for the data type of the sort key in the configuration if any

    — DOCUMENT = All attributes except the partition and sort key of a Dynamo DB table item aggregated into a NoSQL JSON column

    If the source format is a CSV file, a default schema is not supported for the target table. You can create a schema file with a table definition containing the same number of columns and data types as the source CSV file. For more details on the Schema file creation, see Providing Table Schema.

For all the other sources, the default schema will be as follows:

```
CREATE TABLE IF NOT EXISTS <tablename> (ID LONG GENERATED ALWAYS AS
IDENTITY, DOCUMENT JSON, PRIMARY KEY(ID))
```

Where:

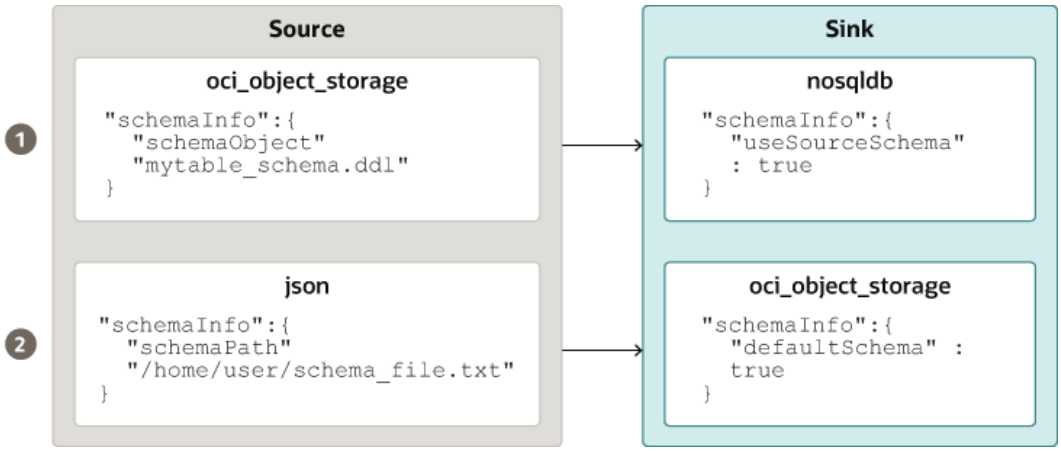
— tablename = value provided for the table attribute in the configuration.

— ID = An auto-generated LONG value.

— DOCUMENT = The JSON record provided by the source is aggregated into the DOCUMENT column.

> **Note:**
>
> If the _id value is not provided as a string in the MongoDB-formatted JSON file, NoSQL Database Migrator converts it into a string before inserting it into the default schema.

- **Providing Table Schema:** NoSQL Database Migrator allows the source to provide schema definitions for the table data using `schemaInfo` attribute. The `schemaInfo` attribute is available in all the data sources that do not have an implicit schema already defined. Sink data stores can choose any one of the following options.
  - Use the default schema defined by the NoSQL Database Migrator.
  - Use the source-provided schema.
  - Override the source-provided schema by defining its own schema. For example, if you want to transform the data from the source schema to another schema, you need to override the source-provided schema and use the transformation capability of the NoSQL Database Migrator tool.



The table schema file, for example, `mytable_schema.ddl` can include table DDL statements. The NoSQL Database Migrator tool executes this table schema file before

starting the migration. The migrator tool supports no more than one DDL statement per line in the schema file. For example,

```
CREATE TABLE IF NOT EXISTS(id INTEGER, name STRING, age INTEGER,
PRIMARY KEY(SHARD(ID)))
```

> **Note:**
>
> Migration will fail if the table is present at the sink and the DDL in the `schemaPath` is different than the table.

- **Create Sink Table:** Once you identify the sink table schema, create the sink table either through the Admin CLI or using the `schemaInfo` attribute of the sink configuration file. See Sink Configuration Templates .

> **Note:**
>
> If the source is a CSV file, create a file with the DDL commands for the schema of the target table. Provide the file path in `schemaInfo.schemaPath` parameter of the sink configuration file.

Migrating TTL Metadata for Table Rows

You can choose to include the TTL metadata for table rows along with the actual data when performing migration of NoSQL tables. The NoSQL Database Migrator provides a configuration parameter to support the export and import of table row TTL metadata. Additionally, the tool provides an option to select the relative expiry time for table rows during the import operation. You can optionally export or import TTL metadata using the `includeTTL` parameter.

> **Note:**
>
> The support for migrating TTL metadata for table rows is *only* available for Oracle NoSQL Database and Oracle NoSQL Database Cloud Service.

Exporting TTL metadata

When a table is exported, TTL data is exported for the table rows that have a valid expiration time. If a row does not expire, then it is not included explicitly in the exported data because its expiration value is always 0. TTL information is contained in the `_metadata` JSON object for each exported row. The NoSQL Database Migrator exports the expiration time for each row as the number of milliseconds since the UNIX epoch (Jan 1st, 1970). For example,

```
//Row 1
{
    "id" : 1,
    "name" : "xyz",
    "age" : 45,
```

```
    "_metadata" : {
        "expiration" : 1629709200000   //Row Expiration time in milliseconds
    }
}

//Row 2
{
    "id" : 2,
    "name" : "abc",
    "age" : 52,
    "_metadata" : {
        "expiration" : 1629709400000   //Row Expiration time in milliseconds
    }
}

//Row 3 No Metadata for below row as it will not expire
{
    "id" : 3,
    "name" : "def",
    "age" : 15
}
```

Importing TTL metadata

You can optionally import TTL metadata using a configuration parameter, `includeTTL`. The import operation handles the following use cases when migrating table rows containing TTL metadata. These use-cases are applicable *only* when the `includeTTL` configuration parameter is specified.

In the use-cases 2 and 3, the default Reference Time of import operation is the current time in milliseconds, obtained from System.currentTimeMillis(), of the machine where the NoSQL Database Migrator tool is running. But you can also set a custom Reference Time using the `ttlRelativeDate` configuration parameter if you want to extend the expiration time and import rows that would otherwise expire immediately.

- **Use-case 1**: No TTL metadata information is present in the importing table row.
  When you import a JSON source file produced from an external source or exported using earlier versions of the migrator, the importing row does not have TTL information. But since the `includeTTL` configuration parameter is equal to `true`, the migrator set the TTL=0 for the table row, which means the importing table row never expires.
- **Use-case 2**: TTL value of the source table row is expired relative to the Reference Time when the table row gets imported.
  When you export a table row to a file and try to import it after the expiration time of the table row, the table row is ignored and is not written into the store.
- **Use-case 3**: TTL value of the source table row is not expired relative to the Reference Time when the table row gets imported.
  When you export a table row to a file and try to import it before the expiration time of the table row, the table row gets imported with a TTL value. But the new TTL value for the table row may not be equal to exported TTL value because of the integer hour and day window constraints in the TimeToLive class. For example,

  Exported table row

  ```
  {
    "id" : 8,
  ```

```
  "name" : "xyz",
  "_metadata" : {
    "expiration" : 1629709200000 //Monday, August 23, 2021 9:00:00
AM in UTC
  }
}
```

The reference time while importing is 1629707962582, which is Monday, August 23, 2021 8:39:22.582 AM.

Imported table row

```
{
  "id" : 8,
  "name" : "xyz",
  "_metadata" : {
    "ttl" :  1629712800000 //Monday, August 23, 2021 10:00:00 AM UTC
  }
}
```

Run the `runMigrator` command

The `runMigrator` executable file is available in the extracted NoSQL Database Migrator files. You must install Java 8 or higher version and bash on your system to successfully run the `runMigrator` command.

You can run the `runMigrator` command in two ways:

1. By creating the configuration file using the runtime options of the `runMigrator` command as shown below.

   ```
   [~]$ ./runMigrator
   configuration file is not provided. Do you want to generate
   configuration?
   (y/n)


   [n]: y
   ...
   ...
   ```

   - When you invoke the `runMigrator` utility, it provides a series of runtime options and creates the configuration file based on your choices for each option.
   - After the utility creates the configuration file, you have a choice to either proceed with the migration activity in the same run or save the configuration file for a future migration.
   - Irrespective of your decision to proceed or defer the migration activity with the generated configuration file, the file will be available for edits or customization to meet your future requirements. You can use the customized configuration file for migration later.

2. By passing a manually created configuration file (in the JSON format) as a runtime parameter using the `-c` or `--config` option. You must create the configuration file

manually before running the `runMigrator` command with the `-c` or `--config` option. For any help with the source and sink configuration parameters, see Sources and Sinks.

```
[~]$ ./runMigrator -c </path/to/the/configuration/json/file>
```

Logging Migrator Progress

NoSQL Database Migrator tool provides options, which enables trace, debugging, and progress messages to be printed to standard output or to a file. This option can be useful in tracking the progress of migration operation, particularly for very large tables or data sets.

- **Log Levels**
  To control the logging behavior through the NoSQL Database Migrator tool, pass the --log-level or -l run time parameter to the `runMigrator` command. You can specify the amount of log information to write by passing the appropriate log level value.

  ```
  $./runMigrator --log-level <loglevel>
  ```

  Example:

  ```
  $./runMigrator --log-level debug
  ```

  Table 5-1 Supported Log Levels for NoSQL Database Migrator

Log Level	Description
warning	Prints errors and warnings.
info (**default**)	Prints the progress status of data migration such as validating source, validating sink, creating tables, and count of number of data records migrated.
debug	Prints additional debug information.
all	Prints everything. This level turns on all levels of logging.

- **Log File:**
  You can specify the name of the log file using `--log-file` or `-f` parameter. If `--log-file` is passed as run time parameter to the `runMigrator` command, the NoSQL Database Migrator writes all the log messages to the file else to the standard output.

  ```
  $./runMigrator --log-file <log file name>
  ```

  Example:

  ```
  $./runMigrator --log-file nosql_migrator.log
  ```

Sources and Sinks

Learn about the different sources and sinks supported by the Oracle NoSQL Database Migrator utility and their configuration templates.

Topics:

- Supported Sources and Sinks
- Source and Sink Security
- Source Configuration Templates
- Sink Configuration Templates
- Transformation Configuration Templates
- Mapping of DynamoDB table to Oracle NoSQL table
- Oracle NoSQL to Parquet Data Type Mapping
- Mapping of DynamoDB types to Oracle NoSQL types

Supported Sources and Sinks

This topic provides the list of the sources and sinks supported by the Oracle NoSQL Database Migrator.

You can use any combination of a valid source and sink from this table for the migration activity. However, you must ensure that at least one of the ends, that is, source or sink must be an Oracle NoSQL product. You can not use the NoSQL Database Migrator to move the NoSQL table data from one file to another.

Type (value)	**Format (value)**	**Valid Source**	**Valid Sink**
Oracle NoSQL Database (`nosqldb`)	NA	Y	Y
Oracle NoSQL Database Cloud Service (`nosqldb_cloud`)	NA	Y	Y
File system (`file`)	JSON (`json`)	Y	Y
	MongoDB JSON (`mongodb_json`)	Y	N
	DynamoDB JSON (`dynamodb_json`)	Y	N
	Parquet(`parquet`)	N	Y

Type (value)	Format (value)	Valid Source	Valid Sink
	CSV (`csv`)	Y	N
OCI Object Storage (`object_storage_oci`)	JSON (`json`)	Y	Y
	MongoDB JSON (`mongodb_json`)	Y	N
	Parquet(`parquet`)	N	Y
	CSV (`csv`)	Y	N
AWS S3	DynamoDB JSON (`dynamodb_json`)	Y	N

Note:

Many configuration parameters are common across the source and sink configuration. For ease of reference, the description for such parameters is repeated for each source and sink in the documentation sections, which explain configuration file formats for various types of sources and sinks. In all the cases, the syntax and semantics of the parameters with the same name are identical.

Source and Sink Security

Some of the source and sink types have optional or mandatory security information for authentication purposes.

All sources and sinks that use services in the Oracle Cloud Infrastructure (OCI) can use certain parameters for providing optional security information. This information can be provided using an OCI configuration file or Instance Principal.

Oracle NoSQL Database sources and sinks require mandatory security information if the installation is secure and uses an Oracle Wallet-based authentication. This information can be provided by adding a jar file to the `<MIGRATOR_HOME>/lib` directory.

Wallet-based Authentication

If an Oracle NoSQL Database installation uses Oracle Wallet-based authentication, you need an additional jar file that is part of the EE installation. For more information, see Oracle Wallet.

Without this jar file, you will get the following error message:

```
Could not find kvstore-ee.jar in lib directory. Copy kvstore-
ee.jar to lib directory.
```

To prevent the exception shown above, you must copy the `kvstore-ee.jar` file from your EE server package to the `<MIGRATOR_HOME>/lib` directory. *<MIGRATOR_HOME>* is the `nosql-migrator-M.N.O/` directory created by extracting the Oracle NoSQL Database Migrator package and M.N.O represent the software release.major.minor numbers. For example, `nosql-migrator-1.1.0/lib`.

> **Note:**
>
> The wallet-based authentication is supported ONLY in the Enterprise Edition (EE) of Oracle NoSQL Database.

Authenticating with Instance Principals

Instance principals is an IAM service feature that enables instances to be authorized actors (or principals) that can perform actions on service resources. Each compute instance has its own identity, and it authenticates using the certificates added to it.

Oracle NoSQL Database Migrator provides an option to connect to a NoSQL cloud and OCI Object Storage sources and sinks using instance principal authentication. It is only supported when the NoSQL Database Migrator tool is used within an OCI compute instance, for example, the NoSQL Database Migrator tool running in a VM hosted on OCI. To enable this feature use the useInstancePrincipal attribute of the NoSQL cloud source and sink configuration file. For more information on configuration parameters for different types of sources and sinks, see Source Configuration Templates and Sink Configuration Templates .

For more information on instance principals, see Calling Services from an Instance.

Source Configuration Templates

Learn about the source configuration file formats for each valid source and the purpose of each configuration parameter.

For the configuration file template, see **Configuration File** in Terminology used with Oracle NoSQL Database Migrator.

For details on valid sink formats for each of the source, see Sink Configuration Templates.

Topics

- JSON as the File Source
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from a JSON file as a source to a valid sink.
- JSON File in OCI Object Storage Bucket
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from a JSON file in the OCI Object Storage bucket as a source to a valid sink.
- MongoDB-Formatted JSON File
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from a MongoDB-Formatted JSON file as a source to a valid sink.

- **MongoDB-Formatted JSON File in OCI Object Storage bucket**
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from a MongoDB-Formatted JSON file in the OCI Object Storage bucket as a source to a valid sink.
- **DynamoDB-Formatted JSON File stored in AWS S3**
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from a DynamoDB-Formatted JSON file in the AWS S3 storage as a source to a valid sink.
- DynamoDB-Formatted JSON File
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from a DynamoDB-Formatted JSON file as a source to a valid sink.
- Oracle NoSQL Database
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from Oracle NoSQL Database tables as a source to a valid sink.
- Oracle NoSQL Database Cloud Service
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from Oracle NoSQL Database Cloud Service tables as a source to a valid sink.
- CSV as the File Source
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from CSV file as a source to a valid sink.
- CSV file in OCI Object Storage Bucket
  The source configuration template for the Oracle NoSQL Database Migrator to copy the data from CSV file stored in OCI Object Storage bucket as a source to a valid sink.

JSON as the File Source

The configuration file format for JSON File as a source of NoSQL Database Migrator is shown below.

You can migrate a JSON source file from a file path or a directory by specifying the file path or directory in the source configuration template.

A sample JSON source file is as follows:


```
{"Item":{"PK":{"S":"ACCT#82691500"},"SK":
{"S":"ACCT#82691500"},"AccountIndexId":{"S":"ACCT#82691500"},"Emailid":
{"S":"alejandro.rosalez11@example.org"},"AccountId":
{"N":"82691500"},"PlasticCardNumber":{"S":"9610432116466295"},"FirstName":
{"S":"Alejandro"},"Addresses":{"M":{"RESIDENCE":{"M":{"city":{"S":"Any
Town"},"country":{"S":"USA"},"street":{"S":"123 Any Street"}}},"BUSINESS":
{"M":{"city":{"S":"Anytown"},"country":{"S":"country"},"street":{"S":"221
Main Street"}}}}},"LastName":{"S":"Rosalez"}}}
{"Item":{"PK":{"S":"ACCT#76584123"},"SK":
{"S":"ACCT#76584123"},"AccountIndexId":{"S":"ACCT#76584123"},"Emailid":
{"S":"zhang.wei@example.com"},"AccountId":
{"N":"76584123"},"PlasticCardNumber":{"S":"4235400034568756"},"FirstName":
{"S":"Zhang"},"Addresses":{"M":{"RESIDENCE":{"M":{"city":{"S":"Any
Town"},"country":{"S":"USA"},"street":{"S":"135 Any Street"}}},"BUSINESS":
{"M":{"city":{"S":"AnyTown"},"country":{"S":"country"},"street":{"S":"100
Main Street"}}}}},"LastName":{"S":"Wei"},"AuthUsers":{"M":{"AUTHUSER-2":{"M":
{"Name":{"S":"Mateo Jackson"},"PlasticCardNumber":
```

```
{"S":"4036516984267960"}}},"AUTHUSER-1":{"M":{"Name":{"S":"Paulo
Santos"},"PlasticCardNumber":{"S":"4036546984262340"}}}}}}}
```

Source Configuration Template

```
"source" : {
  "type" : "file",
  "format" : "json",
  "dataPath": "</path/to/a/json/file>",
  "schemaInfo": {
                "schemaPath": "</path/to/schema/file>"
  }
}
```

Source Parameters

- type
- format
- dataPath
- schemaInfo
- schemaInfo.schemaPath

type

- **Purpose:** Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "file"`

format

- **Purpose:** Specifies the source format.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "json"`

dataPath

- **Purpose:** Specifies the absolute path to a file or directory containing the JSON data for migration.

  You must ensure that this data matches with the NoSQL table schema defined at the sink. If you specify a directory, the NoSQL Database Migrator identifies all the files with the `.json` extension in that directory for the migration. Sub-directories are not supported.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Specifying a JSON file

```
"dataPath" : "/home/user/sample.json"
```

– Specifying a directory

```
"dataPath" : "/home/user"
```

schemaInfo

- **Purpose:** Specifies the schema of the source data being migrated. This schema is passed to the NoSQL sink.
- **Data Type:** Object
- **Mandatory (Y/N):** N

schemaInfo.schemaPath

- **Purpose:** Specifies the absolute path to the schema definition file containing DDL statements for the NoSQL table being migrated.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**

```
"schemaInfo" : {
            "schemaPath" : "/home/user/mytable/Schema/schema.ddl"
             }
```

JSON File in OCI Object Storage Bucket

The configuration file format for JSON file in OCI Object Storage bucket as a source of NoSQL Database Migrator is shown below.

You can migrate a JSON file in the OCI Object Storage bucket by specifying the name of the bucket in the source configuration template.

A sample JSON source file in the OCI Object Storage bucket is as follows:


```
{"Item":{"PK":{"S":"ACCT#82691500"},"SK":
{"S":"ACCT#82691500"},"AccountIndexId":{"S":"ACCT#82691500"},"Emailid":
{"S":"alejandro.rosalez11@example.org"},"AccountId":
{"N":"82691500"},"PlasticCardNumber":{"S":"9610432116466295"},"FirstName":
{"S":"Alejandro"},"Addresses":{"M":{"RESIDENCE":{"M":{"city":{"S":"Any
Town"},"country":{"S":"USA"},"street":{"S":"123 Any Street"}}},"BUSINESS":
{"M":{"city":{"S":"Anytown"},"country":{"S":"country"},"street":{"S":"221
Main Street"}}}}},"LastName":{"S":"Rosalez"}}}
{"Item":{"PK":{"S":"ACCT#76584123"},"SK":
{"S":"ACCT#76584123"},"AccountIndexId":{"S":"ACCT#76584123"},"Emailid":
{"S":"zhang.wei@example.com"},"AccountId":
{"N":"76584123"},"PlasticCardNumber":{"S":"4235400034568756"},"FirstName":
{"S":"Zhang"},"Addresses":{"M":{"RESIDENCE":{"M":{"city":{"S":"Any
Town"},"country":{"S":"USA"},"street":{"S":"135 Any Street"}}},"BUSINESS":
{"M":{"city":{"S":"AnyTown"},"country":{"S":"country"},"street":{"S":"100
Main Street"}}}}},"LastName":{"S":"Wei"},"AuthUsers":{"M":{"AUTHUSER-2":{"M":
{"Name":{"S":"Mateo Jackson"},"PlasticCardNumber":
{"S":"4036516984267960"}}},"AUTHUSER-1":{"M":{"Name":{"S":"Paulo
Santos"},"PlasticCardNumber":{"S":"4036546984262340"}}}}}}
```

> **Note:**
>
> The valid sink types for OCI Object Storage source type are `nosqldb` and `nosqldb_cloud`.

Source Configuration Template

```
"source" : {
  "type" : "object_storage_oci",
  "format" : "json",
  "endpoint" : "<OCI Object Storage service endpoint URL or region
ID>",
  "namespace" : "<OCI Object Storage namespace>",
  "bucket" : "<bucket name>",
  "prefix" : "<object prefix>",
  "schemaInfo" : {
      "schemaObject" : "<object name>"
  },
  "credentials" : "</path/to/oci/config/file>",
  "credentialsProfile" : "<profile name in oci config file>",
  "useInstancePrincipal" : <true|false>
}
```

Source Parameters

- type
- format
- endpoint
- namespace
- bucket
- prefix
- schemaInfo
- schemaInfo.schemaObject
- credentials
- credentialsProfile
- useInstancePrincipal

type

- **Purpose:** Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "object_storage_oci"`

format

- **Purpose:** Specifies the source format.

- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "json"`

endpoint

- **Purpose:** Specifies the OCI Object Storage service endpoint URL or region ID.

  You can either specify the complete URL or the Region ID alone. See Data Regions and Associated Service URLs in *Using Oracle NoSQL Database Cloud Service* for the list of data regions supported for Oracle NoSQL Database Cloud Service.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Region ID: `"endpoint" : "us-ashburn-1"`
  - URL format: `"endpoint" : "https://objectstorage.us-ashburn-1.oraclecloud.com"`

namespace

- **Purpose:** Specifies the OCI Object Storage service namespace. This is an optional parameter. If you don't specify this parameter, the default namespace of the tenancy is used.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"namespace" : "my-namespace"`

bucket

- **Purpose:** Specifies the name of the bucket, which contains the source JSON files. Ensure that the required bucket already exists in the OCI Object Storage instance and has read permissions.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"bucket" : "staging_bucket"`

prefix

- **Purpose:** Used for filtering the objects that are being migrated from the bucket. All the objects with the given prefix present in the bucket are migrated. For more information about prefixes, see Object Naming Using Prefixes and Hierarchies.

  If you do not provide any value, no filter is applied and all the objects present in the bucket are migrated.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"prefix" : "my_table/Data/000000.json"` (migrates only `000000.json`)

2. `"prefix" : "my_table/Data"` (migrates all the objects with prefix `my_table/Data`)

schemaInfo

- **Purpose:** Specifies the schema of the source data being migrated. This schema is passed to the NoSQL sink.
- **Data Type:** Object
- **Mandatory (Y/N):** N

schemaInfo.schemaObject

- **Purpose:** Specifies the name of the object in the bucket where NoSQL table schema definitions for the data being migrated are stored.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**

```
"schemaInfo" : {
            "schemaObject" : "mytable/Schema/schema.ddl"
             }
```

credentials

- **Purpose:** Absolute path to a file containing OCI credentials.

  If not specified, it defaults to `$HOME/.oci/config`

  See Example Configuration for an example of the credentials file.

> **Note:**
>
> Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**

  1. `"credentials" : "/home/user/.oci/config"`
  2. `"credentials" : "/home/user/security/config"`

credentialsProfile

- **Purpose:** Name of the configuration profile to be used to connect to the Oracle NoSQL Database Cloud Service. User account credentials are referred to as a 'profile'.

  If you do not specify this value, it defaults to the `DEFAULT` profile.

> **Note:**
>
> This parameter is valid ONLY if the `credentials` parameter is specified.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"credentialsProfile" : "DEFAULT"`
  2. `"credentialsProfile": "ADMIN_USER"`

useInstancePrincipal

- **Purpose:** Specifies whether or not the NoSQL Migrator tool uses instance principal authentication to connect to Oracle NoSQL Database Cloud Service. For more information on Instance Principal authentication method, see Source and Sink Security.

  If not specified, it defaults to false.

> **Note:**
>
> – It is supported ONLY when the NoSQL Database Migrator tool is running within an OCI compute instance, for example, NoSQL Database Migrator tool running in a VM hosted on OCI.
>
> – Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useInstancePrincipal" : true`

MongoDB-Formatted JSON File

The configuration file format for MongoDB-formatted JSON File as a source of NoSQL Database Migrator is shown below.

You can copy a file or directory containing the MongoDB exported JSON data for migration by specifying the file or directory in the source configuration template.

A sample MongoDB-formatted JSON File is as follows:

```
{"_id":0,"name":"Aimee Zank","scores":
[{"score":1.463179736705023,"type":"exam"},
{"score":11.78273309957772,"type":"quiz"},
{"score":35.8740349954354,"type":"homework"}]}
{"_id":1,"name":"Aurelia Menendez","scores":
[{"score":60.06045071030959,"type":"exam"},
{"score":52.79790691903873,"type":"quiz"},
```

```
{"score":71.76133439165544,"type":"homework"}]}
{"_id":2,"name":"Corliss Zuk","scores":
[{"score":67.03077096065002,"type":"exam"},
{"score":6.301851677835235,"type":"quiz"},
{"score":66.28344683278382,"type":"homework"}]}
{"_id":3,"name":"Bao Ziglar","scores":
[{"score":71.64343899778332,"type":"exam"},
{"score":24.80221293650313,"type":"quiz"},
{"score":42.26147058804812,"type":"homework"}]}
{"_id":4,"name":"Zachary Langlais","scores":
[{"score":78.68385091304332,"type":"exam"},
{"score":90.2963101368042,"type":"quiz"},
{"score":34.41620148042529,"type":"homework"}]}
```

MongoDB supports two types of extensions to the JSON format of files, *Canonical mode* and *Relaxed mode*. You can supply the MongoDB-formatted JSON file that is generated using the *mongoexport* tool in either Canonical or Relaxed mode. Both the modes are supported by the NoSQL Database Migrator for migration.

For more information on the MongoDB Extended JSON (v2) file, See mongoexport_formats.

For more information on the generation of MongoDB-formatted JSON file, see mongoexport for more information.

Source Configuration Template

```
"source" : {
    "type" : "file",
    "format" : "mongodb_json",
    "dataPath": "</path/to/a/json/file>",
    "schemaInfo": {
                "schemaPath": "</path/to/schema/file>"
    }
}
```

Source Parameters

- type
- format
- dataPath
- schemaInfo
- schemaInfo.schemaPath

type

- **Purpose:** Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "file"`

format

- **Purpose:** Specifies the source format.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "mongodb_json"`

dataPath

- **Purpose:** Specifies the absolute path to a file or directory containing the MongoDB exported JSON data for migration.

  You must have generated these files using the *mongoexport* tool. See mongoexport for more information.

  You can supply the MongoDB-formatted JSON file that is generated using the mongoexport tool in either canonical or relaxed mode. Both the modes are supported by the NoSQL Database Migrator for migration.

  For more information on the MongoDB Extended JSON (v2) file, See mongoexport_formats.

  If you specify a directory, the NoSQL Database Migrator identifies all the files with the `.json` extension in that directory for the migration. Sub-directories are not supported. You must ensure that this data matches with the NoSQL table schema defined at the sink.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Specifying a MongoDB formatted JSON file

    ```
    "dataPath" : "/home/user/sample.json"
    ```
  - Specifying a directory

    ```
    "dataPath" : "/home/user"
    ```

schemaInfo

- **Purpose:** Specifies the schema of the source data being migrated. This schema is passed to the NoSQL sink.
- **Data Type:** Object
- **Mandatory (Y/N):** N

schemaInfo.schemaPath

- **Purpose:** Specifies the absolute path to the schema definition file containing DDL statements for the NoSQL table being migrated.
- **Data Type:** string
- **Mandatory (Y/N):** Y

- **Example:**

```
"schemaInfo" : {
            "schemaPath" : "/home/user/mytable/Schema/schema.ddl"
             }
```

MongoDB-Formatted JSON File in OCI Object Storage bucket

The configuration file format for MongoDB-Formatted JSON file in OCI Object Storage bucket as a source of NoSQL Database Migrator is shown below.

You can copy the MongoDB exported JSON data in the OCI Object Storage bucket for migration by specifying the name of the bucket in the source configuration template.

A sample MongoDB-formatted JSON File is as follows:

```
{"_id":0,"name":"Aimee Zank","scores":
[{"score":1.463179736705023,"type":"exam"},
{"score":11.78273309957772,"type":"quiz"},
{"score":35.8740349954354,"type":"homework"}]}
{"_id":1,"name":"Aurelia Menendez","scores":
[{"score":60.06045071030959,"type":"exam"},
{"score":52.79790691903873,"type":"quiz"},
{"score":71.76133439165544,"type":"homework"}]}
{"_id":2,"name":"Corliss Zuk","scores":
[{"score":67.03077096065002,"type":"exam"},
{"score":6.301851677835235,"type":"quiz"},
{"score":66.28344683278382,"type":"homework"}]}
{"_id":3,"name":"Bao Ziglar","scores":
[{"score":71.64343899778332,"type":"exam"},
{"score":24.80221293650313,"type":"quiz"},
{"score":42.26147058804812,"type":"homework"}]}
{"_id":4,"name":"Zachary Langlais","scores":
[{"score":78.68385091304332,"type":"exam"},
{"score":90.2963101368042,"type":"quiz"},
{"score":34.41620148042529,"type":"homework"}]}
```

Extract the data from MongoDB using the *mongoexport* utility and upload it to the OCI Object Storage bucket. See mongoexport for more information.

> **Note:**
>
> The valid sink types for OCI Object Storage source type are `nosqldb` and `nosqldb_cloud`.

Source Configuration Template

```
"source" : {
  "type" : "object_storage_oci",
  "format" : "mongodb_json",
  "endpoint" : "<OCI Object Storage service endpoint URL or region
ID>",
```

```
  "namespace" : "<OCI Object Storage namespace>",
  "bucket" : "<bucket name>",
  "prefix" : "<object prefix>",
  "schemaInfo" : {
      "schemaObject" : "<object name>"
  },
  "credentials" : "</path/to/oci/config/file>",
  "credentialsProfile" : "<profile name in oci config file>",
  "useInstancePrincipal" : <true|false>
}
```

Source Parameters

- type
- format
- endpoint
- namespace
- bucket
- prefix
- schemaInfo
- schemaInfo.schemaObject
- credentials
- credentialsProfile
- useInstancePrincipal

type

- **Purpose:** Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "object_storage_oci"`

format

- **Purpose:** Specifies the source format.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "mongodb_json"`

endpoint

- **Purpose:** Specifies the OCI Object Storage service endpoint URL or region ID.

  You can either specify the complete URL or the Region ID alone. See Data Regions and Associated Service URLs in *Using Oracle NoSQL Database Cloud Service* for the list of data regions supported for Oracle NoSQL Database Cloud Service.
- **Data Type:** string

- **Mandatory (Y/N):** Y
- **Example:**
  - Region ID: `"endpoint" : "us-ashburn-1"`
  - URL format: `"endpoint" : "https://objectstorage.us-ashburn-1.oraclecloud.com"`

namespace

- **Purpose:** Specifies the OCI Object Storage service namespace. This is an optional parameter. If you don't specify this parameter, the default namespace of the tenancy is used.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"namespace" : "my-namespace"`

bucket

- **Purpose:** Specifies the name of the bucket, which contains the source MongoDB-Formatted JSON files. Ensure that the required bucket already exists in the OCI Object Storage instance and has read permissions.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"bucket" : "staging_bucket"`

prefix

- **Purpose:** Used for filtering the objects that are being migrated from the bucket. All the objects with the given prefix present in the bucket are migrated. For more information about prefixes, see Object Naming Using Prefixes and Hierarchies.

  If you do not provide any value, no filter is applied and all the MongoDB JSON formatted objects present in the bucket are migrated. Extract the data from MongoDB using the mongoexport utility and upload it to the OCI Object Storage bucket. See mongoexport for more information.

  If you do not provide any value, no filter is applied and all the objects present in the bucket are migrated.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"prefix" : "mongo_export/Data/table.json"` (migrates only `table.json`)
  2. `"prefix" : "mongo_export/Data"` (migrates all the objects with prefix `mongo_export/Data`)

schemaInfo

- **Purpose:** Specifies the schema of the source data being migrated. This schema is passed to the NoSQL sink.
- **Data Type:** Object

- **Mandatory (Y/N):** N

schemaInfo.schemaObject

- **Purpose:** Specifies the name of the object in the bucket where NoSQL table schema definitions for the data being migrated are stored.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**

```
"schemaInfo" : {
            "schemaObject" : "mytable/Schema/schema.ddl"
             }
```

credentials

- **Purpose:** Absolute path to a file containing OCI credentials.

  If not specified, it defaults to `$HOME/.oci/config`

  See Example Configuration for an example of the credentials file.

> **Note:**
>
> Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"credentials" : "/home/user/.oci/config"`
  2. `"credentials" : "/home/user/security/config"`

credentialsProfile

- **Purpose:** Name of the configuration profile to be used to connect to the Oracle NoSQL Database Cloud Service. User account credentials are referred to as a 'profile'.

  If you do not specify this value, it defaults to the `DEFAULT` profile.

> **Note:**
>
> This parameter is valid ONLY if the `credentials` parameter is specified.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**

1. `"credentialsProfile" : "DEFAULT"`
2. `"credentialsProfile": "ADMIN_USER"`

useInstancePrincipal

- **Purpose:** Specifies whether or not the NoSQL Migrator tool uses instance principal authentication to connect to Oracle NoSQL Database Cloud Service. For more information on Instance Principal authentication method, see Source and Sink Security.

  If not specified, it defaults to false.

  > **Note:**
  >
  > – It is supported ONLY when the NoSQL Database Migrator tool is running within an OCI compute instance, for example, NoSQL Database Migrator tool running in a VM hosted on OCI.
  >
  > – Even though the credentials and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useInstancePrincipal" : true`

DynamoDB-Formatted JSON File stored in AWS S3

The configuration file format for DynamoDB-formatted JSON File in AWS S3 as a source of NoSQL Database Migrator is shown below.

You can migrate a file containing the DynamoDB exported JSON data from the AWS S3 storage by specifying the path in the source configuration template.

A sample DynamoDB-formatted JSON File is as follows:


```
{"Item":{"Id":{"N":"101"},"Phones":{"L":[{"L":[{"S":"555-222"},
{"S":"123-567"}]}]},"PremierCustomer":{"BOOL":false},"Address":{"M":
{"Zip":{"N":"570004"},"Street":{"S":"21 main"},"DoorNum":
{"N":"201"},"City":{"S":"London"}}},"FirstName":
{"S":"Fred"},"FavNumbers":{"NS":["10"]},"LastName":
{"S":"Smith"},"FavColors":{"SS":["Red","Green"]},"Age":{"N":"22"}}}
{"Item":{"Id":{"N":"102"},"Phones":{"L":[{"L":
[{"S":"222-222"}]}]},"PremierCustomer":{"BOOL":false},"Address":{"M":
{"Zip":{"N":"560014"},"Street":{"S":"32 main"},"DoorNum":
{"N":"1024"},"City":{"S":"Wales"}}},"FirstName":
{"S":"John"},"FavNumbers":{"NS":["10"]},"LastName":
{"S":"White"},"FavColors":{"SS":["Blue"]},"Age":{"N":"48"}}}
```

You must export the DynamoDB table to AWS S3 storage as specified in Exporting DynamoDB table data to Amazon S3.

The valid sink types for DynamoDB-formatted JSON stored in AWS S3 are `nosqldb` and `nosqldb_cloud`.

Source Configuration Template

```
"source" : {
    "type" : "aws_s3",
    "format" : "dynamodb_json",
    "s3URL" : "<S3 object url>",
    "credentials" : "</path/to/aws/credentials/file>",
    "credentialsProfile" : <"profile name in aws credentials file">
}
```

Source Parameters:

- type
- format
- s3URL
- credentials
- credentialsProfile

type

- **Purpose:**Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):**Y
- **Example:** `"type" : "aws_s3"`

format

- **Purpose:**Specifies the source format.
- **Data Type:** string
- **Mandatory (Y/N):**Y
- **Example:** `"format" : "dynamodb_json"`

> **Note:**
>
> If the value of the "**type**" is `aws_s3`, then format must be `dynamodb_json`.

s3URL

- **Purpose:** Specifies the URL of an exported DynamoDB table stored in AWS S3. You can obtain this URL from the AWS S3 console. Valid URL format is `https://<bucket-name>.<s3_endpoint>/<prefix>`. The migrator will look for `json.gz` files in the prefix for import.

> **Note:**
>
> You must export DynamoDB table as specified in Exporting DynamoDB table data to Amazon S3.

- **Data Type:** string
- **Mandatory:** Yes
- **Example**: `https://my-bucket.s3.ap-south-1.amazonaws.com/AWSDynamoDB/01649660790057-14f642be`

credentials

- **Purpose:** Specifies the absolute path to a file containing the AWS credentials. If not specified, it defaults to `$HOME/.aws/credentials`. Please refer to Configuration and credential file settings for more details on the credentials file.
- **Data Type:** string
- **Mandatory (Y/N):**N
- **Example**:

```
"credentials" : "/home/user/.aws/credentials"
"credentials" : "/home/user/security/credentials
```

> **Note:**
>
> The Migrator does not log any of the credentials information. You should properly protect the credentials file from unauthorized access.

credentialsProfile

- **Purpose:** Name of the profile in the AWS credentials file to be used to connect to AWS S3. User account credentials are referred to as a *profile*. If you do not specify this value, it defaults to the default profile. Please refer to Configuration and credential file settings for more details on the credentials file.
- **Data Type:** string
- **Mandatory (Y/N):**N
- **Example:**

```
"credentialsProfile" : "default"
        "credentialsProfile": "test"
```

DynamoDB-Formatted JSON File

The configuration file format for DynamoDB-formatted JSON File as a source of NoSQL Database Migrator is shown below.

You can migrate a file or directory containing the DynamoDB exported JSON data from a file system by specifying the path in the source configuration template.

A sample DynamoDB-formatted JSON File is as follows:

```
{"Item":{"Id":{"N":"101"},"Phones":{"L":[{"L":[{"S":"555-222"},
{"S":"123-567"}]}]},"PremierCustomer":{"BOOL":false},"Address":{"M":{"Zip":
{"N":"570004"},"Street":{"S":"21 main"},"DoorNum":{"N":"201"},"City":
{"S":"London"}}},"FirstName":{"S":"Fred"},"FavNumbers":{"NS":
["10"]},"LastName":{"S":"Smith"},"FavColors":{"SS":["Red","Green"]},"Age":
{"N":"22"}}}
{"Item":{"Id":{"N":"102"},"Phones":{"L":[{"L":
[{"S":"222-222"}]}]},"PremierCustomer":{"BOOL":false},"Address":{"M":{"Zip":
{"N":"560014"},"Street":{"S":"32 main"},"DoorNum":{"N":"1024"},"City":
{"S":"Wales"}}},"FirstName":{"S":"John"},"FavNumbers":{"NS":
["10"]},"LastName":{"S":"White"},"FavColors":{"SS":["Blue"]},"Age":
{"N":"48"}}}
```

You must copy the exported DynamoDB table data from AWS S3 storage to a local mounted file system.

The valid sink types for DynamoDB JSON file are `nosqldb` and `nosqldb_cloud`.

Source Configuration Template

```
"source" : {
    "type" : "file",
    "format" : "dynamodb_json",
    "dataPath" : "<path to a file or directory containing exported DDB table
data>"
}
```

Source Parameters:

- type
- format
- dataPath

type

- **Purpose:**Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):**Y
- **Example:** `"type" : "file"`

format

- **Purpose:**Specifies the source format.
- **Data Type:** string
- **Mandatory (Y/N):**Y
- **Example:** `"format" : "dynamodb_json"`

dataPath

- **Purpose:** Specifies the absolute path to a file or directory containing the exported DynamoDB table data. You must copy exported DynamoDB table data from AWS S3 to a

local mounted file system. You must ensure that this data matches with the NoSQL table schema defined at the sink. If you specify a directory, the NoSQL Database Migrator identifies all the files with the `.json.gz` extension in that directory and the `data`sub-directory.

- **Data Type:** string
- **Mandatory (Y/N):**Y
- **Example:**
  - Specifying a file

    ```
    "dataPath" : "/home/user/AWSDynamoDB/01639372501551-bb4dd8c3/
    data/zclclwucjy6v5mkefvckxzhfvq.json.gz"
    ```

  - Specifying a directory

    ```
    "dataPath" : "/home/user/AWSDynamoDB/01639372501551-bb4dd8c3"
    ```

Oracle NoSQL Database

The configuration file format for Oracle NoSQL Database as a source of NoSQL Database Migrator is shown below.

You can migrate a table from Oracle NoSQL Database by specifying the table name in the source configuration template.

A sample Oracle NoSQL Database table is as follows:

```
{"id":20,"firstName":"Jane","lastName":"Smith","otherNames":
[{"first":"Jane","last":"teacher"}],"age":25,"income":55000,"address":
{"city":"San Jose","number":201,"phones":
[{"area":608,"kind":"work","number":6538955},
{"area":931,"kind":"home","number":9533341},
{"area":931,"kind":"mobile","number":9533382}],"state":"CA","street":"A
tlantic Ave","zip":95005},"connections":[40,75,63],"expenses":null}
{"id":10,"firstName":"John","lastName":"Smith","otherNames":
[{"first":"Johny","last":"chef"}],"age":22,"income":45000,"address":
{"city":"Santa Cruz","number":101,"phones":
[{"area":408,"kind":"work","number":4538955},
{"area":831,"kind":"home","number":7533341},
{"area":831,"kind":"mobile","number":7533382}],"state":"CA","street":"P
acific Ave","zip":95008},"connections":[30,55,43],"expenses":null}
{"id":30,"firstName":"Adam","lastName":"Smith","otherNames":
[{"first":"Adam","last":"handyman"}],"age":45,"income":75000,"address":
{"city":"Houston","number":301,"phones":
[{"area":618,"kind":"work","number":6618955},
{"area":951,"kind":"home","number":9613341},
{"area":981,"kind":"mobile","number":9613382}],"state":"TX","street":"I
ndian Ave","zip":95075},"connections":[60,45,73],"expenses":null}
```

Source Configuration Template

```
"source" : {
    "type": "nosqldb",
```

```
    "table" : "<fully qualified table name>",
    "storeName" : "<store name>",
    "helperHosts" : ["hostname1:port1","hostname2:port2,..."],
    "security" : "</path/to/store/security/file>",
    "requestTimeoutMs" : 5000,
    "includeTTL": <true|false>
}
```

Source Parameters

- type
- table
- storeName
- helperHosts
- security
- requestTimeoutMs
- includeTTL

type

- **Purpose:** Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "nosqldb"`

table

- **Purpose:** Fully qualified table name from which to migrate the data.

  Format: `[namespace_name:]<table_name>`

  If the table is in the DEFAULT namespace, you can omit the `namespace_name`. The table must exist in the store.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - With the DEFAULT namespace `"table" :"mytable"`
  - With a non-default namespace `"table" : "mynamespace:mytable"`
  - To specify a child table `"table" : "mytable.child"`

storeName

- **Purpose:** Name of the Oracle NoSQL Database store.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"storeName" : "kvstore"`

helperHosts

- **Purpose:** A list of host and registry port pairs in the `hostname:port` format. Delimit each item in the list using a comma. You must specify at least one helper host.
- **Data Type:** array of strings
- **Mandatory (Y/N):** Y
- **Example:** `"helperHosts" : ["localhost:5000","localhost:6000"]`

security

- **Purpose:**

  If your store is a secure store, provide the absolute path to the security login file that contains your store credentials. See Configuring Security with Remote Access in *Administrator's Guide* to know more about the security login file.

  You can use either password file based authentication or wallet based authentication. However, the wallet based authentication is supported only in the Enterprise Edition (EE) of Oracle NoSQL Database. For more information on wallet-based authentication, see Source and Sink Security.

  The Community Edition(CE) edition supports password file based authentication only.
- **Data Type:** string
- **Mandatory (Y/N):** Y for a secure store
- **Example:**
  `"security" : "/home/user/client.credentials"`

  Example security file content for password file based authentication:

  ```
  oracle.kv.password.noPrompt=true
  oracle.kv.auth.username=admin
  oracle.kv.auth.pwdfile.file=/home/nosql/login.passwd
  oracle.kv.transport=ssl
  oracle.kv.ssl.trustStore=/home/nosql/client.trust
  oracle.kv.ssl.protocols=TLSv1.2
  oracle.kv.ssl.hostnameVerifier=dnmatch(CN\=NoSQL)
  ```

  Example security file content for wallet based authentication:

  ```
  oracle.kv.password.noPrompt=true
  oracle.kv.auth.username=admin
  oracle.kv.auth.wallet.dir=/home/nosql/login.wallet
  oracle.kv.transport=ssl
  oracle.kv.ssl.trustStore=/home/nosql/client.trust
  oracle.kv.ssl.protocols=TLSv1.2
  oracle.kv.ssl.hostnameVerifier=dnmatch(CN\=NoSQL)
  ```

requestTimeoutMs

- **Purpose:** Specifies the time to wait for each read operation from the store to complete. This is provided in milliseconds. The default value is 5000. The value can be any positive integer.

- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"requestTimeoutMs" : 5000`

includeTTL

- **Purpose:** Specifies whether or not to include TTL metadata for table rows when exporting Oracle NoSQL Database tables. If set to true, the TTL data for rows also gets included in the data provided by the source. TTL is present in the _metadata JSON object associated with each row. The expiration time for each row gets exported as the number of milliseconds since the UNIX epoch (Jan 1st, 1970).

  If you do not specify this parameter, it defaults to `false`.

  Only the rows having a positive expiration value for TTL get included as part of the exported rows. If a row does not expire, which means TTL=0, then its TTL metadata is not included explicitly. For example, if ROW1 expires at 2021-10-19 00:00:00 and ROW2 does not expire, the exported data looks like as follows:

```
//ROW1
{
  "id" : 1,
  "name" : "abc",
  "_metadata" : {
    "expiration" : 1634601600000
  }
}

//ROW2
{
  "id" : 2,
  "name" : "xyz"
}
```

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"includeTTL" : true`

Oracle NoSQL Database Cloud Service

The configuration file format for Oracle NoSQL Database Cloud Service as a source of NoSQL Database Migrator is shown below.

You can migrate a table from Oracle NoSQL Database Cloud Service by specifying the name or OCID of the compartment in which the table resides in the source configuration template.

A sample Oracle NoSQL Database Cloud Service table is as follows:

```
{"id":20,"firstName":"Jane","lastName":"Smith","otherNames":
[{"first":"Jane","last":"teacher"}],"age":25,"income":55000,"address":
{"city":"San Jose","number":201,"phones":
[{"area":608,"kind":"work","number":6538955},
{"area":931,"kind":"home","number":9533341},
{"area":931,"kind":"mobile","number":9533382}],"state":"CA","street":"Atlanti
```

```
c Ave","zip":95005},"connections":[40,75,63],"expenses":null}
{"id":10,"firstName":"John","lastName":"Smith","otherNames":
[{"first":"Johny","last":"chef"}],"age":22,"income":45000,"address":
{"city":"Santa Cruz","number":101,"phones":
[{"area":408,"kind":"work","number":4538955},
{"area":831,"kind":"home","number":7533341},
{"area":831,"kind":"mobile","number":7533382}],"state":"CA","street":"P
acific Ave","zip":95008},"connections":[30,55,43],"expenses":null}
{"id":30,"firstName":"Adam","lastName":"Smith","otherNames":
[{"first":"Adam","last":"handyman"}],"age":45,"income":75000,"address":
{"city":"Houston","number":301,"phones":
[{"area":618,"kind":"work","number":6618955},
{"area":951,"kind":"home","number":9613341},
{"area":981,"kind":"mobile","number":9613382}],"state":"TX","street":"I
ndian Ave","zip":95075},"connections":[60,45,73],"expenses":null}
```

Source Configuration Template

```
"source" : {
  "type" : "nosqldb_cloud",
  "endpoint" : "<Oracle NoSQL Cloud Service Endpoint. You can either
specify  the complete URL or the Region ID alone>",
  "table" : "<table name>",
  "compartment" : "<OCI compartment name or id>",
  "credentials" : "</path/to/oci/credential/file>",
  "credentialsProfile" : "<oci credentials profile name>",
  "readUnitsPercent" : <table readunits percent>,
  "requestTimeoutMs" : <timeout in milli seconds>,
  "useInstancePrincipal" : <true|false>,
  "includeTTL": <true|false>
}
```

Source Parameters

- type
- endpoint
- table
- compartment
- credentials
- credentialsProfile
- readUnitsPercent
- requestTimeoutMs
- useInstancePrincipal
- includeTTL

type

- **Purpose:** Identifies the source type.
- **Data Type:** string

- **Mandatory (Y/N):** Y
- **Example:** `"type" : "nosqldb_cloud"`

endpoint

- **Purpose:** Specifies the Service Endpoint of the Oracle NoSQL Database Cloud Service.

  You can either specify the complete URL or the Region ID alone. See Data Regions and Associated Service URLs in *Using Oracle NoSQL Database Cloud Service* for the list of data regions supported for Oracle NoSQL Database Cloud Service.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Region ID: `"endpoint" : "us-ashburn-1"`
  - URL format: `"endpoint" : "https://nosql.us-ashburn-1.oci.oraclecloud.com/"`

table

- **Purpose:** Name of the table from which to migrate the data.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - To specify a table `"table" : "myTable"`
  - To specify a child table `"table" : "mytable.child"`

compartment

- **Purpose:** Specifies the name or OCID of the compartment in which the table resides.

  If you do not provide any value, it defaults to the *root* compartment.

  You can find your compartment's OCID from the Compartment Explorer window under Governance in the OCI Cloud Console.
- **Data Type:** string
- **Mandatory (Y/N):** Yes, if the table is not in the root compartment of the tenancy OR when the `useInstancePrincipal` parameter is set to true.

> **Note:**
>
> If the `useInstancePrincipal` parameter is set to true, the compartment must specify the compartment OCID and not the name.

- **Example:**
  - Compartment name

    `"compartment" : "mycompartment"`
  - Compartment name qualified with its parent compartment

    `"compartment" : "parent.childcompartment"`

- No value provided. Defaults to the root compartment.

  ```
  "compartment": ""
  ```

- Compartment OCID

  ```
  "compartment" : "ocid1.tenancy.oc1...4ksd"
  ```

credentials

- **Purpose:** Absolute path to a file containing OCI credentials.

  If not specified, it defaults to `$HOME/.oci/config`

  See Example Configuration for an example of the credentials file.

  > **Note:**
  >
  > Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**

  1. `"credentials" : "/home/user/.oci/config"`
  2. `"credentials" : "/home/user/security/config"`

credentialsProfile

- **Purpose:** Name of the configuration profile to be used to connect to the Oracle NoSQL Database Cloud Service. User account credentials are referred to as a 'profile'.

  If you do not specify this value, it defaults to the `DEFAULT` profile.

  > **Note:**
  >
  > This parameter is valid ONLY if the `credentials` parameter is specified.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**

  1. `"credentialsProfile" : "DEFAULT"`
  2. `"credentialsProfile": "ADMIN_USER"`

readUnitsPercent

- **Purpose:** Percentage of table read units to be used while migrating the NoSQL table.

  The default value is 90. The valid range is any integer between 1 to 100. Please note that amount of time required to migrate data is directly proportional to this attribute. It's better to increase the read throughput of the table for the migration activity. You can reduce the read throughput after the migration process completes.

  To learn the daily limits on throughput changes, see Cloud Limits in *Using Oracle NoSQL Database Cloud Service*.

  The default value is 90. The valid range is any integer between 1 to 100.

  > **Note:**
  >
  > The time required for the data migration is directly proportional to the `writeUnitsPercent` value.

  See Troubleshooting the Oracle NoSQL Database Migrator to learn how to use this attribute to improve the data migration speed.

- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"readUnitsPercent" : 90`

requestTimeoutMs

- **Purpose:** Specifies the time to wait for each read operation in the sink to complete. This is provided in milliseconds. The default value is 5000. The value can be any positive integer.
- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"requestTimeoutMs" : 5000`

useInstancePrincipal

- **Purpose:** Specifies whether or not the NoSQL Migrator tool uses instance principal authentication to connect to Oracle NoSQL Database Cloud Service. For more information on Instance Principal authentication method, see Source and Sink Security.

  If not specified, it defaults to false.

> **Note:**
>
> – It is supported ONLY when the NoSQL Database Migrator tool is running within an OCI compute instance, for example, NoSQL Database Migrator tool running in a VM hosted on OCI.
>
> – Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useInstancePrincipal" : true`

includeTTL

- **Purpose:** Specifies whether or not to include TTL metadata for table rows when exporting Oracle NoSQL Database tables. If set to true, the TTL data for rows also gets included in the data provided by the source. TTL is present in the _metadata JSON object associated with each row. The expiration time for each row gets exported as the number of milliseconds since the UNIX epoch (Jan 1st, 1970).

  If you do not specify this parameter, it defaults to `false`.

  Only the rows having a positive expiration value for TTL get included as part of the exported rows. If a row does not expire, which means TTL=0, then its TTL metadata is not included explicitly. For example, if ROW1 expires at 2021-10-19 00:00:00 and ROW2 does not expire, the exported data looks like as follows:

```
//ROW1
{
  "id" : 1,
  "name" : "abc",
  "_metadata" : {
    "expiration" : 1634601600000
  }
}

//ROW2
{
  "id" : 2,
  "name" : "xyz"
}
```

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"includeTTL" : true`

CSV as the File Source

The configuration file format for the CSV file as a source of NoSQL Database Migrator is shown below. The CSV file must conform to the `RFC4180` format.

You can migrate a CSV file or a directory containing the CSV data by specifying the file name or directory in the source configuration template.

A sample CSV file is as follows:

```
1,"Computer Science","San Francisco","2500"
2,"Bio-Technology","Los Angeles","1200"
3,"Journalism","Las Vegas","1500"
4,"Telecommunication","San Francisco","2500"
```

Source Configuration Template

```
"source" : {
    "type" : "file",
    "format" : "csv",
    "dataPath": "</path/to/a/csv/file-or-directory>",
    "hasHeader" : <true | false>,
    "columns" : ["column1", "column2", ....],
    "csvOptions" : {
                 "trim" : <true | false>,
                 "encoding" : "<character set encoding>"
    }
}
```

Source Parameters

- type
- format
- dataPath
- hasHeader
- columns
- csvOptions
- csvOptions.trim
- csvOptions.encoding

type

- **Purpose:** Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "file"`

format

- **Purpose:** Specifies the source format.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "csv"`

datapath

- **Purpose:** Specifies the absolute path to a file or directory containing the CSV data for migration. If you specify a directory, NoSQL Database Migrator imports all the files with the `.csv` or `.CSV` extension in that directory. All the CSV files are copied into a single table, but not in any particular order.

  CSV files must conform to the `RFC4180` standard. You must ensure that the data in each CSV file matches with the NoSQL Database table schema defined in the sink table. Sub-directories are not supported.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Specifying a CSV file

    ```
    "dataPath" : "/home/user/sample.csv"
    ```
  - Specifying a directory

    ```
    "dataPath" : "/home/user"
    ```

> **Note:**
>
> The CSV files must contain only scalar values. Importing CSV files containing complex types such as MAP, RECORD, ARRAY, and JSON is not supported. The NoSQL Database Migrator tool does not check for the correctness of the data in the input CSV file. The NoSQL Database Migrator tool supports the importing of CSV data that conforms to the `RFC4180` format. CSV files containing data that does not conform to the `RFC4180` standard may not get copied correctly or may result in an error. If the input data is corrupted, the NoSQL Database Migrator tool will not parse the CSV records. If any errors are encountered during migration, the NoSQL Database Migrator tool logs the information about the failed input records for debugging and informative purposes. For more details, see *Logging Migrator Progress* in Using Oracle NoSQL Data Migrator.

hasHeader

- **Purpose:** Specifies if the CSV file has a header or not. If this is set to `true`, the first line is ignored. If it is set to `false`, the first line is considered a CSV record. The default value is `false`.
- **Data Type:** Boolean

- **Mandatory (Y/N):** N
- **Example:** `"hasHeader" : "false"`

columns

- **Purpose:** Specifies the list of NoSQL Database table column names. The order of the column names indicates the mapping of the CSV file fields with corresponding NoSQL Database table columns. If the order of the input CSV file columns does not match the existing or newly created NoSQL Database table columns, you can map the ordering using this parameter. Also, when importing into a table that has an Identity Column, you can skip the Identity column name in the `columns` configuration.

> **Note:**
>
> – If the NoSQL Database table has additional columns that are not available in the CSV file, the values of the missing columns are updated with the default value as defined in the NoSQL Database table. If a default value is not provided, a Null value is inserted during migration. For more information on default values, see Data Type Definitions section in the *SQL Reference Guide*.
>
> – If the CSV file has additional columns that are not defined in the NoSQL Database table, the additional column information is ignored.
>
> – If any value in the CSV record is empty, it is set to the default value of the corresponding columns in the NoSQL Database table. If a default value is not provided, a Null value is inserted during migration.

- **Data Type:** Array of Strings
- **Mandatory (Y/N):** N
- **Example:** `"columns" : ["table_column_1", "table_column_2"]`

csvOptions

- **Purpose:** Specifies the formatting options for a CSV file. Provide the character set encoding format of the CSV file and choose whether or not to trim the blank spaces.
- **Data Type:** Object
- **Mandatory (Y/N):** N

csvOptions.trim

- **Purpose:** Specifies if the leading and trailing blanks of a CSV field value must be trimmed. The default value is `false`.
- **Data Type:** Boolean
- **Mandatory (Y/N):** N
- **Example:** `"trim" : "true"`

csvOptions.encoding

- **Purpose:** Specifies the character set to decode the CSV file. The default value is `UTF-8`. The supported character sets are `US-ASCII`, `ISO-8859-1`, `UTF-8`, and `UTF-16`.

- **Data Type:** String
- **Mandatory (Y/N):** N
- **Example:** `"encoding" : "UTF-8"`

CSV file in OCI Object Storage Bucket

The configuration file format for the CSV file in OCI Object Storage bucket as a source of NoSQL Database Migrator is shown below. The CSV file must conform to the `RFC4180` format.

You can migrate a CSV file in the OCI Object Storage bucket by specifying the name of the bucket in the source configuration template.

A sample CSV file in the OCI Object Storage bucket is as follows:

```
1,"Computer Science","San Francisco","2500"
2,"Bio-Technology","Los Angeles","1200"
3,"Journalism","Las Vegas","1500"
4,"Telecommunication","San Francisco","2500"
```

> **Note:**
>
> The valid sink types for OCI Object Storage source type are `nosqldb` and `nosqldb_cloud`.

Source Configuration Template

```
"source" : {
  "type" : "object_storage_oci",
  "format" : "csv",
  "endpoint" : "<OCI Object Storage service endpoint URL or region
ID>",
  "namespace" : "<OCI Object Storage namespace>",
  "bucket" : "<bucket name>",
  "prefix" : "<object prefix>",
  "credentials" : "</path/to/oci/config/file>",
  "credentialsProfile" : "<profile name in oci config file>",
  "useInstancePrincipal" : <true|false>,
   "hasHeader" : <true | false>,
   "columns" : ["column1", "column2", ....],
   "csvOptions" : {
                  "trim" : <true | false>,
                  "encoding" : "<character set encoding>"
   }
 }
```

Source Parameters

- type
- format

- endpoint
- namespace
- bucket
- prefix
- credentials
- credentialsProfile
- useInstancePrincipal
- hasHeader
- columns
- csvOptions
- csvOptions.trim
- csvOptions.encoding

type

- **Purpose:** Identifies the source type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "object_storage_oci"`

format

- **Purpose:** Specifies the source format.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "csv"`

endpoint

- **Purpose:** Specifies the OCI Object Storage service endpoint URL or region ID.

  You can either specify the complete URL or the Region ID alone. See Data Regions and Associated Service URLs in *Using Oracle NoSQL Database Cloud Service* for the list of data regions supported for Oracle NoSQL Database Cloud Service.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Region ID: `"endpoint" : "us-ashburn-1"`
  - URL format: `"endpoint" : "https://objectstorage.us-ashburn-1.oraclecloud.com"`

namespace

- **Purpose:** Specifies the OCI Object Storage service namespace. This is an optional parameter. If you don't specify this parameter, the default namespace of the tenancy is used.

- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"namespace" : "my-namespace"`

bucket

- **Purpose:** Specifies the name of the bucket, which contains the source CSV files. The NoSQL Database Migrator imports all the files with the `.csv` or `.CSV` extension object-wise and copies them into a single table in the same order.

  Ensure that the required bucket already exists in the OCI Object Storage instance and has read permissions.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"bucket" : "staging_bucket"`

> **Note:**
>
> The CSV files must contain only scalar values. Importing CSV files containing complex types such as MAP, RECORD, ARRAY, and JSON is not supported. The NoSQL Database Migrator tool does not check for the correctness of the data in the input CSV file. The NoSQL Database Migrator tool supports the importing of CSV data that conforms to the `RFC4180` format. CSV files containing data that does not conform to the `RFC4180` standard may not get copied correctly or may result in an error. If the input data is corrupted, the NoSQL Database Migrator tool will not parse the CSV records. If any errors are encountered during migration, the NoSQL Database Migrator tool logs the information about the failed input records for debugging and informative purposes. For more details, see *Logging Migrator Progress* in Using Oracle NoSQL Data Migrator.

prefix

- **Purpose:** Used for filtering the objects that are being migrated from the bucket. All the objects with the given prefix present in the bucket are migrated. For more information about prefixes, see Object Naming Using Prefixes and Hierarchies.

  If you do not provide any value, filter is not applied and all the objects present in the bucket are migrated.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"prefix" : "my_table/Data/000000.json"` (migrates only `000000.json`)
  2. `"prefix" : "my_table/Data"` (migrates all the objects with prefix `my_table/Data`)

credentials

- **Purpose:** Absolute path to a file containing OCI credentials.

If not specified, it defaults to `$HOME/.oci/config`

See Example Configuration for an example of the credentials file.

> **Note:**
>
> You must specify either `credentials` or **useInstancePrincipal** parameters in the configuration template.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"credentials" : "/home/user/.oci/config"`
  2. `"credentials" : "/home/user/security/config"`

credentialsProfile

- **Purpose:** Name of the configuration profile to be used to connect to the Oracle NoSQL Database Cloud Service. User account credentials are referred to as a *profile*.

  If you do not specify this value, it defaults to the `DEFAULT` profile.

> **Note:**
>
> This parameter is valid only if the credentials parameter is specified.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"credentialsProfile" : "DEFAULT"`
  2. `"credentialsProfile": "ADMIN_USER"`

useInstancePrincipal

- **Purpose:** Specifies whether or not the NoSQL Database Migrator tool uses instance principal authentication to connect to Oracle NoSQL Database Cloud Service. For more information on Instance Principal authentication method, see Source and Sink Security.

  The default value is `false`.

> **Note:**
>
> – The authentication with Instance Principals is supported only when the NoSQL Database Migrator tool is running within an OCI compute instance, for example, NoSQL Database Migrator tool running in a VM hosted on OCI.
>
> – You must specify either `credentials` or `useInstancePrincipal` parameters in the configuration template.

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useInstancePrincipal" : true`

hasHeader

- **Purpose:** Specifies if the CSV file has a header or not. If this is set to `true`, the first line is ignored. If it is set to `false`, the first line is considered a CSV record. The default value is `false`.
- **Data Type:** Boolean
- **Mandatory (Y/N):** N
- **Example:** `"hasHeader" : "false"`

columns

- **Purpose:** Specifies the list of NoSQL Database table column names. The order of the column names indicates the mapping of the CSV file fields with corresponding NoSQL Database table columns. If the order of the input CSV file columns does not match the existing or newly created NoSQL Database table columns, you can map the ordering using this parameter. Also, when importing into a table that has an Identity Column, you can skip the Identity column name in the `columns` configuration.

> **Note:**
>
> – If the NoSQL Database table has additional columns that are not available in the CSV file, the values of the missing columns are updated with the default value as defined in the NoSQL Database table. If a default value is not provided, a Null value is inserted during migration. For more information on default values, see Data Type Definitions section in the *SQL Reference Guide*.
>
> – If the CSV file has additional columns that are not defined in the NoSQL Database table, the additional column information is ignored.
>
> – If any value in the CSV record is empty, it is set to the default value of the corresponding columns in the NoSQL Database table. If a default value is not provided, a Null value is inserted during migration.

- **Data Type:** Array of Strings
- **Mandatory (Y/N):** N
- **Example:** `"columns" : ["table_column_1", "table_column_2"]`

csvOptions

- **Purpose:** Specifies the formatting options for a CSV file. Provide the character set encoding format of the CSV file and choose whether or not to trim the blank spaces.
- **Data Type:** Object
- **Mandatory (Y/N):** N

csvOptions.trim

- **Purpose:** Specifies if the leading and trailing blanks of a CSV field value must be trimmed. The default value is `false`.
- **Data Type:** Boolean
- **Mandatory (Y/N):** N
- **Example:** `"trim" : "true"`

csvOptions.encoding

- **Purpose:** Specifies the character set to decode the CSV file. The default value is `UTF-8`. The supported character sets are `US-ASCII`, `ISO-8859-1`, `UTF-8`, and `UTF-16`.
- **Data Type:** String
- **Mandatory (Y/N):** N
- **Example:** `"encoding" : "UTF-8"`

Sink Configuration Templates

Learn about the sink configuration file formats for each valid sink and the purpose of each configuration parameter.

For the configuration file template, see **Configuration File** in Terminology used with Oracle NoSQL Database Migrator.
For details on valid source formats for each of the sinks, see Source Configuration Templates.

Topics

- JSON as the File Sink
  The sink configuration template for the Oracle NoSQL Database Migrator to copy the data from a valid source to a JSON file as the sink.
- Parquet File
  The sink configuration template for the Oracle NoSQL Database Migrator to copy the data from a valid source to a Parquet file as the sink.
- JSON File in OCI Object Storage Bucket
  The sink configuration template for the Oracle NoSQL Database Migrator to copy the data from a valid source to a JSON file in the OCI Object Storage bucket as the sink.
- Parquet File in OCI Object Storage Bucket

The sink configuration template for the Oracle NoSQL Database Migrator to copy the data from a valid source to a Parquet file in the OCI Object Storage bucket as the sink.

- Oracle NoSQL Database
  The sink configuration template for the Oracle NoSQL Database Migrator to copy the data from a valid source to Oracle NoSQL Database tables as the sink.
- Oracle NoSQL Database Cloud Service
  The sink configuration template for the Oracle NoSQL Database Migrator to copy the data from a valid source to Oracle NoSQL Database Cloud Service tables as the sink.

JSON as the File Sink

The configuration file format for JSON File as a sink of NoSQL Database Migrator is shown below.

Sink Configuration Template

```
"sink" : {
    "type" : "file",
    "format" : "json",
    "dataPath": "</path/to/a/file>",
    "schemaPath" : "<path/to/a/file>",
    "pretty" : <true|false>,
    "useMultiFiles" : <true|false>,
    "chunkSize" : <size in MB>
}
```

Sink Parameters

- type
- format
- dataPath
- schemaPath
- pretty
- useMultiFiles
- chunkSize

type

- **Purpose:** Identifies the sink type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "file"`

format

- **Purpose:** Specifies the sink format.
- **Data Type:** string

- **Mandatory (Y/N):** Y
- **Example:** `"format" : "json"`

dataPath

- **Purpose:** Specifies the absolute path to a file where the source data will be copied in the JSON format.

  If the file does not exist in the specified data path, the NoSQL Database Migrator creates it. If it exists already, the NoSQL Database Migrator will overwrite its contents with the source data.

  You must ensure that the parent directory for the file specified in the data path is valid.

  > **Note:**
  >
  > If the `useMultiFiles` parameter is set to true, specify the path to a directory else specifies the path to the file.

- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - With `useMultiFiles` parameter set to true
    `"dataPath" :"/home/user/data"`
  - With `useMultiFiles` parameter not specified or it is set to false
    `"dataPath" :"/home/user/sample.json"`

schemaPath

- **Purpose:** Specifies the absolute path to write schema information provided by the source.

  If this value is not defined, the source schema information will not be migrated to the sink. If this value is specified, the migrator utility writes the schema of the source table into the file specified here.

  The schema information is written as one DDL command per line in this file. If the file does not exist in the specified data path, NoSQL Database Migrator creates it. If it exists already, NoSQL Database Migrator will overwrite its contents with the source data. You must ensure that the parent directory for the file specified in the data path is valid.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:** `"schemaPath" : "/home/user/schema_file"`

pretty

- **Purpose:** Specifies whether to beautify the JSON output to increase readability or not.

  If not specified, it defaults to false.

- **Data Type:** boolean
- **Mandatory (Y/N):** N

- **Example:** `"pretty" : true`

useMultiFiles

- **Purpose:** Specifies whether or not to split the NoSQL table data into multiple files when migrating source data to a file.

  If not specified, it defaults to false.

  If set to true, when migrating source data to a file, the NoSQL table data is split into multiple smaller files. For example, `<chunk>.json`, where chunk=000000,000001,000002, and so forth.

```
dataPath
         |--000000.json
         |--000001.json
```

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useMultiFiles" : true`

chunkSize

- **Purpose:** Specifies the maximum size of a "chunk" of table data to be stored at the sink. During migration, a table is split into chunkSize chunks and each chunk is written as a separate file to the sink. When the source data being migrated exceeds this size, a new file is created.

  If not specified, defaults to 32MB. The valid value is an integer between 1 to 1024.

> **Note:**
>
> This parameter is applicable ONLY when the `useMultiFiles` parameter is set to true.

- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"chunkSize" : 40`

Parquet File

The configuration file format for Parquet File as a sink of NoSQL Database Migrator is shown below.

Sink Configuration Template

```
"sink" : {
    "type" : "file",
    "format" : "parquet",
    "dataPath": "</path/to/a/dir>",
    "chunkSize" : <size in MB>,
    "compression": "<SNAPPY|GZIP|NONE>",
    "parquetOptions": {
```

```
      "useLogicalJson": <true|false>,
      "useLogicalEnum": <true|false>,
      "useLogicalUUID": <true|false>,
      "truncateDoubleSpecials": <true|false>
    }
}
```

Sink Parameters

- type
- format
- dataPath
- chunkSize
- compression
- parquetOptions
- parquetOptions.useLogicalJson
- parquetOptions.useLogicalEnum
- parquetOptions.useLogicalUUID
- parquetOptions.truncateDoubleSpecials

type

- **Purpose:** Identifies the sink type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "file"`

format

- **Purpose:** Specifies the sink format.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "parquet"`

dataPath

- **Purpose:** Specifies the path to a directory to use for storing the migrated NoSQL table data. Ensure that the directory already exists and has read and write permissions.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"dataPath" : "/home/user/migrator/my_table"`

chunkSize

- **Purpose:** Specifies the maximum size of a "chunk" of table data to be stored at the sink. During migration, a table is split into chunkSize chunks and each chunk is written as a

separate file to the sink. When the source data being migrated exceeds this size, a new file is created.

If not specified, defaults to 32MB. The valid value is an integer between 1 to 1024.

- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"chunkSize" : 40`

compression

- **Purpose:** Specifies the compression type to use to compress the Parquet data. Valid values are `SNAPPY`, `GZIP`, and `NONE`.

  If not specified, it defaults to `SNAPPY`.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:** `"compression" : "GZIP"`

parquetOptions

- **Purpose:** Species the options to select Parquet logical types for NoSQL ENUM, JSON, and UUID columns.

  If you do not specify this parameter, the NoSQL Database Migrator writes the data of ENUM, JSON, and UUID columns as String.
- **Data Type:** object
- **Mandatory (Y/N):** N

parquetOptions.useLogicalJson

- **Purpose:** Specifies whether or not to write NoSQL JSON column data as Parquet logical JSON type. For more information see Parquet Logical Type Definitions.

  If not specified or set to false, NoSQL Database Migrator writes the NoSQL JSON column data as String.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useLogicalJson" : true`

parquetOptions.useLogicalEnum

- **Purpose:** Specifies whether or not to write NoSQL ENUM column data as Parquet logical ENUM type. For more information see Parquet Logical Type Definitions.

  If not specified or set to false, NoSQL Database Migrator writes the NoSQL ENUM column data as String.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useLogicalEnum" : true`

parquetOptions.useLogicalUUID

- **Purpose:** Specifies whether or not to write NoSQL UUID column data as Parquet logical UUID type. For more information see Parquet Logical Type Definitions.

  If not specified or set to false, NoSQL Database Migrator writes the NoSQL UUID column data as String.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useLogicalUUID" : true`

parquetOptions.truncateDoubleSpecials

- **Purpose:** Specifies whether or not to truncate the double +Infinity, -Infinity, and Nan values.

  By default, it is set to `false`. If set to `true`,
  - Positive_Infinity is truncated to Double.MAX_VALUE.
  - NEGATIVE_INFINITY is truncated to -Double.MAX_VALUE.
  - NaN is truncated to 9.9999999999999990E307.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"truncateDoubleSpecials" : true`

JSON File in OCI Object Storage Bucket

The configuration file format for JSON file in OCI Object Storage bucket as a sink of NoSQL Database Migrator is shown below.

> **Note:**
>
> The valid source types for OCI Object Storage source type are `nosqldb` and `nosqldb_cloud`.

Sink Configuration Template

```
"sink" : {
  "type" : "object_storage_oci",
  "format" : "json",
  "endpoint" : "<OCI Object Storage service endpoint URL or region ID>",
  "namespace" : "<OCI Object Storage namespace>",
  "bucket" : "<bucket name>",
  "prefix" : "<object prefix>",
  "chunkSize" : <size in MB>,
  "pretty" : <true|false>,
  "credentials" : "</path/to/oci/config/file>",
  "credentialsProfile" : "<profile name in oci config file>",
```

```
  "useInstancePrincipal" : <true|false>
}
```

Source Parameters

- type
- format
- endpoint
- namespace
- bucket
- prefix
- chunkSize
- pretty
- credentials
- credentialsProfile
- useInstancePrincipal

type

- **Purpose:** Identifies the sink type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "object_storage_oci"`

format

- **Purpose:** Specifies the sink format.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "json"`

endpoint

- **Purpose:** Specifies the OCI Object Storage service endpoint URL or region ID.

  You can either specify the complete URL or the Region ID alone. See Data Regions and Associated Service URLs in *Using Oracle NoSQL Database Cloud Service* for the list of data regions supported for Oracle NoSQL Database Cloud Service.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Region ID: `"endpoint" : "us-ashburn-1"`
  - URL format: `"endpoint" : "https://objectstorage.us-ashburn-1.oraclecloud.com"`

namespace

- **Purpose:** Specifies the OCI Object Storage service namespace. This is an optional parameter. If you don't specify this parameter, the default namespace of the tenancy is used.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"namespace" : "my-namespace"`

bucket

- **Purpose:** Specifies the bucket name to use for storing the migrated data. Ensure that the required bucket already exists in the OCI Object Storage instance and has write permissions.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"bucket" : "staging_bucket"`

prefix

- **Purpose:** Specifies the prefix that is added to the object name when objects are created in the bucket. The prefix acts as a logical container or directory for storing data. For more information about prefixes, see Object Naming Using Prefixes and Hierarchies.

  If not specified, the table name from the source is used as the prefix. If any object with the same name already exists in the bucket, it is overwritten.

  Schema is migrated to the `<prefix>/Schema /schema.ddl` file and source data is migrated to the `<prefix>/Data/<chunk>.json` file(s), where chunk=000000.json, 000001.json, and so forth.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"prefix" : "my_export"`
  2. `"prefix" : "my_export/2021-04-05/"`

chunkSize

- **Purpose:** Specifies the maximum size of a "chunk" of table data to be stored at the sink. During migration, a table is split into chunkSize chunks and each chunk is written as a separate file to the sink. When the source data being migrated exceeds this size, a new file is created.

  If not specified, defaults to 32MB. The valid value is an integer between 1 to 1024.
- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"chunkSize" : 40`

pretty

- **Purpose:** Specifies whether to beautify the JSON output to increase readability or not.

  If not specified, it defaults to false.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"pretty" : true`

credentials

- **Purpose:** Absolute path to a file containing OCI credentials.

  If not specified, it defaults to `$HOME/.oci/config`

  See Example Configuration for an example of the credentials file.

  > **Note:**
  >
  > Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**

  1. `"credentials" : "/home/user/.oci/config"`
  2. `"credentials" : "/home/user/security/config"`

credentialsProfile

- **Purpose:** Name of the configuration profile to be used to connect to the Oracle NoSQL Database Cloud Service. User account credentials are referred to as a 'profile'.

  If you do not specify this value, it defaults to the `DEFAULT` profile.

  > **Note:**
  >
  > This parameter is valid ONLY if the `credentials` parameter is specified.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**

  1. `"credentialsProfile" : "DEFAULT"`

2. `"credentialsProfile": "ADMIN_USER"`

useInstancePrincipal

- **Purpose:** Specifies whether or not the NoSQL Migrator tool uses instance principal authentication to connect to Oracle NoSQL Database Cloud Service. For more information on Instance Principal authentication method, see Source and Sink Security.

  If not specified, it defaults to false.

  > **Note:**
  >
  > – It is supported ONLY when the NoSQL Database Migrator tool is running within an OCI compute instance, for example, NoSQL Database Migrator tool running in a VM hosted on OCI.
  >
  > – Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useInstancePrincipal" : true`

Parquet File in OCI Object Storage Bucket

The configuration file format for Parquet file in OCI Object Storage bucket as a sink of NoSQL Database Migrator is shown below.

> **Note:**
>
> The valid source types for OCI Object Storage source type are `nosqldb` and `nosqldb_cloud`.

Sink Configuration Template

```
"sink" : {
  "type" : "object_storage_oci",
  "format" : "parquet",
  "endpoint" : "<OCI Object Storage service endpoint URL or region ID>",
  "namespace" : "<OCI Object Storage namespace>",
  "bucket" : "<bucket name>",
  "prefix" : "<object prefix>",
  "chunkSize" : <size in MB>,
  "compression": "<SNAPPY|GZIP|NONE>",
  "parquetOptions": {
    "useLogicalJson": <true|false>,
    "useLogicalEnum": <true|false>,
    "useLogicalUUID": <true|false>,
    "truncateDoubleSpecials": <true|false>
```

```
  },
  "credentials" : "</path/to/oci/config/file>",
  "credentialsProfile" : "<profile name in oci config file>",
  "useInstancePrincipal" : <true|false>
}
```

Source Parameters

- type
- format
- endpoint
- namespace
- bucket
- prefix
- chunkSize
- compression
- parquetOptions
- parquetOptions.useLogicalJson
- parquetOptions.useLogicalEnum
- parquetOptions.useLogicalUUID
- parquetOptions.truncateDoubleSpecials
- credentials
- credentialsProfile
- useInstancePrincipal

type

- **Purpose:** Identifies the sink type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "object_storage_oci"`

format

- **Purpose:** Specifies the sink format.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"format" : "parquet"`

endpoint

- **Purpose:** Specifies the OCI Object Storage service endpoint URL or region ID.

  You can either specify the complete URL or the Region ID alone. See Data Regions and Associated Service URLs in *Using Oracle NoSQL Database Cloud*

Service for the list of data regions supported for Oracle NoSQL Database Cloud Service.

- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Region ID: `"endpoint" : "us-ashburn-1"`
  - URL format: `"endpoint" : "https://objectstorage.us-ashburn-1.oraclecloud.com"`

namespace

- **Purpose:** Specifies the OCI Object Storage service namespace. This is an optional parameter. If you don't specify this parameter, the default namespace of the tenancy is used.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"namespace" : "my-namespace"`

bucket

- **Purpose:** Specifies the bucket name to use for storing the migrated data. Ensure that the required bucket already exists in the OCI Object Storage instance and has write permissions.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"bucket" : "staging_bucket"`

prefix

- **Purpose:** Specifies the prefix that is added to the object name when objects are created in the bucket. The prefix acts as a logical container or directory for storing data. For more information about prefixes, see Object Naming Using Prefixes and Hierarchies.

  If not specified, the table name from the source is used as the prefix. If any object with the same name already exists in the bucket, it is overwritten.

  Source data is migrated to the `<prefix>/Data/<chunk>.parquet` file(s), where chunk=000000.parquet, 000001.parquet, and so forth.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"prefix" : "my_export"`
  2. `"prefix" : "my_export/2021-04-05/"`

chunkSize

- **Purpose:** Specifies the maximum size of a "chunk" of table data to be stored at the sink. During migration, a table is split into chunkSize chunks and each chunk is written as a separate file to the sink. When the source data being migrated exceeds this size, a new file is created.

If not specified, defaults to 32MB. The valid value is an integer between 1 to 1024.

- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"chunkSize" : 40`

compression

- **Purpose:** Specifies the compression type to use to compress the Parquet data. Valid values are `SNAPPY`, `GZIP`, and `NONE`.

  If not specified, it defaults to `SNAPPY`.
- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:** `"compression" : "GZIP"`

parquetOptions

- **Purpose:** Species the options to select Parquet logical types for NoSQL ENUM, JSON, and UUID columns.

  If you do not specify this parameter, the NoSQL Database Migrator writes the data of ENUM, JSON, and UUID columns as String.
- **Data Type:** object
- **Mandatory (Y/N):** N

parquetOptions.useLogicalJson

- **Purpose:** Specifies whether or not to write NoSQL JSON column data as Parquet logical JSON type. For more information see Parquet Logical Type Definitions.

  If not specified or set to false, NoSQL Database Migrator writes the NoSQL JSON column data as String.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useLogicalJson" : true`

parquetOptions.useLogicalEnum

- **Purpose:** Specifies whether or not to write NoSQL ENUM column data as Parquet logical ENUM type. For more information see Parquet Logical Type Definitions.

  If not specified or set to false, NoSQL Database Migrator writes the NoSQL ENUM column data as String.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useLogicalEnum" : true`

parquetOptions.useLogicalUUID

- **Purpose:** Specifies whether or not to write NoSQL UUID column data as Parquet logical UUID type. For more information see Parquet Logical Type Definitions.

If not specified or set to false, NoSQL Database Migrator writes the NoSQL UUID column data as String.

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useLogicalUUID" : true`

parquetOptions.truncateDoubleSpecials

- **Purpose:** Specifies whether or not to truncate the double +Infinity, -Infinity, and Nan values.

  By default, it is set to `false`. If set to `true`,

  – Positive_Infinity is truncated to Double.MAX_VALUE.
  – NEGATIVE_INFINITY is truncated to -Double.MAX_VALUE.
  – NaN is truncated to 9.9999999999999990E307.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"truncateDoubleSpecials" : true`

credentials

- **Purpose:** Absolute path to a file containing OCI credentials.

  If not specified, it defaults to `$HOME/.oci/config`

  See Example Configuration for an example of the credentials file.

> **Note:**
>
> Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**

  1. `"credentials" : "/home/user/.oci/config"`
  2. `"credentials" : "/home/user/security/config"`

credentialsProfile

- **Purpose:** Name of the configuration profile to be used to connect to the Oracle NoSQL Database Cloud Service. User account credentials are referred to as a 'profile'.

  If you do not specify this value, it defaults to the `DEFAULT` profile.

> **Note:**
>
> This parameter is valid ONLY if the `credentials` parameter is specified.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"credentialsProfile" : "DEFAULT"`
  2. `"credentialsProfile": "ADMIN_USER"`

useInstancePrincipal

- **Purpose:** Specifies whether or not the NoSQL Migrator tool uses instance principal authentication to connect to Oracle NoSQL Database Cloud Service. For more information on Instance Principal authentication method, see Source and Sink Security.

  If not specified, it defaults to false.

> **Note:**
>
> – It is supported ONLY when the NoSQL Database Migrator tool is running within an OCI compute instance, for example, NoSQL Database Migrator tool running in a VM hosted on OCI.
>
> – Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"useInstancePrincipal" : true`

Oracle NoSQL Database

The configuration file format for Oracle NoSQL Database as a sink of NoSQL Database Migrator is shown below.

Sink Configuration Template

```
"sink" : {
  "type": "nosqldb",
  "table" : "<fully qualified table name>",
  "schemaInfo" : {
    "schemaPath" : "</path/to/a/schema/file>",
    "defaultSchema" : <true|false>,
```

```
    "useSourceSchema" : <true|false>,
    "DDBPartitionKey" : <"name:type">,
    "DDBSortKey" : "<name:type>"
  },
  "overwrite" : <true|false>,
  "storeName" : "<store name>",
  "helperHosts" : ["hostname1:port1","hostname2:port2,..."],
  "security" : "</path/to/store/credentials/file>",
  "requestTimeoutMs" : <timeout in milli seconds>,
  "includeTTL": <true|false>,
  "ttlRelativeDate": "<date-to-use in UTC>"
}
```

Sink Parameters

- type
- table
- schemaInfo
- schemaInfo.schemaPath
- schemaInfo.defaultSchema
- schemaInfo.useSourceSchema
- schemaInfo.DDBPartitionKey
- schemaInfo.DDBSortKey
- overwrite
- storeName
- helperHosts
- security
- requestTimeoutMs
- includeTTL
- ttlRelativeDate

type

- **Purpose:** Identifies the sink type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "nosqldb"`

table

- **Purpose:** Fully qualified table name from which to migrate the data.

  Format: `[namespace_name:]<table_name>`

  If the table is in the DEFAULT namespace, you can omit the `namespace_name`. The table must exist in the store during the migration, and its schema must match with the source data.

If the table is not available in the sink, you can use the `schemaInfo` parameter to instruct the NoSQL Database Migrator to create the table also in the sink.

- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - With the DEFAULT namespace `"table" :"mytable"`
  - With a non-default namespace `"table" : "mynamespace:mytable"`
  - To specify a child table `"table" : "mytable.child"`

> **Note:**
>
> You can migrate the child tables from a valid data source to Oracle NoSQL Database. The NoSQL Database Migrator copies only a single table in each execution. Ensure that the parent table is migrated before the child table.

schemainfo

- **Purpose:** Specifies the schema for the data being migrated. If this is not specified, the assumes that the table already exists in the sink's store.
- **Data Type:** Object
- **Mandatory (Y/N):** N

schemaInfo.schemaPath

- **Purpose:** Specifies the absolute path to a file containing DDL statements for the NoSQL table.

  The NoSQL Database Migrator executes the DDL commands listed in this file before migrating the data.

  The NoSQL Database Migrator does not support more than one DDL statement per line in the `schemaPath` file.
- **Data Type:** string
- **Mandatory:** Y, only when the `schemaInfo.defaultSchema` parameter is set to No.

schemaInfo.defaultSchema

- **Purpose:** Setting this parameter to *true* instructs the NoSQL Database Migrator to create a table with default schema. The default schema is defined by the migrator itself. For more information about default schema definitions, see *Default Schema* in Using Oracle NoSQL Data Migrator.
- **Data Type:** boolean
- **Mandatory:** N

> **Note:**
>
> `defaultSchema` and `schemaPath` are mutually exclusive

- **Example:**
  - With Default Schema:

    ```
    "schemaInfo" : {
          "defaultSchema" : true
        }
    ```

  - With a pre-defined schema:

    ```
    "schemaInfo" : {
          "schemaPath" : "<complete/path/to/the/schema/definition/file>"
        }
    ```

schemaInfo.useSourceSchema

- **Purpose:** Specifies whether or not the sink uses the table schema definition provided by the source when migrating NoSQL tables.
- **Data Type:** boolean
- **Mandatory (Y/N):** N

> **Note:**
>
> `defaultSchema`, `schemaPath`, and `useSourceSchema` parameters are mutually exclusive. Specify ONLY one of these parameters.

- **Example:**
  - With Default Schema:

    ```
    "schemaInfo" : {
          "defaultSchema" : true
        }
    ```

  - With a pre-defined schema:

    ```
    "schemaInfo" : {
          "schemaPath" : "<complete/path/to/the/schema/definition/file>"
        }
    ```

  - With source schema:

    ```
    "schemaInfo" : {
          "useSourceSchema" : true
        }
    ```

schemaInfo.DDBPartitionKey

- **Purpose:** Specifies the DynamoDB partition key and the corresponding Oracle NoSQL Database type to be used in the sink Oracle NoSQL Database table. This key will be used as a NoSQL DB table shard key. This is applicable only when `defaultSchema` is set to true and the source format is `dynamodb_json`. See Mapping of DynamoDB types to Oracle NoSQL types for more details.
- **Mandatory:** Yes if `defaultSchema` is true and the source is dynamodb_json.
- **Example:** `"DDBPartitionKey" : "PersonID:INTEGER"`

> **Note:**
>
> If the partition key contains dash(-) or dot(.), Migrator will replace it with underscore(_) as NoSQL column name does not support dot and dash.

schemaInfo.DDBSortKey

- **Purpose:** Specifies the DynamoDB sort key and its corresponding Oracle NoSQL Database type to be used in the target Oracle NoSQL Database table. If the importing DynamoDB table does not have a sort key this should not be set. This key will be used as a non-shard portion of the primary key in the NoSQL DB table. This is applicable only when `defaultSchema` is set to true and the source is `dynamodb_json`. See Mapping of DynamoDB types to Oracle NoSQL types for more details.
- **Mandatory:** No
- **Example:**`"DDBSortKey" : "Skey:STRING"`

> **Note:**
>
> If the sort key contains dash(-) or dot(.), Migrator will replace it with underscore(_) as NoSQL column name does not support dot and dash.

overwrite

- **Purpose:** Indicates the behavior of NoSQL Database Migrator when the record being migrated from the source is already present in the sink.

  If the value is set to false, when migrating tables the NoSQL Database Migrator skips those records for which the same primary key already exists in the sink.

  If the value is set to true, when migrating tables the NoSQL Database Migrator overwrites those records for which the same primary key already exists in the sink.

  If not specified, it defaults to true.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"overwrite" : false`

storeName

- **Purpose:** Name of the Oracle NoSQL Database store.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"storeName" : "kvstore"`

helperHosts

- **Purpose:** A list of host and registry port pairs in the `hostname:port` format. Delimit each item in the list using a comma. You must specify at least one helper host.
- **Data Type:** array of strings
- **Mandatory (Y/N):** Y
- **Example:** `"helperHosts" : ["localhost:5000","localhost:6000"]`

security

- **Purpose:**

  If your store is a secure store, provide the absolute path to the security login file that contains your store credentials. See Configuring Security with Remote Access in *Administrator's Guide* to know more about the security login file.

  You can use either password file based authentication or wallet based authentication. However, the wallet based authentication is supported only in the Enterprise Edition (EE) of Oracle NoSQL Database. For more information on wallet-based authentication, see Source and Sink Security.
- **Data Type:** string
- **Mandatory (Y/N):** Y for a secure store
- **Example:**
  `"security" : "/home/user/client.credentials"`

  Example security file content for password file based authentication:

  ```
  oracle.kv.password.noPrompt=true
  oracle.kv.auth.username=admin
  oracle.kv.auth.pwdfile.file=/home/nosql/login.passwd
  oracle.kv.transport=ssl
  oracle.kv.ssl.trustStore=/home/nosql/client.trust
  oracle.kv.ssl.protocols=TLSv1.2
  oracle.kv.ssl.hostnameVerifier=dnmatch(CN\=NoSQL)
  ```

  Example security file content for wallet based authentication:

  ```
  oracle.kv.password.noPrompt=true
  oracle.kv.auth.username=admin
  oracle.kv.auth.wallet.dir=/home/nosql/login.wallet
  oracle.kv.transport=ssl
  oracle.kv.ssl.trustStore=/home/nosql/client.trust
  oracle.kv.ssl.protocols=TLSv1.2
  oracle.kv.ssl.hostnameVerifier=dnmatch(CN\=NoSQL)
  ```

requestTimeoutMs

- **Purpose:** Specifies the time to wait for each write operation in the sink to complete. This is provided in milliseconds. The default value is 5000. The value can be any positive integer.
- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"requestTimeoutMs" : 5000`

includeTTL

- **Purpose:** Specifies whether or not to include TTL metadata for table rows provided by the source when importing Oracle NoSQL Database tables.

  If you do not specify this parameter, it defaults to `false`. In that case, the NoSQL Database Migrator does not include TTL metadata for table rows provided by the source when importing Oracle NoSQL Database tables.

  If set to true, the NoSQL Database Migrator tool performs the following checks on the TTL metadata when importing a table row:

  – If you import a row that does not have `_metadata` definition, the NoSQL Database Migrator tool sets the TTL to 0, which means the row never expires.

  – If you import a row that has `_metadata` definition, the NoSQL Database Migrator tool compares the TTL value against a Reference Time when a row gets imported. If the row has already expired relative to the Reference Time, then it is skipped. If the row has not expired, then it is imported along with the TTL value. By default, the Reference Time of import operation is the current time in milliseconds, obtained from System.currentTimeMillis(), of the machine where the NoSQL Database Migrator tool is running. But you can also set a custom Reference Time using the `ttlRelativeDate` configuration parameter if you want to extend the expiration time and import rows that would otherwise expire immediately.
    The formula to calculate the expiration time of a row is as follows:

    ```
    expiration = (TTL value of source row in milliseconds -
    Reference Time in milliseconds)
    if (expiration <= 0) then it indicates that row has expired.
    ```

    > **Note:**
    >
    > Since Oracle NoSQL TTL boundaries are in hours and days, in some cases, the TTL of the imported row might get adjusted to the nearest hour or day. For example, consider a row that has expiration value of `1629709200000 (2021-08-23 09:00:00)` and Reference Time value is `1629707962582 (2021-08-23 08:39:22)`. Here, even though the row is not expired relative to the Reference Time when this data gets imported, the new TTL for the row is `1629712800000 (2021-08-23 10:00:00)`.

- **Data Type:** boolean

- **Mandatory (Y/N):** N
- **Example:** `"includeTTL" : true`

ttlRelativeDate

- **Purpose:** Specify a UTC date in the YYYY-MM-DD hh:mm:ss format used to set the TTL expiry of table rows during importing into the Oracle NoSQL Database.

  If a table row in the data you are exporting has expired, you can set the `ttlRelativeDate` parameter to a date before the expiration time of the table row in the exported data.

  If you do not specify this parameter, it defaults to the current time in milliseconds, obtained from System.currentTimeMillis(), of the machine where the NoSQL Database Migrator tool is running.
- **Data Type:** date
- **Mandatory (Y/N):** N
- **Example:** `"ttlRelativeDate" : "2021-01-03 04:31:17"`
  Let us consider a scenario where table rows expire after seven days from 1-Jan-2021. After exporting this table, on 7-Jan-2021, you run into an issue with your table and decide to import the data. The table rows are going to expire in one day (data expiration date minus the default value of `ttlRelativedate` configuration parameter, which is the current date).But if you want to extend the expiration date of table rows to five days instead of one day, use the `ttlRelativeDate` parameter and choose an earlier date. Therefore, in this scenario if you want to extend expiration time of the table rows by five days, set the value of `ttlRelativeDate` configuration parameters to 3-Jan-2021, which is used as Reference Time when table rows get imported.

Oracle NoSQL Database Cloud Service

The configuration file format for Oracle NoSQL Database Cloud Service as a sink of NoSQL Database Migrator is shown below.

Sink Configuration Template

```
"sink" : {
    "type" : "nosqldb_cloud",
    "endpoint" : "<Oracle NoSQL Cloud Service Endpoint>",
    "table" : "<table name>",
    "compartment" : "<OCI compartment name or id>",
    "schemaInfo" : {
      "schemaPath" : "</path/to/a/schema/file>",
      "defaultSchema" : <true|false>,
      "useSourceSchema" : <true|false>,
      "DDBPartitionKey" : <"name:type">,
      "DDBSortKey" : "<name:type>",
      "onDemandThroughput" : <true|false>,
      "readUnits" : <table read units>,
      "writeUnits" : <table write units>,
      "storageSize" : <storage size in GB>
    },
    "credentials" : "</path/to/oci/credential/file>",
    "credentialsProfile" : "<oci credentials profile name>",
    "writeUnitsPercent" : <table writeunits percent>,
```

```
    "requestTimeoutMs" : <timeout in milli seconds>,
    "useInstancePrincipal" : <true|false>,
    "overwrite" : <true|false>,
    "includeTTL": <true|false>,
    "ttlRelativeDate" : "<date-to-use in UTC>"
}
```

Sink Parameters

- type
- endpoint
- table
- compartment
- schemaInfo
- schemaInfo.schemaPath
- schemaInfo.defaultSchema
- schemaInfo.useSourceSchema
- schemaInfo.DDBPartitionKey
- schemaInfo.DDBSortKey
- schemaInfo.onDemandThroughput
- schemaInfo.readUnits
- schemaInfo.writeUnits
- schemaInfo.storageSize
- credentials
- credentialsProfile
- writeUnitsPercent
- requestTimeoutMs
- useInstancePrincipal
- overwrite
- includeTTL
- ttlRelativeDate

type

- **Purpose:** Identifies the sink type.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** `"type" : "nosqldb_cloud"`

endpoint

- **Purpose:** Specifies the Service Endpoint of the Oracle NoSQL Database Cloud Service.

You can either specify the complete URL or the Region ID alone. See Data Regions and Associated Service URLs in *Using Oracle NoSQL Database Cloud Service* for the list of data regions supported for Oracle NoSQL Database Cloud Service.

- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - Region ID: `"endpoint" : "us-ashburn-1"`
  - URL format: `"endpoint" : "https://nosql.us-ashburn-1.oci.oraclecloud.com/"`

table

- **Purpose:** Name of the table to which to migrate the data.

  You must ensure that this table exists in your Oracle NoSQL Database Cloud Service. Otherwise, you have to use the `schemaInfo` object in the sink configuration to instruct the NoSQL Database Migrator to create the table.

  The schema of this table must match the source data.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:**
  - To specify a table `"table" : "mytable"`
  - To specify a child table `"table" : "mytable.child"`

> **Note:**
>
> You can migrate the child tables from a valid data source to Oracle NoSQL Database Cloud Service. The NoSQL Database Migrator copies only a single table in each execution. Ensure that the parent table is migrated before the child table.

compartment

- **Purpose:** Specifies the name or OCID of the compartment in which the table resides.

  If you do not provide any value, it defaults to the *root* compartment.

  You can find your compartment's OCID from the Compartment Explorer window under Governance in the OCI Cloud Console.
- **Data Type:** string
- **Mandatory (Y/N):** Y if the table is not in the root compartment of the tenancy OR when the `useInstancePrincipal` parameter is set to true.

> **Note:**
>
> If the `useInstancePrincipal` parameter is set to true, the compartment must specify the compartment OCID and not the name.

- **Example:**
  - Compartment name

    ```
    "compartment" : "mycompartment"
    ```
  - Compartment name qualified with its parent compartment

    ```
    "compartment" : "parent.childcompartment"
    ```
  - No value provided. Defaults to the root compartment.

    ```
    "compartment": ""
    ```
  - Compartment OCID

    ```
    "compartment" : "ocid1.tenancy.oc1...4ksd"
    ```

schemaInfo

- **Purpose:** Specifies the schema for the data being migrated.

  If you do not specify this parameter, the NoSQL Database Migrator assumes that the table already exists in your Oracle NoSQL Database Cloud Service.

  If this parameter is not specified and the table does not exist in the sink, the migration fails.
- **Data Type:** Object
- **Mandatory (Y/N):** N

schemaInfo.schemaPath

- **Purpose:** Specifies the absolute path to a file containing DDL statements for the NoSQL table.

  The NoSQL Database Migrator executes the DDL commands listed in this file before migrating the data.

  The NoSQL Database Migrator does not support more than one DDL statement per line in the `schemaPath` file.
- **Data Type:** string
- **Mandatory:** Y, only when the `schemaInfo.defaultSchema` parameter is set to No.

schemaInfo.defaultSchema

- **Purpose:** Setting this parameter to *Yes* instructs the NoSQL Database Migrator to create a table with default schema. The default schema is defined by the migrator itself. For more information about default schema definitions, see *Default Schema* in Using Oracle NoSQL Data Migrator.
- **Data Type:** boolean
- **Mandatory:** Y, only when the `schemaInfo.defaultSchema` parameter is set to No.

> **Note:**
>
> `defaultSchema` and `schemaPath` are mutually exclusive

schemaInfo.useSourceSchema

- **Purpose:** Specifies whether or not the sink uses the table schema definition provided by the source when migrating NoSQL tables.
- **Data Type:** boolean
- **Mandatory (Y/N):** N

> **Note:**
>
> `defaultSchema`, `schemaPath`, and `useSourceSchema` parameters are mutually exclusive. Specify ONLY one of these parameters.

- **Example:**
  - With Default Schema:

```
"schemaInfo" : {
      "defaultSchema" : true,
      "readUnits" : 100,
      "writeUnits" : 60,
      "storageSize" : 1
    }
```

  - With a pre-defined schema:

```
"schemaInfo" : {
      "schemaPath" : "<complete/path/to/the/schema/definition/file>",
      "readUnits" : 100,
      "writeUnits" : 100,
      "storageSize" : 1
    }
```

  - With source schema:

```
"schemaInfo" : {
      "useSourceSchema" : true,
      "readUnits" : 100,
      "writeUnits" : 60,
      "storageSize" : 1
    }
```

schemaInfo.DDBPartitionKey

- **Purpose:** Specifies the DynamoDB partition key and the corresponding Oracle NoSQL Database type to be used in the sink Oracle NoSQL Database table. This key will be used as a NoSQL DB table shard key. This is applicable only when `defaultSchema` is set to true and the source format is `dynamodb_json`. See Mapping of DynamoDB types to Oracle NoSQL types for more details.
- **Mandatory:** Yes if `defaultSchema` is true and the source is dynamodb_json.
- **Example:** `"DDBPartitionKey" : "PersonID:INTEGER"`

> **Note:**
>
> If the partition key contains dash(-) or dot(.), Migrator will replace it with underscore(_) as NoSQL column name does not support dot and dash.

schemaInfo.DDBSortKey

- **Purpose:** Specifies the DynamoDB sort key and its corresponding Oracle NoSQL Database type to be used in the target Oracle NoSQL Database table. If the importing DynamoDB table does not have a sort key this should not be set. This key will be used as a non-shard portion of the primary key in the NoSQL DB table. This is applicable only when `defaultSchema` is set to true and the source is `dynamodb_json`.See Mapping of DynamoDB types to Oracle NoSQL types for more details.
- **Mandatory:** No
- **Example:**`"DDBSortKey" : "Skey:STRING"`

> **Note:**
>
> If the sort key contains dash(-) or dot(.), Migrator will replace it with underscore(_) as NoSQL column name does not support dot and dash.

schemaInfo.onDemandThroughput

- **Purpose:** Specifies to create the table with on-demand read and write throughput. If this parameter is not set, the table is created with provisioned capacity. The default value is `false`.

> **Note:**
>
> This parameter is not applicable for child tables as they share the throughput of the top-level parent table.

- **Data Type:** Boolean
- **Mandatory:** N
- **Example:**`"onDemandThroughput" : "true"`

schemaInfo.readUnits

- **Purpose:** Specifies the read throughput of the new table.

> **Note:**
>
> – This parameter is not applicable for tables provisioned with on-demand capacity.
>
> – This parameter is not applicable for child tables as they share the read throughput of the top-level parent table.

- **Data Type:** integer
- **Mandatory:** Y when the table is not a child table or when `schemaInfo.onDemandThroughput` parameter is set to `false`, else N.
- **Example:**`"readUnits" : 100`

schemaInfo.writeUnits

- **Purpose:** Specifies the write throughput of the new table.

> **Note:**
>
> – This parameter is not applicable for tables provisioned with on-demand capacity.
>
> – This parameter is not applicable for child tables as they share the write throughput of the top-level parent table.

- **Data Type:** integer
- **Mandatory:** Y when the table is not a child table or when `schemaInfo.onDemandThroughput` parameter is set to `false`, else N.
- **Example:**`"writeUnits" : 100`

schemaInfo.storageSize

- **Purpose:** Specifies the storage size of the new table in GB

> **Note:**
>
> This parameter is not applicable for child tables as they share the storage size of the top-level parent table.

- **Data Type:** integer
- **Mandatory:** Y when the table is not a child table, else N.
- **Example:**
  - With `schemaPath`

```
"schemaInfo" : {
  "schemaPath" : "</path/to/a/schema/file>",
  "readUnits" : 500,
```

```
  "writeUnits" : 1000,
  "storageSize" : 5 }
```

– With `defaultSchema`

```
"schemaInfo" : {
  "defaultSchema" :Yes,
  "readUnits" : 500,
  "writeUnits" : 1000,
  "storageSize" : 5
}
```

credentials

- **Purpose:** Absolute path to a file containing OCI credentials.

  If not specified, it defaults to `$HOME/.oci/config`

  See Example Configuration for an example of the credentials file.

  > **Note:**
  >
  > Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"credentials" : "/home/user/.oci/config"`
  2. `"credentials" : "/home/user/security/config"`

credentialsProfile

- **Purpose:** Name of the configuration profile to be used to connect to the Oracle NoSQL Database Cloud Service.

  If you do not specify this value, it defaults to the `DEFAULT` profile.

  > **Note:**
  >
  > This parameter is valid ONLY if the `credentials` parameter is specified.

- **Data Type:** string
- **Mandatory (Y/N):** N
- **Example:**
  1. `"credentialsProfile" : "DEFAULT"`

2. `"credentialsProfile": "ADMIN_USER"`

writeUnitsPercent

- **Purpose:** Specifies the Percentage of table write units to be used during the migration activity.

  The default value is 90. The valid range is any integer between 1 to 100.

  > **Note:**
  >
  > The time required for the data migration is directly proportional to the `writeUnitsPercent` value.

  See Troubleshooting the Oracle NoSQL Database Migrator to learn how to use this attribute to improve the data migration speed.
- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"writeUnitsPercent" : 90`

requestTimeoutMs

- **Purpose:** Specifies the time to wait for each write operation in the sink to complete. This is provided in milliseconds. The default value is 5000. The value can be any positive integer.
- **Data Type:** integer
- **Mandatory (Y/N):** N
- **Example:** `"requestTimeoutMs" : 5000`

useInstancePrincipal

- **Purpose:** Specifies whether or not the NoSQL Migrator tool uses instance principal authentication to connect to Oracle NoSQL Database Cloud Service. For more information on Instance Principal authentication method, see Source and Sink Security

  If not specified, it defaults to false.

  > **Note:**
  >
  > – It is supported ONLY when the NoSQL Database Migrator tool is running within an OCI compute instance, for example, NoSQL Database Migrator tool running in a VM hosted on OCI.
  >
  > – Even though the `credentials` and `useInstancePrincipal` parameters are not mandatory individually, one of these parameters MUST be specified. Additionally, these two parameters are mutually exclusive. Specify ONLY one of these parameters, but not both at the same time.
- **Data Type:** boolean

- **Mandatory (Y/N):** N
- **Example:** `"useInstancePrincipal" : true`

overwrite

- **Purpose:** Indicates the behavior of NoSQL Database Migrator when the record being migrated from the source is already present in the sink.

  If the value is set to false, when migrating tables the NoSQL Database Migrator skips those records for which the same primary key already exists in the sink.

  If the value is set to true, when migrating tables the NoSQL Database Migrator overwrites those records for which the same primary key already exists in the sink.

  If not specified, it defaults to true.
- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"overwrite" : false`

includeTTL

- **Purpose:** Specifies whether or not to include TTL metadata for table rows provided by the source when importing Oracle NoSQL Database tables.

  If you do not specify this parameter, it defaults to `false`. In that case, the NoSQL Database Migrator does not include TTL metadata for table rows provided by the source when importing Oracle NoSQL Database tables.

  If set to true, the NoSQL Database Migrator tool performs the following checks on the TTL metadata when importing a table row:

  – If you import a row that does not have `_metadata` definition, the NoSQL Database Migrator tool sets the TTL to 0, which means the row never expires.

  – If you import a row that has `_metadata` definition, the NoSQL Database Migrator tool compares the TTL value against a Reference Time when a row gets imported. If the row has already expired relative to the Reference Time, then it is skipped. If the row has not expired, then it is imported along with the TTL value. By default, the Reference Time of import operation is the current time in milliseconds, obtained from System.currentTimeMillis(), of the machine where the NoSQL Database Migrator tool is running. But you can also set a custom Reference Time using the `ttlRelativeDate` configuration parameter if you want to extend the expiration time and import rows that would otherwise expire immediately.
    The formula to calculate the expiration time of a row is as follows:

    ```
    expiration = (TTL value of source row in milliseconds -
    Reference Time in milliseconds)
    if (expiration <= 0) then it indicates that row has expired.
    ```

> **Note:**
>
> Since Oracle NoSQL TTL boundaries are in hours and days, in some cases, the TTL of the imported row might get adjusted to the nearest hour or day. For example, consider a row that has expiration value of `1629709200000 (2021-08-23 09:00:00)` and Reference Time value is `1629707962582 (2021-08-23 08:39:22)`. Here, even though the row is not expired relative to the Reference Time when this data gets imported, the new TTL for the row is `1629712800000 (2021-08-23 10:00:00)`.

- **Data Type:** boolean
- **Mandatory (Y/N):** N
- **Example:** `"includeTTL" : true`

ttlRelativeDate

- **Purpose:** Specify a UTC date in the YYYY-MM-DD hh:mm:ss format used to set the TTL expiry of table rows during importing into the Oracle NoSQL Database.

  If a table row in the data you are exporting has expired, you can set the `ttlRelativeDate` parameter to a date before the expiration time of the table row in the exported data.

  If you do not specify this parameter, it defaults to the current time in milliseconds, obtained from System.currentTimeMillis(), of the machine where the NoSQL Database Migrator tool is running.
- **Data Type:** date
- **Mandatory (Y/N):** N
- **Example:** `"ttlRelativeDate" : "2021-01-03 04:31:17"`
  Let us consider a scenario where table rows expire after seven days from 1-Jan-2021. After exporting this table, on 7-Jan-2021, you run into an issue with your table and decide to import the data. The table rows are going to expire in one day (data expiration date minus the default value of `ttlRelativedate` configuration parameter, which is the current date).But if you want to extend the expiration date of table rows to five days instead of one day, use the `ttlRelativeDate` parameter and choose an earlier date. Therefore, in this scenario if you want to extend expiration time of the table rows by five days, set the value of `ttlRelativeDate` configuration parameters to 3-Jan-2021, which is used as Reference Time when table rows get imported.

Transformation Configuration Templates

This topic explains the configuration parameters for the different transformations supported by the Oracle NoSQL Database Migrator. For the complete configuration file template, see **Configuration File** in Terminology used with Oracle NoSQL Database Migrator.

Oracle NoSQL Database Migrator lets you modify the data, that is, add data transformations as part of the migration activity. You can define multiple transformations in a single migration. In such a case, the order of transformations is vital because the source data undergoes each transformation in the given order. The output of one transformation becomes the input to the next one in the migrator pipeline.

The different transformations supported by the NoSQL Data Migrator are:

Table 5-2 Transformations

Transformation Config Attribute	You can use this transformation to ...
`ignoreFields`	Ignore the identified columns from the source row before writing to the sink.
`includeFields`	Include the identified columns from the source row before writing to the sink.
`renameFields`	Rename the identified columns from the source row before writing to the sink.
`aggregateFields`	Aggregate multiple columns from the source into a single column in the sink. As part of this transformation, you can also identify the columns that you want to exclude in the aggregation. Those fields will be skipped from the aggregated column.

You can find the configuration template for each supported transformation below.

ignoreFields

The configuration file format for the `ignoreFields` transformation is shown below.

Transformation Configuration Template

```
"transforms" : {
  "ignoreFields" : ["<field1>","<field2>",...]
}
```

Transformation Parameter

ignoreFields

- **Purpose:** An array of the column names to be ignored from the source records.

  > **Note:**
  >
  > You can supply only top-level fields. Transformations can not be applied on the data in the nested fields.

- **Data Type:** array of strings
- **Mandatory (Y/N):** Y
- **Example:** To ignore the columns named "name" and "address" from the source record:

  ```
  "ignoreFields" : ["name","address"]
  ```

includeFields

The configuration file format for the `includeFields` transformation is shown below.

Transformation Configuration Template

```
"transforms" : {
  "includeFields" : ["<field1>","<field2>",...]
}
```

Transformation Parameter

includeFields

- **Purpose:** An array of the column names to be included from the source records. It *only* includes the fields specified in the array, the rest of the fields are ignored.

  > **Note:**
  >
  > The NoSQL Database Migrator tool throws an error if you specify an empty array. Additionally, you can specify only the top-level fields. The NoSQL Database Migrator tool does not apply transformations to the data in the nested fields.

- **Data Type:** array of strings
- **Mandatory (Y/N):** Y
- **Example:** To ignore the columns named "age" and "gender" from the source record:

  ```
  "includeFields" : ["age","gender"]
  ```

renameFields

The configuration file format for the `renameFields` transformation is shown below.

Transformation Configuration Template

```
"transforms" : {
  "renameFields" : {
    "<old_name>" : "<new_name>",
    "<old_name>" : "<new_name>,"
    .....
  }
}
```

Transformation Parameter

renameFields

- **Purpose:** Key-Value pairs of the old and new names of the columns to be renamed.

> **Note:**
>
> You can supply only top-level fields. Transformations can not be applied on the data in the nested fields.

- **Data Type:** JSON object
- **Mandatory (Y/N):** Y
- **Example:** To rename the column named "residence" to "address" and the column named "_id" to "id":

```
"renameFields" : { "residence" : "address", "_id" : "id" }
```

aggregateFields

The configuration file format for the `aggregateFields` transformation is shown below.

Transformation Configuration Template

```
"transforms" : {
  "aggregateFields" : {
    "fieldName" : "name of the new aggregate field",
    "skipFields" : ["<field1>","<field2">,...]
  }
}
```

Transformation Parameter

aggregateFields

- **Purpose:** Name of the aggregated field in the sink.
- **Data Type:** string
- **Mandatory (Y/N):** Y
- **Example:** If the given record is:

```
{
  "id" : 100,
  "name" : "john",
  "address" : "USA",
  "age" : 20
}
```

If the aggregate transformation is:

```
"aggregateFields" : {
  "fieldName" : "document",
  "skipFields" : ["id"]
}
```

The aggregated column in the sink looks like:

```
{
  "id": 100,
  "document": {
    "name": "john",
    "address": "USA",
    "age": 20
  }
}
```

Mapping of DynamoDB table to Oracle NoSQL table

In DynamoDB, a table is a collection of items, and each item is a collection of attributes. Each item in the table has a unique identifier, or a primary key. Other than the primary key, the table is schema-less. Each item can have its own distinct attributes.

DynamoDB supports two different kinds of primary keys:

- **Partition key** – A simple primary key, composed of one attribute known as the *partition key*. DynamoDB uses the partition key's value as input to an internal hash function. The output from the hash function determines the partition in which the item will be stored.
- **Partition key and sort key** – As a *composite primary key*, this type of key is composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*. DynamoDB uses the partition key value as input to an internal hash function. The output from the hash function determines the partition in which the item will be stored. All items with the same partition key value are stored together, in sorted order by sort key value.

In contrast, Oracle NoSQL tables support flexible data models with both schema and schema-less design.

There are two different ways of modelling a DynamoDB table:

1. **Modeling DynamoDB table as a JSON document(Recommended)**: In this modeling, you map all the attributes of the Dynamo DB tables into a JSON column of the NoSQL table except partition key and sort key. You will model partition key and sort key as the Primary Key columns of the NoSQL table. You will use `AggregateFields` transform in order to aggregate non-primary key data into a JSON column.

   > **Note:**
   >
   > The Migrator provides a user-friendly configuration `defaultSchema` to automatically create a schema-less DDL table which also aggregates attributes into a JSON column.

2. **Modeling DynamoDB table as fixed columns in NoSQL table:** In this modeling, for each attribute of the DynamoDB table, you will create a column in the NoSQL table as specified in the Mapping of DynamoDB types to Oracle NoSQL types. You will model partition key and sort key attributes as Primary key(s). This should be used only when you are certain that importing DynamoDB table schema is fixed and each item has values for the most of the attributes. If DynamoDB items do not have common attributes, this can result in lot of NoSQL columns with empty values.

> **Note:**
>
> We highly recommend using schema-less tables when migrating data from DynamoDB to Oracle NoSQL Database due to the nature of DynamoDB tables being schema-less. This is especially for large tables where the content of each record may not be uniform across the table.

Oracle NoSQL to Parquet Data Type Mapping

Describes the mapping of Oracle NoSQL data types to Parquet data types.

NoSQL Type	Parquet Type
BOOLEAN	BOOLEAN
INTEGER	INT32
LONG	INT64
FLOAT	DOUBLE
DOUBLE	DOUBLE
BINARY	BINARY
FIXED_BINARY	BINARY
STRING	BINARY(STRING)
ENUM	BINARY(STRING) or BINARY(ENUM), if the logical ENUM is configured
UUID	BINARY(STRING) or FIXED_BINARY(16), if the logical UUID is configured
TIMESTAMP(p)	INT64(TIMESTAMP(p))
NUMBER	DOUBLE

field_name ARRAY(T)	`group field_name(LIST) {`
	`  repeated group list {`
	`  required T element`
	`  }`
	`}`

field_name MAP(T)	`group field_name (MAP) {`
	`  repeated group key_value (MAP_KEY_VALUE) {`
	`  required binary key (STRING);`
	`  required T value;`
	`  }`
	`}`

NoSQL Type	Parquet Type

field_name RECORD(K T N , K T N , ...)
where:
K = Key name
T = Type
N = Nullable or not
	`group field_name {`
	`  ni == true ? optional Ti ki : required Ti ki`
	`}`

JSON	BINARY(STRING) or BINARY(JSON), if logical JSON is configured

> **Note:**
>
> When the NoSQL Number type is converted to Parquet Double type, there may be some loss of precision in case the value cannot be represented in Double. If the number is too big to represent as Double, it is converted to Double.NEGATIVE_INFINITY or Double.POSITIVE_INFINITY.

Mapping of DynamoDB types to Oracle NoSQL types

The table below shows the mapping of DynamoDB types to Oracle NoSQL types.

Table 5-3 Mapping DynamoDB type to Oracle NoSQL type

#	DynamoDB type	JSON type for NoSQL JSON column	Oracle NoSQL type
1	String (S)	JSON String	STRING
2	Number Type (N)	JSON Number	INTEGER/LONG/ FLOAT/DOUBLE/ NUMBER
3	Boolean (BOOL)	JSON Boolean	BOOLEAN
4	Binary type (B) - Byte buffer	BASE-64 encoded JSON String	BINARY
5	NULL	JSON null	NULL
6	String Set (SS)	JSON Array of Strings	ARRAY(STRING)
7	Number Set (NS)	JSON Array of Numbers	ARRAY(INTEGER/ LONG/FLOAT/DOUBLE/ NUMBER)
8	Binary Set (BS)	JSON Array of Base-64 encoded Strings	ARRAY(BINARY)
9	LIST (L)	Array of JSON	ARRAY(JSON)
10	MAP (M)	JSON Object	JSON
11	PARTITION KEY	NA	PRIMARY KEY and SHARD KEY
12	SORT KEY	NA	PRIMARY KEY
13	Attribute names with dash and dot	JSON field names with a underscore	Column names with underscore

Few additional points to consider while mapping DynamoDB types to Oracle NoSQL types:

- DynamoDB Supports only one data type for Numbers and can have up to 38 digits of precision,on contrast Oracle NoSQL supports many types to choose from based on the range and precision of the data.You can select the appropriate Number type that fits the range of your input data. If you are not sure of the nature of the data, NoSQL NUMBER type can be used.
- DynamoDB Supports only one data type for Numbers and can have up to 38 digits of precision,on contrast Oracle NoSQL supports many types to choose from based on the range and precision of the data.You can select the appropriate Number type that fits the range of your input data. If you are not sure of the nature of the data, NoSQL NUMBER type can be used.
- Partition key in DynamoDB has a limit of 2048 bytes but Oracle NoSQL Cloud Service has a limit of 64 bytes for the Primary key/Shard key.
- Sort key in DynamoDB has a limit of 1024 bytes but Oracle NoSQL Cloud Service has a limit of 64 bytes for the Primary key.
- Attribute names in DynamoDB can be 64KB long but Oracle NoSQL Cloud service column names have a limit of 64 characters.

Use Case Demonstrations

Learn how to perform data migration using the Oracle NoSQL Database Migrator for specific use cases. You can find detailed systematic instructions with code examples to perform migration in each of the use cases listed below.

Topics:

- Migrate from Oracle NoSQL Database Cloud Service to a JSON file
- Migrate from Oracle NoSQL Database On-Premise to Oracle NoSQL Database Cloud Service
- Migrate from JSON file source to Oracle NoSQL Database Cloud Service
- **Migrate from MongoDB JSON file to an Oracle NoSQL Database Cloud Service**
- **Migrate from DynamoDB JSON file in AWS S3 to an Oracle NoSQL Database Cloud Service**
- **Migrate from DynamoDB JSON file to Oracle NoSQL Database**
- Migrate from CSV file to Oracle NoSQL Database

Migrate from Oracle NoSQL Database Cloud Service to a JSON file

This example shows how to use the Oracle NoSQL Database Migrator to copy data and the schema definition of a NoSQL table from Oracle NoSQL Database Cloud Service (NDCS) to a JSON file.

Use Case

An organization decides to train a model using the Oracle NoSQL Database Cloud Service (NDCS) data to predict future behaviors and provide personalized recommendations. They can take a periodic copy of the NDCS tables' data to a JSON

file and apply it to the analytic engine to analyze and train the model. Doing this helps them separate the analytical queries from the low-latency critical paths.

Example

For the demonstration, let us look at how to migrate the data and schema definition of a NoSQL table called `myTable` from NDCS to a JSON file.

Prerequisites

- Identify the source and sink for the migration.
  - Source: Oracle NoSQL Database Cloud Service
  - Sink: JSON file
- Identify your OCI cloud credentials and capture them in the OCI config file. Save the config file in `/home/.oci/config`. See Acquiring Credentials in *Using Oracle NoSQL Database Cloud Service*.

```
[DEFAULT]
tenancy=ocid1.tenancy.oc1....
user=ocid1.user.oc1....
fingerprint= 43:d1:....
key_file=</fully/qualified/path/to/the/private/key/>
pass_phrase=<passphrase>
```

- Identify the region endpoint and compartment name for your Oracle NoSQL Database Cloud Service.
  - endpoint: `us-phoenix-1`
  - compartment: `developers`

Procedure

To migrate the data and schema definition of `myTable` from Oracle NoSQL Database Cloud Service to a JSON file:

1. Open the command prompt and navigate to the directory where you extracted the NoSQL Database Migrator utility.
2. To generate the configuration file using the NoSQL Database Migrator, run the `runMigrator` command without any runtime parameters.

```
[~/nosqlMigrator/nosql-migrator-1.0.0]$./runMigrator
```

3. As you did not provide the configuration file as a runtime parameter, the utility prompts if you want to generate the configuration now. Type **`y`**.

```
configuration file is not provided. Do you want to generate
configuration? (y/n) [n]: y

This command provides a walkthrough of creating a valid config for
Oracle NoSQL data migrator.
```

4. Based on the prompts from the utility, choose your options for the Source configuration.

```
Enter a location for your config [./migrator-config.json]: /home/
apothula/nosqlMigrator/NDCS2JSON
Select the source:
1) nosqldb
2) nosqldb_cloud
3) file
#? 2
Configuration for source type=nosqldb_cloud
Enter endpoint URL or region of the Oracle NoSQL Database Cloud: us-
phoenix-1
Enter table name: myTable
Enter compartment name or id of the source table []: developers
Enter path to the file containing OCI credentials [/home/
apothula/.oci/config]:
Enter the profile name in OCI credentials file [DEFAULT]:
Enter percentage of table read units to be used for migration
operation. (1-100) [90]:
Enter store operation timeout in milliseconds. (1-30000) [5000]:
```

5. Based on the prompts from the utility, choose your options for the Sink configuration.

```
Select the sink:
1) nosqldb
2) nosqldb_cloud
3) file
#? 3
Configuration for sink type=file
Enter path to a file to store JSON data: /home/apothula/
nosqlMigrator/myTableJSON
Would you like to store JSON in pretty format? (y/n) [n]: y
Would you like to migrate the table schema also? (y/n) [y]: y
Enter path to a file to store table schema: /home/apothula/
nosqlMigrator/myTableSchema
```

6. Based on the prompts from the utility, choose your options for the source data transformations. The default value is `n`.

```
Would you like to add transformations to source data? (y/n) [n]:
```

7. Enter your choice to determine whether to proceed with the migration in case any record fails to migrate.

```
Would you like to continue migration in case of any record/row is
failed to migrate?: (y/n) [n]:
```

8. The utility displays the generated configuration on the screen.

```
generated configuration is:
{
  "source": {
```

```
      "type": "nosqldb_cloud",
      "endpoint": "us-phoenix-1",
      "table": "myTable",
      "compartment": "developers",
      "credentials": "/home/apothula/.oci/config",
      "credentialsProfile": "DEFAULT",
      "readUnitsPercent": 90,
      "requestTimeoutMs": 5000
    },
    "sink": {
      "type": "file",
      "format": "json",
      "schemaPath": "/home/apothula/nosqlMigrator/myTableSchema",
      "pretty": true,
      "dataPath": "/home/apothula/nosqlMigrator/myTableJSON"
    },
    "abortOnError": true,
    "migratorVersion": "1.0.0"
  }
```

9. Finally, the utility prompts for your choice to decide whether to proceed with the migration with the generated configuration file or not. The default option is `y`.

> **Note:**
>
> If you select `n`, you can use the generated configuration file to run the migration using the `./runMigrator -c` or the `./runMigrator --config` option.

```
would you like to run the migration with above configuration?
If you select no, you can use the generated configuration file to run the
migration using
./runMigrator --config /home/apothula/nosqlMigrator/NDCS2JSON
(y/n) [y]:
```

10. The NoSQL Database Migrator migrates your data and schema from NDCS to the JSON file.

```
Records provided by source=10,Records written to sink=10,Records failed=0.
Elapsed time: 0min 1sec 277ms
Migration completed.
```

Validation

To validate the migration, you can open the JSON Sink files and view the schema and data.

```
-- Exported myTable Data

[~/nosqlMigrator]$cat myTableJSON
{
  "id" : 10,
  "document" : {
    "course" : "Computer Science",
    "name" : "Neena",
```

```
    "studentid" : 105
  }
}
{
  "id" : 3,
  "document" : {
  "course" : "Computer Science",
    "name" : "John",
    "studentid" : 107
  }
}
{
  "id" : 4,
  "document" : {
    "course" : "Computer Science",
    "name" : "Ruby",
    "studentid" : 100
  }
}
{
  "id" : 6,
  "document" : {
    "course" : "Bio-Technology",
    "name" : "Rekha",
    "studentid" : 104
  }
}
{
  "id" : 7,
  "document" : {
    "course" : "Computer Science",
    "name" : "Ruby",
    "studentid" : 100
  }
}
{
  "id" : 5,
  "document" : {
    "course" : "Journalism",
    "name" : "Rani",
    "studentid" : 106
  }
}
{
  "id" : 8,
  "document" : {
    "course" : "Computer Science",
    "name" : "Tom",
    "studentid" : 103
  }
}
{
  "id" : 9,
  "document" : {
    "course" : "Computer Science",
```

```
    "name" : "Peter",
    "studentid" : 109
  }
}
{
  "id" : 1,
  "document" : {
    "course" : "Journalism",
    "name" : "Tracy",
    "studentid" : 110
  }
}
{
  "id" : 2,
  "document" : {
    "course" : "Bio-Technology",
    "name" : "Raja",
    "studentid" : 108
  }
}


-- Exported myTable Schema

[~/nosqlMigrator]$cat myTableSchema
CREATE TABLE IF NOT EXISTS myTable (id INTEGER, document JSON, PRIMARY
KEY(SHARD(id)))
```

Migrate from Oracle NoSQL Database On-Premise to Oracle NoSQL Database Cloud Service

This example shows how to use the Oracle NoSQL Database Migrator to copy data and the schema definition of a NoSQL table from Oracle NoSQL Database to Oracle NoSQL Database Cloud Service (NDCS).

Use Case

As a developer, you are exploring options to avoid the overhead of managing the resources, clusters, and garbage collection for your existing NoSQL Database KVStore workloads. As a solution, you decide to migrate your existing on-premise KVStore workloads to Oracle NoSQL Database Cloud Service because NDCS manages them automatically.

Example

For the demonstration, let us look at how to migrate the data and schema definition of a NoSQL table called `myTable` from the NoSQL Database KVStore to NDCS. We will also use this use case to show how to run the `runMigrator` utility by passing a precreated configuration file.

Prerequisites

- Identify the source and sink for the migration.
  - Source: Oracle NoSQL Database
  - Sink: Oracle NoSQL Database Cloud Service

- Identify your OCI cloud credentials and capture them in the OCI config file. Save the config file in `/home/.oci/config`. See Acquiring Credentials in *Using Oracle NoSQL Database Cloud Service*.

```
[DEFAULT]
tenancy=ocid1.tenancy.oc1....
user=ocid1.user.oc1....
fingerprint= 43:d1:....
key_file=</fully/qualified/path/to/the/private/key/>
pass_phrase=<passphrase>
```

- Identify the region endpoint and compartment name for your Oracle NoSQL Database Cloud Service.
  - endpoint: `us-phoenix-1`
  - compartment: `developers`
- Identify the following details for the on-premise KVStore:
  - storeName: `kvstore`
  - helperHosts: `<hostname>:5000`
  - table: `myTable`

Procedure

To migrate the data and schema definition of `myTable` from NoSQL Database KVStore to NDCS:

1. Prepare the configuration file (in JSON format) with the identified Source and Sink details. See Source Configuration Templates and Sink Configuration Templates .

```
{
  "source" : {
    "type" : "nosqldb",
    "storeName" : "kvstore",
    "helperHosts" : ["<hostname>:5000"],
    "table" : "myTable",
    "requestTimeoutMs" : 5000
  },
  "sink" : {
    "type" : "nosqldb_cloud",
    "endpoint" : "us-phoenix-1",
    "table" : "myTable",
    "compartment" : "developers",
    "schemaInfo" : {
      "schemaPath" : "<complete/path/to/the/JSON/file/with/DDL/
commands/for/the/schema/definition>",
      "readUnits" : 100,
      "writeUnits" : 100,
      "storageSize" : 1
    },
    "credentials" : "<complete/path/to/oci/config/file>",
    "credentialsProfile" : "DEFAULT",
    "writeUnitsPercent" : 90,
    "requestTimeoutMs" : 5000
  },
```

```
  "abortOnError" : true,
  "migratorVersion" : "1.0.0"
}
```

2. Open the command prompt and navigate to the directory where you extracted the NoSQL Database Migrator utility.
3. Run the `runMigrator` command by passing the configuration file using the `--config` or `-c` option.

```
[~/nosqlMigrator/nosql-migrator-1.0.0]$./runMigrator --config <complete/
path/to/the/JSON/config/file>
```

4. The utility proceeds with the data migration, as shown below.

```
Records provided by source=10, Records written to sink=10, Records
failed=0.
Elapsed time: 0min 10sec 426ms
Migration completed.
```

Validation
To validate the migration, you can login to your NDCS console and verify that `myTable` is created with the source data.

Migrate from JSON file source to Oracle NoSQL Database Cloud Service

This example shows the usage of Oracle NoSQL Database Migrator to copy data from a JSON file source to Oracle NoSQL Database Cloud Service.

After evaluating multiple options, an organization finalizes Oracle NoSQL Database Cloud Service as its NoSQL Database platform. As its source contents are in JSON file format, they are looking for a way to migrate them to Oracle NoSQL Database Cloud Service.

In this example, you will learn to migrate the data from a JSON file called `SampleData.json`. You run the `runMigrator` utility by passing a pre-created configuration file. If the configuration file is not provided as a run time parameter, the `runMigrator` utility prompts you to generate the configuration through an interactive procedure.

Prerequisites

- Identify the source and sink for the migration.
  - Source: JSON source file.
    `SampleData.json` is the source file. It contains multiple JSON documents with one document per line, delimited by a new line character.

```
{"id":6,"val_json":{"array":
["q","r","s"],"date":"2023-02-04T02:38:57.520Z","nestarray":[[1,2,3],
[10,20,30]],"nested":{"arrayofobjects":
[{"datefield":"2023-03-04T02:38:57.520Z","numfield":30,"strfield":"foo
54"},
{"datefield":"2023-02-04T02:38:57.520Z","numfield":56,"strfield":"bar2
3"}],"nestNum":10,"nestString":"bar"},"num":1,"string":"foo"}}
{"id":3,"val_json":{"array":
["g","h","i"],"date":"2023-02-02T02:38:57.520Z","nestarray":[[1,2,3],
[10,20,30]],"nested":{"arrayofobjects":
```

```
[{"datefield":"2023-02-02T02:38:57.520Z","numfield":28,"strfield"
:"foo3"},
{"datefield":"2023-02-02T02:38:57.520Z","numfield":38,"strfield":
"bar"}],"nestNum":10,"nestString":"bar"},"num":1,"string":"foo"}}
{"id":7,"val_json":{"array":
["a","b","c"],"date":"2023-02-20T02:38:57.520Z","nestarray":
[[1,2,3],[10,20,30]],"nested":{"arrayofobjects":
[{"datefield":"2023-01-20T02:38:57.520Z","numfield":28,"strfield"
:"foo"},
{"datefield":"2023-01-22T02:38:57.520Z","numfield":38,"strfield":
"bar"}],"nestNum":10,"nestString":"bar"},"num":1,"string":"foo"}}
{"id":4,"val_json":{"array":
["j","k","l"],"date":"2023-02-03T02:38:57.520Z","nestarray":
[[1,2,3],[10,20,30]],"nested":{"arrayofobjects":
[{"datefield":"2023-02-03T02:38:57.520Z","numfield":28,"strfield"
:"foo"},
{"datefield":"2023-02-03T02:38:57.520Z","numfield":38,"strfield":
"bar"}],"nestNum":10,"nestString":"bar"},"num":1,"string":"foo"}}
```

  - Sink: Oracle NoSQL Database Cloud Service.

- Identify your OCI cloud credentials and capture them in the configuration file. Save the config file in `/home/user/.oci/config`. For more details, see Acquiring Credentials in *Using Oracle NoSQL Database Cloud Service*.

```
[DEFAULT]
tenancy=ocid1.tenancy.oc1....
user=ocid1.user.oc1....
fingerprint= 43:d1:....
region=us-ashburn-1
key_file=</fully/qualified/path/to/the/private/key/>
pass_phrase=<passphrase>
```

- Identify the region endpoint and compartment name for your Oracle NoSQL Database Cloud Service.
  - endpoint: `us-ashburn-1`
  - compartment: `Training-NoSQL`
- Identify the following details for the JSON source file:
  - schemaPath: `<absolute path to the schema definition file containing DDL statements for the NoSQL table at the sink>`.

    In this example, the DDL file is `schema_json.ddl`.

```
create table Migrate_JSON (id INTEGER, val_json JSON, PRIMARY
    KEY(id));
```

    The Oracle NoSQL Database Migrator provides an option to create a table with the default schema if the `schemaPath` is not provided. For more details, see *Identify the Source and Sink* topic in the Workflow for Oracle NoSQL Database Migrator.
  - Datapath: `<absolute path to a file or directory containing the JSON data for migration>`.

Procedure

To migrate the JSON source file from `SampleData.json` to Oracle NoSQL Database Cloud Service, perform the following:

1. Prepare the configuration file (in JSON format) with the identified source and sink details. See Source Configuration Templates and Sink Configuration Templates .

```
{
  "source" : {
    "type" : "file",
    "format" : "json",
    "schemaInfo" : {
      "schemaPath" : "[~/nosql-migrator-1.5.0]/schema_json.ddl"
    },
    "dataPath" : "[~/nosql-migrator-1.5.0]/SampleData.json"
  },
  "sink" : {
    "type" : "nosqldb_cloud",
    "endpoint" : "us-ashburn-1",
    "table" : "Migrate_JSON",
    "compartment" : "Training-NoSQL",
    "includeTTL" : false,
    "schemaInfo" : {
      "readUnits" : 100,
      "writeUnits" : 60,
      "storageSize" : 1,
      "useSourceSchema" : true
    },
    "credentials" : "/home/user/.oci/config",
    "credentialsProfile" : "DEFAULT",
    "writeUnitsPercent" : 90,
    "overwrite" : true,
    "requestTimeoutMs" : 5000
  },
  "abortOnError" : true,
  "migratorVersion" : "1.5.0"
}
```

2. Open the command prompt and navigate to the directory where you extracted the Oracle NoSQL Database Migrator utility.

3. Run the `runMigrator` command by passing the configuration file using the `--config` or `-c` option.

```
[~/nosql-migrator-1.5.0]$./runMigrator --config <complete/path/to/the/
config/file>
```

4. The utility proceeds with the data migration, as shown below. The `Migrate_JSON` table is created at the sink with the schema provided in the `schemaPath`.

```
creating source from given configuration:
source creation completed
creating sink from given configuration:
sink creation completed
creating migrator pipeline
```

```
migration started
[cloud sink] : start loading DDLs
[cloud sink] : executing DDL: create table Migrate_JSON (id
INTEGER, val_json JSON, PRIMARY KEY(id)),limits: [100, 60, 1]
[cloud sink] : completed loading DDLs
[cloud sink] : start loading records
[json file source] : start parsing JSON records from file:
SampleData.json
[INFO] migration completed.
Records provided by source=4, Records written to sink=4, Records
failed=0, Records skipped=0.
Elapsed time: 0min 5sec 778ms
Migration completed.
```

Validation

To validate the migration, you can log in to your Oracle NoSQL Database Cloud Service console and verify that the `Migrate_JSON` table is created with the source data. For the procedure to access the console, see Accessing the Service from the Infrastructure Console article in the *Oracle NoSQL Database Cloud Service* document.

Figure 5-1 Oracle NoSQL Database Cloud Service Console Tables

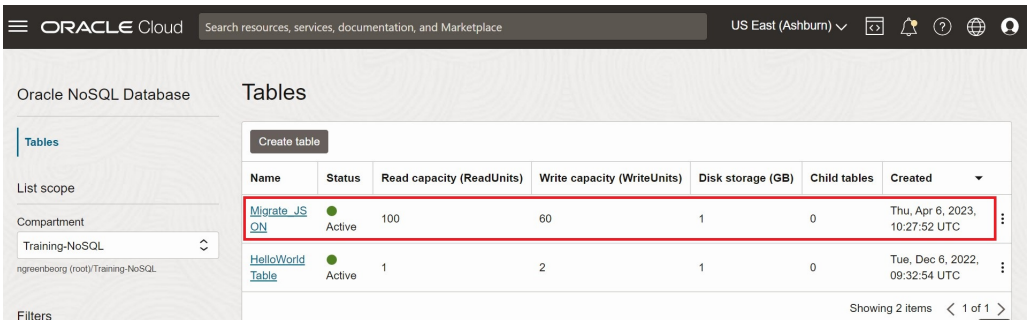
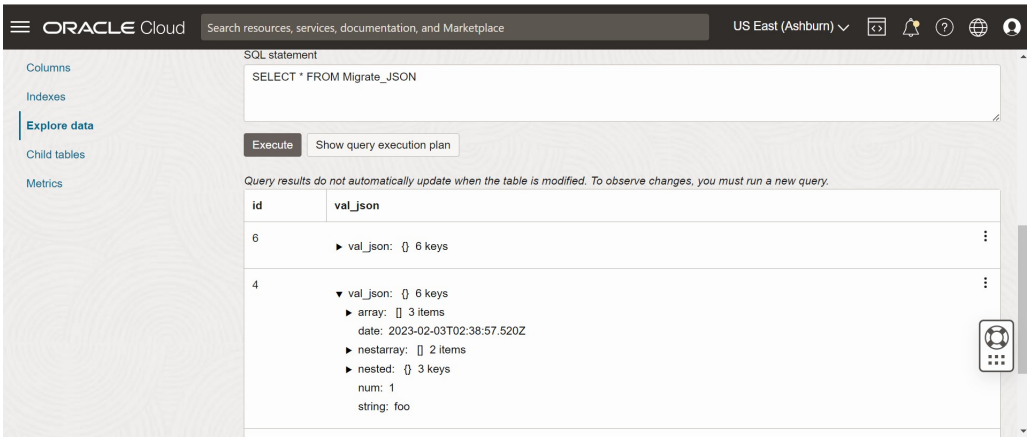


Figure 5-2 Oracle NoSQL Database Cloud Service Console Table Data



Migrate from MongoDB JSON file to an Oracle NoSQL Database Cloud Service

This example shows how to use the Oracle NoSQL Database Migrator to copy Mongo-DB Formatted Data to the Oracle NoSQL Database (NDCS).

Use Case

After evaluating multiple options, an organization finalizes Oracle NoSQL Database as its NoSQL Database platform. As its NoSQL tables and data are in MongoDB, they are looking for a way to migrate those tables and data to Oracle NDCS.

You can copy a file or directory containing the MongoDB exported JSON data for migration by specifying the file or directory in the source configuration template.

A sample MongoDB-formatted JSON File is as follows:

```
{"_id":0,"name":"Aimee Zank","scores":
[{"score":1.463179736705023,"type":"exam"},
{"score":11.78273309957772,"type":"quiz"},
{"score":35.8740349954354,"type":"homework"}]}
{"_id":1,"name":"Aurelia Menendez","scores":
[{"score":60.06045071030959,"type":"exam"},
{"score":52.79790691903873,"type":"quiz"},
{"score":71.76133439165544,"type":"homework"}]}
{"_id":2,"name":"Corliss Zuk","scores":
[{"score":67.03077096065002,"type":"exam"},
{"score":6.301851677835235,"type":"quiz"},
{"score":66.28344683278382,"type":"homework"}]}
{"_id":3,"name":"Bao Ziglar","scores":
[{"score":71.64343899778332,"type":"exam"},
{"score":24.80221293650313,"type":"quiz"},
{"score":42.26147058804812,"type":"homework"}]}
{"_id":4,"name":"Zachary Langlais","scores":
[{"score":78.68385091304332,"type":"exam"},
{"score":90.2963101368042,"type":"quiz"},
{"score":34.41620148042529,"type":"homework"}]}
```

MongoDB supports two types of extensions to the JSON format of files, *Canonical mode* and *Relaxed mode*. You can supply the MongoDB-formatted JSON file that is generated using the *mongoexport* tool in either Canonical or Relaxed mode. Both the modes are supported by the NoSQL Database Migrator for migration.

For more information on the MongoDB Extended JSON (v2) file, See mongoexport_formats.

For more information on the generation of MongoDB-formatted JSON file, See mongoexport.

Example

For the demonstration, let us look at how to migrate a MongoDB-formatted JSON file to NDCS. We will use a manually created configuration file for this example.

Prerequisites

- Identify the source and sink for the migration.
  - Source: MongoDB-Formatted JSON File

  - Sink: Oracle NoSQL Database
- Extract the data from Mongo DB using the *mongoexport* utility. See mongoexport for more information.
- Create a NoSQL table in the sink with a table schema that matches the data in the Mongo-DB-formatted JSON file. As an alternative, you can instruct the NoSQL Database Migratorto create a table with the default schema structure by setting the `defaultSchema` attribute to true.

> **Note:**
>
> For a *MongoDB-Formatted JSON* source, the default schema for the table will be as:
>
> ```
> CREATE TABLE IF NOT EXISTS <tablename>(ID STRING, DOCUMENT
> JSON,PRIMARY KEY(SHARD(ID))
> ```
>
> Where:
>
> - `tablename` = value of the table config.
> - `ID` = `_id` value from the mongoDB exported JSON source file.
> - `DOCUMENT` = The entire contents of the mongoDB exported JSON source file is aggregated into the `DOCUMENT` column excluding the `_id` field.

- Identify your OCI cloud credentials and capture them in the OCI config file. Save the config file in `/home/.oci/config`.See Acquiring Credentials in *Using Oracle NoSQL Database Cloud Service*.

```
[DEFAULT]
tenancy=ocid1.tenancy.oc1....
user=ocid1.user.oc1....
fingerprint= 43:d1:....
key_file=</fully/qualified/path/to/the/private/key/>
pass_phrase=<passphrase>
```

- Identify the region endpoint and compartment name for your Oracle NoSQL Database.
  - endpoint: `us-phoenix-1`
  - compartment: `developers`

Procedure
To migrate the MongoDB-formatted JSON data to the Oracle NoSQL Database:

1. Prepare the configuration file (in JSON format) with the identified Source and Sink details. See Source Configuration Templates and Sink Configuration Templates .

```
{
  "source" : {
    "type" : "file",
    "format" : "mongodb_json",
```

```
    "dataPath" : "<complete/path/to/the/MongoDB/Formatted/JSON/file>"
  },
  "sink" : {
    "type" : "nosqldb_cloud",
    "endpoint" : "us-phoenix-1",
    "table" : "mongoImport",
    "compartment" : "developers",
    "schemaInfo" : {
      "defaultSchema" : true,
      "readUnits" : 100,
      "writeUnits" : 60,
      "storageSize" : 1
    },
    "credentials" : "<complete/path/to/the/oci/config/file>",
    "credentialsProfile" : "DEFAULT",
    "writeUnitsPercent" : 90,
    "requestTimeoutMs" : 5000
  },
  "abortOnError" : true,
  "migratorVersion" : "1.0.0"
}
```

2. Open the command prompt and navigate to the directory where you extracted the NoSQL Database Migrator utility.
3. Run the `runMigrator` command by passing the configuration file using the `--config` or `-c` option.

```
[~/nosqlMigrator/nosql-migrator-1.0.0]$./runMigrator --config <complete/
path/to/the/JSON/config/file>
```

4. The utility proceeds with the data migration, as shown below.

```
Records provided by source=29,353, Records written to sink=29,353,
Records failed=0.
Elapsed time: 9min 9sec 630ms
Migration completed.
```

Validation
To validate the migration, you can login to your NDCS console and verify that `myTable` is created with the source data.

Migrate from DynamoDB JSON file to Oracle NoSQL Database

This example shows how to use the Oracle NoSQL Database Migrator to copy DynamoDB JSON file to Oracle NoSQL Database.

Use Case:

After evaluating multiple options, an organization finalizes Oracle NoSQL Database over DynamoDB database. The organization wants to migrate their tables and data from DynamoDB to Oracle NoSQL Database (On-premises).

See Mapping of DynamoDB table to Oracle NoSQL table for more details.

You can migrate a file or directory containing the DynamoDB exported JSON data from a file system by specifying the path in the source configuration template.

A sample DynamoDB-formatted JSON File is as follows:

```
{"Item":{"Id":{"N":"101"},"Phones":{"L":[{"L":[{"S":"555-222"},
{"S":"123-567"}]}]},"PremierCustomer":{"BOOL":false},"Address":{"M":
{"Zip":{"N":"570004"},"Street":{"S":"21 main"},"DoorNum":
{"N":"201"},"City":{"S":"London"}}},"FirstName":
{"S":"Fred"},"FavNumbers":{"NS":["10"]},"LastName":
{"S":"Smith"},"FavColors":{"SS":["Red","Green"]},"Age":{"N":"22"}}}
{"Item":{"Id":{"N":"102"},"Phones":{"L":[{"L":
[{"S":"222-222"}]}]},"PremierCustomer":{"BOOL":false},"Address":{"M":
{"Zip":{"N":"560014"},"Street":{"S":"32 main"},"DoorNum":
{"N":"1024"},"City":{"S":"Wales"}}},"FirstName":
{"S":"John"},"FavNumbers":{"NS":["10"]},"LastName":
{"S":"White"},"FavColors":{"SS":["Blue"]},"Age":{"N":"48"}}}
```

You copy the exported DynamoDB table data from AWS S3 storage to a local mounted file system.

Example:

For this demonstration, you will learn how to migrate a DynamoDB JSON file to Oracle NoSQL Database(On-premises). You will use a manually created configuration file for this example.

Prerequisites

- Identify the source and sink for the migration.
  - **Source:** DynamoDB JSON File
  - **Sink:** Oracle NoSQL Database (On-premises)
- In order to import DynamoDB table data to Oracle NoSQL Database, you must first export the DynamoDB table to S3. Refer to steps provided in Exporting DynamoDB table data to Amazon S3 to export your table. While exporting, you select the format as **DynamoDB JSON**. The exported data contains DynamoDB table data in multiple `gzip` files as shown below.

  ```
  / 01639372501551-bb4dd8c3
  |-- 01639372501551-bb4dd8c3 ==> exported data prefix
  |----data
  |------sxz3hjr3re2dzn2ymgd2gi4iku.json.gz  ==>table data
  |----manifest-files.json
  |----manifest-files.md5
  |----manifest-summary.json
  |----manifest-summary.md5
  |----_started
  ```

- You must download the files from AWS s3. The structure of the files after the download will be as shown below.

  ```
  download-dir/01639372501551-bb4dd8c3
  |----data
  ```

```
|------sxz3hjr3re2dzn2ymgd2gi4iku.json.gz  ==>table data
|----manifest-files.json
|----manifest-files.md5
|----manifest-summary.json
|----manifest-summary.md5
|----_started
```

Procedure

To migrate the DynamoDB JSON data to the Oracle NoSQL Database:

1. Prepare the configuration file (in JSON format) with the identified source and sink details.See Source Configuration Templates and Sink Configuration Templates .
   You can choose one of the following two options.

   - **Option 1:** Importing DynamoDB table a as JSON document using default schema config.
     Here the `defaultSchema` is `TRUE` and so the migrator creates the default schema at the sink. You need to specify the `DDBPartitionKey` and the corresponding NoSQL column type. Else an error is thrown.

```
{
 "source" : {
   "type" : "file",
   "format" : "dynamodb_json",
   "dataPath" : "<complete/path/to/the/DynamoDB/Formatted/JSON/file>"
 },
 "sink" : {
    "type" : "nosqldb",
    "table" : "<table_name>",
    "storeName" : "kvstore",
    "helperHosts" : ["<hostname>:5000"]
    "schemaInfo" : {
       "defaultSchema" : true,
       "DDBPartitionKey" : "<PrimaryKey:Datatype>",
     },
  },
  "abortOnError" : true,
  "migratorVersion" : "1.0.0"
}
```

     For a DynamoDB JSON source, the default schema for the table will be as shown below:

```
CREATE TABLE IF NOT EXISTS <TABLE_NAME>(DDBPartitionKey_name
DDBPartitionKey_type,
[DDBSortKey_name DDBSortKey_type], DOCUMENT JSON,
PRIMARY KEY(SHARD(DDBPartitionKey_name),[DDBSortKey_name]))
```

     Where

     TABLE_NAME = value provided for the sink 'table' in the configuration

     DDBPartitionKey_name = value provided for the partiiton key in the configuration

DDBPartitionKey_type = value provided for the data type of the partition key in the configuration

DDBSortKey_name = value provided for the sort key in the configuration if any

DDBSortKey_type = value provided for the data type of the sort key in the configuration if any

DOCUMENT = All attributes except the partition and sort key of a Dynamo DB table item aggregated into a NoSQL JSON column

- **Option 2:** Importing DynamoDB table as fixed columns using a user-supplied schema file.
  Here the `defaultSchema` is `FALSE` and you specify the schemaPath as a file containing your DDL statement. See Mapping of DynamoDB types to Oracle NoSQL types for more details.

> **Note:**
>
> If the Dynamo DB table has a data type that is not supported in NoSQL, the migration fails.

A sample schema file is shown below.

```
CREATE TABLE IF NOT EXISTS sampledynDBImp (AccountId
INTEGER,document JSON,
PRIMARY KEY(SHARD(AccountId)));
```

The schema file is used to create the table at the sink as part of the migration. As long as the primary key data is provided, the input JSON record will be inserted, otherwise it throws an error.

> **Note:**
>
> If the input data does not contain a value for a particular column(other than the primary key) then the column default value will be used. The default value should be part of the column definition while creating the table. For example `id INTEGER not null default 0`. If the column does not have a default definition then SQL NULL is inserted if no values are provided for the column.

```
{
  "source" : {
    "type" : "file",
    "format" : "dynamodb_json",
    "dataPath" : "<complete/path/to/the/DynamoDB/Formatted/JSON/
file>"
  },
  "sink" : {
    "type" : "nosqldb",
    "table" : "<table_name>",
    "schemaInfo" : {
```

```
            "defaultSchema" : false,
            "readUnits" : 100,
            "writeUnits" : 60,
            "schemaPath" : "<full path of the schema file with the DDL
statement>",
            "storageSize" : 1
        },
        "storeName" : "kvstore",
        "helperHosts" : ["<hostname>:5000"]
      },
      "abortOnError" : true,
      "migratorVersion" : "1.0.0"
    }
```

2. Open the command prompt and navigate to the directory where you extracted the NoSQL Database Migrator utility.
3. Run the `runMigrator` command by passing the configuration file using the `--config` or `-c` option.

```
[~/nosqlMigrator/nosql-migrator-1.0.0]$./runMigrator
--config <complete/path/to/the/JSON/config/file>
```

4. The utility proceeds with the data migration, as shown below.

```
Records provided by source=7..,
Records written to sink=7,
Records failed=0,
Records skipped=0.
Elapsed time: 0 min 2sec 50ms
Migration completed.
```

Validation

Start the SQL prompt in your KVStore.

```
 java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Verify that the new table is created with the source data:

```
desc <table_name>
SELECT * from <table_name>
```

Migrate from DynamoDB JSON file in AWS S3 to an Oracle NoSQL Database Cloud Service

This example shows how to use the Oracle NoSQL Database Migrator to copy DynamoDB JSON file stored in an AWS S3 store to the Oracle NoSQL Database Cloud Service (NDCS).

Use Case:

After evaluating multiple options, an organization finalizes Oracle NoSQL Database Cloud Service over DynamoDB database. The organization wants to migrate their tables and data from DynamoDB to Oracle NoSQL Database Cloud Service.

See Mapping of DynamoDB table to Oracle NoSQL table for more details.

You can migrate a file containing the DynamoDB exported JSON data from the AWS S3 storage by specifying the path in the source configuration template.

A sample DynamoDB-formatted JSON File is as follows:

```
{"Item":{"Id":{"N":"101"},"Phones":{"L":[{"L":[{"S":"555-222"},
{"S":"123-567"}]}]},"PremierCustomer":{"BOOL":false},"Address":{"M":
{"Zip":{"N":"570004"},"Street":{"S":"21 main"},"DoorNum":
{"N":"201"},"City":{"S":"London"}}},"FirstName":
{"S":"Fred"},"FavNumbers":{"NS":["10"]},"LastName":
{"S":"Smith"},"FavColors":{"SS":["Red","Green"]},"Age":{"N":"22"}}}
{"Item":{"Id":{"N":"102"},"Phones":{"L":[{"L":
[{"S":"222-222"}]}]},"PremierCustomer":{"BOOL":false},"Address":{"M":
{"Zip":{"N":"560014"},"Street":{"S":"32 main"},"DoorNum":
{"N":"1024"},"City":{"S":"Wales"}}},"FirstName":
{"S":"John"},"FavNumbers":{"NS":["10"]},"LastName":
{"S":"White"},"FavColors":{"SS":["Blue"]},"Age":{"N":"48"}}}
```

You export the DynamoDB table to AWS S3 storage as specified in Exporting DynamoDB table data to Amazon S3.

Example:

For this demonstration, you will learn how to migrate a DynamoDB JSON file in an AWS S3 source to NDCS. You will use a manually created configuration file for this example.

Prerequisites

- Identify the source and sink for the migration.
  - **Source:** DynamoDB JSON File in AWS S3
  - **Sink:** Oracle NoSQL Database Cloud Service
- Identify the table in AWS DynamoDB that needs to be migrated to NDCS. Login to your AWS console using your credentials. Go to **DynamoDB**. Under **Tables**, choose the table to be migrated.
- Create an object bucket and export the table to S3. From your AWS console, go to **S3**. Under buckets, create a new object bucket. Go back to DynamoDB and click **Exports to S3**. Provide the source table and the destination S3 bucket and click **Export**.
  Refer to steps provided in Exporting DynamoDB table data to Amazon S3 to export your table. While exporting, you select the format as **DynamoDB JSON**. The exported data contains DynamoDB table data in multiple `gzip` files as shown below.

  ```
  / 01639372501551-bb4dd8c3
  |-- 01639372501551-bb4dd8c3 ==> exported data prefix
  |----data
  |------sxz3hjr3re2dzn2ymgd2gi4iku.json.gz  ==>table data
  |----manifest-files.json
  |----manifest-files.md5
  |----manifest-summary.json
  ```

```
|----manifest-summary.md5
|----_started
```

- You need aws credentials (including access key ID and secret access key) and config files (credentials and optionally config) to access AWS S3 from the migrator. See Set and view configuration settings for more details on the configuration files. See Creating a key pair for more details on creating access keys.
- Identify your OCI cloud credentials and capture them in the OCI config file. Save the config file in a directory `.oci` under your home directory (`~/.oci/config`). See Acquiring Credentials for more details.

```
[DEFAULT]
tenancy=ocid1.tenancy.oc1....
user=ocid1.user.oc1....
fingerprint= 43:d1:....
key_file=</fully/qualified/path/to/the/private/key/>
pass_phrase=<passphrase>
```

- Identify the region endpoint and compartment name for your Oracle NoSQL Database. For example,
  - **endpoint:** us-phoenix-1
  - **compartment:** developers

Procedure

To migrate the DynamoDB JSON data to the Oracle NoSQL Database:

1. Prepare the configuration file (in JSON format) with the identified source and sink details. See Source Configuration Templates and Sink Configuration Templates .
   You can choose one of the following two options.
   - **Option 1:** Importing DynamoDB table a as JSON document using default schema config.
     Here the `defaultSchema` is `TRUE` and so the migrator creates the default schema at the sink. You need to specify the `DDBPartitionKey` and the corresponding NoSQL column type. Else an error is thrown.

```
{
 "source" : {
   "type" : "aws_s3",
   "format" : "dynamodb_json",
   "s3URL" : "<https://<bucket-name>.<s3_endpoint>/export_path>",
   "credentials" : "</path/to/aws/credentials/file>",
   "credentialsProfile" : <"profile name in aws credentials file">
 },
 "sink" : {
   "type" : "nosqldb_cloud",
   "endpoint" : "<region_name>",
   "table" : "<table_name>",
   "compartment" : "<compartment_name>",
   "schemaInfo" : {
      "defaultSchema" : true,
      "readUnits" : 100,
      "writeUnits" : 60,
```

```
      "DDBPartitionKey" : "<PrimaryKey:Datatype>",
      "storageSize" : 1
   },
   "credentials" : "<complete/path/to/the/oci/config/file>",
   "credentialsProfile" : "DEFAULT",
   "writeUnitsPercent" : 90,
   "requestTimeoutMs" : 5000
 },
 "abortOnError" : true,
 "migratorVersion" : "1.0.0"
}
```

For a DynamoDB JSON source, the default schema for the table will be as shown below:

```
CREATE TABLE IF NOT EXISTS <TABLE_NAME>(DDBPartitionKey_name
DDBPartitionKey_type,
[DDBSortKey_name DDBSortKey_type], DOCUMENT JSON,
PRIMARY KEY(SHARD(DDBPartitionKey_name),[DDBSortKey_name]))
```

Where

TABLE_NAME = value provided for the sink 'table' in the configuration

DDBPartitionKey_name = value provided for the partiiton key in the configuration

DDBPartitionKey_type = value provided for the data type of the partition key in the configuration

DDBSortKey_name = value provided for the sort key in the configuration if any

DDBSortKey_type = value provided for the data type of the sort key in the configuration if any

DOCUMENT = All attributes except the partition and sort key of a Dynamo DB table item aggregated into a NoSQL JSON column

- **Option 2:** Importing DynamoDB table as fixed columns using a user-supplied schema file.
  Here the `defaultSchema` is `FALSE` and you specify the schemaPath as a file containing your DDL statement. See Mapping of DynamoDB types to Oracle NoSQL types for more details.

> **Note:**
>
> If the Dynamo DB table has a data type that is not supported in NoSQL, the migration fails.

A sample schema file is shown below.

```
CREATE TABLE IF NOT EXISTS sampledynDBImp (AccountId
INTEGER,document JSON,
PRIMARY KEY(SHARD(AccountId)));
```

The schema file is used to create the table at the sink as part of the migration. As long as the primary key data is provided, the input JSON record will be inserted, otherwise it throws an error.

> **Note:**
>
> If the input data does not contain a value for a particular column(other than the primary key) then the column default value will be used. The default value should be part of the column definition while creating the table. For example `id INTEGER not null default 0`. If the column does not have a default definition then SQL NULL is inserted if no values are provided for the column.

```
{
 "source" : {
   "type" : "aws_s3",
   "format" : "dynamodb_json",
   "s3URL" : "<https://<bucket-name>.<s3_endpoint>/export_path>",
   "credentials" : "</path/to/aws/credentials/file>",
   "credentialsProfile" : <"profile name in aws credentials file">
 },
 "sink" : {
   "type" : "nosqldb_cloud",
   "endpoint" : "<region_name>",
   "table" : "<table_name>",
   "compartment" : "<compartment_name>",
   "schemaInfo" : {
      "defaultSchema" : false,
      "readUnits" : 100,
      "writeUnits" : 60,
      "schemaPath" : "<full path of the schema file with the DDL
statement>",
      "storageSize" : 1
   },
   "credentials" : "<complete/path/to/the/oci/config/file>",
   "credentialsProfile" : "DEFAULT",
   "writeUnitsPercent" : 90,
   "requestTimeoutMs" : 5000
 },
 "abortOnError" : true,
 "migratorVersion" : "1.0.0"
}
```

2. Open the command prompt and navigate to the directory where you extracted the NoSQL Database Migrator utility.
3. Run the `runMigrator` command by passing the configuration file using the `--config` or `-c` option.

```
[~/nosqlMigrator/nosql-migrator-1.0.0]$./runMigrator
--config <complete/path/to/the/JSON/config/file>
```

4. The utility proceeds with the data migration, as shown below.

```
Records provided by source=7..,
Records written to sink=7,
Records failed=0,
Records skipped=0.
Elapsed time: 0 min 2sec 50ms
Migration completed.
```

Validation

You can login to your NDCS console and verify that the new table is created with the source data.

Migrate from CSV file to Oracle NoSQL Database

This example shows the usage of Oracle NoSQL Database Migrator to copy data from a CSV file to Oracle NoSQL Database.

Example

After evaluating multiple options, an organization finalizes Oracle NoSQL Database as its NoSQL Database platform. As its source contents are in CSV file format, they are looking for a way to migrate them to Oracle NoSQL Database.

In this example, you will learn to migrate the data from a CSV file called `course.csv`, which contains information about various courses offered by a university. You generate the configuration file from the `runMigrator` utility.

You can also prepare the configuration file with the identified source and sink details. See Sources and Sinks.

Prerequisites

- Identify the source and sink for the migration.
  - Source: CSV file
    In this example, the source file is `course.csv`

    ```
    cat [~/nosql-migrator-1.5.0]/course.csv
    1,"Computer Science", "San Francisco", "2500"
    2,"Bio-Technology", "Los Angeles", "1200"
    3,"Journalism", "Las Vegas", "1500"
    4,"Telecommunication", "San Francisco", "2500"
    ```

  - Sink: Oracle NoSQL Database
- The CSV file must conform to the `RFC4180` format.
- Create a file containing the DDL commands for the schema of the target table, `course`. The table definition must match the CSV data file concerning the number of columns and their types.
  In this example, the DDL file is `mytable_schema.ddl`

  ```
  cat [~/nosql-migrator-1.5.0]/mytable_schema.ddl
  ```

```
create table course (id INTEGER, name STRING, location STRING, fees
INTEGER, PRIMARY KEY(id));
```

Procedure

To migrate the CSV file data from `course.csv` to Oracle NoSQL Database Service, perform the following steps:

1. Open the command prompt and navigate to the directory where you extracted the Oracle NoSQL Database Migrator utility.
2. To generate the configuration file using Oracle NoSQL Database Migrator, execute the `runMigrator` command without any runtime parameters.

   ```
   [~/nosql-migrator-1.5.0]$./runMigrator
   ```

3. As you did not provide the configuration file as a runtime parameter, the utility prompts if you want to generate the configuration now. Type `y`.

   You can choose a location for the configuration file or retain the default location by pressing the `Enter key`.

   ```
   Configuration file is not provided. Do you want to generate
   configuration? (y/n) [n]: y
   Generating a configuration file interactively.

   Enter a location for your config [./migrator-config.json]:
   ./migrator-config.json already exist. Do you want to overwrite?(y/n) [n]:
   y
   ```

4. Based on the prompts from the utility, choose your options for the Source configuration.

   ```
   Select the source:
   1) nosqldb
   2) nosqldb_cloud
   3) file
   4) object_storage_oci
   5) aws_s3
   #? 3

   Configuration for source type=file
   Select the source file format:
   1) json
   2) mongodb_json
   3) dynamodb_json
   4) csv
   #? 4
   ```

5. Provide the path to the source CSV file. Further, based on the prompts from the utility, you can choose to reorder the column names, select the encoding method, and trim the tailing spaces from the target table.

   ```
   Enter path to a file or directory containing csv data: [~/nosql-
   migrator-1.5.0]/course.csv
   ```

```
Does the CSV file contain a headerLine? (y/n) [n]: n
Do you want to reorder the column names of NoSQL table with respect
to
CSV file columns? (y/n) [n]: n
Provide the CSV file encoding. The supported encodings are:
UTF-8,UTF-16,US-ASCII,ISO-8859-1. [UTF-8]:
Do you want to trim the tailing spaces? (y/n) [n]: n
```

6. Based on the prompts from the utility, choose your options for the Sink configuration.

```
Select the sink:
1) nosqldb
2) nosqldb_cloud
#? 1
Configuration for sink type=nosqldb
Enter store name of the Oracle NoSQL Database: mystore
Enter comma separated list of host:port of Oracle NoSQL Database:
<hostname>:5000
```

7. Based on the prompts from the utility, provide the name of the target table.

```
Enter fully qualified table name: course
```

8. Enter your choice to set the TTL value. The default value is `n`.

```
Include TTL data? If you select 'yes' TTL value provided by the
source will be set on imported rows. (y/n) [n]: n
```

9. Based on the prompts from the utility, specify whether or not the target table must be created through the Oracle NoSQL Database Migrator tool. If the table is already created, it is suggested to provide `n`. If the table is not created, the utility will request the path for the file containing the DDL commands for the schema of the target table.

```
Would you like to create table as part of migration process?
Use this option if you want to create table through the migration
tool.
If you select yes, you will be asked to provide a file that contains
table DDL or to use schema provided by the source or default schema.
(y/n) [n]: y
Enter path to a file containing table DDL: [~/nosql-migrator-1.5.0]/
mytable_schema.ddl
Is the store secured? (y/n) [y]: n
would you like to overwrite records which are already present?
If you select 'no' records with same primary key will be skipped
[y/n] [y]: y
Enter store operation timeout in milliseconds. [5000]:
Would you like to add transformations to source data? (y/n) [n]: n
```

10. Enter your choice to determine whether to proceed with the migration in case any record fails to migrate.

```
Would you like to continue migration if any data fails to be migrated?
(y/n) [n]: n
```

11. The utility displays the generated configuration on the screen.

```
Generated configuration is:
{
  "source" : {
    "type" : "file",
    "format" : "csv",
    "dataPath" : "[~/nosql-migrator-1.5.0]/course.csv",
    "hasHeader" : false,
    "csvOptions" : {
      "encoding" : "UTF-8",
      "trim" : false
    }
  },
  "sink" : {
    "type" : "nosqldb",
    "storeName" : "mystore",
    "helperHosts" : ["<hostname>:5000"],
    "table" : "migrated_table",
    "query" : "",
    "includeTTL" : false,
    "schemaInfo" : {
      "schemaPath" : "[~/nosql-migrator-1.5.0]/mytable_schema.ddl"
    },
    "overwrite" : true,
    "requestTimeoutMs" : 5000
  },
  "abortOnError" : true,
  "migratorVersion" : "1.5.0"
}
```

12. Finally, the utility prompts you to specify whether or not to proceed with the migration using the generated configuration file. The default option is `y`.

    Note: If you select `n`, you can use the generated configuration file to perform the migration. Specify the `./runMigrator -c` or the `./runMigrator --config` option.

```
Would you like to run the migration with above configuration?
If you select no, you can use the generated configuration file to
run the migration using:
./runMigrator --config ./migrator-config.json
(y/n) [y]: y
```

13. The NoSQL Database Migrator copies your data from the CSV file to Oracle NoSQL Database.

```
creating source from given configuration:
source creation completed
creating sink from given configuration:
sink creation completed
creating migrator pipeline
migration started
[nosqldb sink] : start loading DDLs
[nosqldb sink] : executing DDL: create table course (id INTEGER,
name STRING, location STRING, fees INTEGER, PRIMARY KEY(id))
[nosqldb sink] : completed loading DDLs
[nosqldb sink] : start loading records
[csv file source] : start parsing CSV records from file: course.csv
migration completed. Records provided by source=4, Records written
to sink=4, Records failed=0,Records skipped=0.
Elapsed time: 0min 0sec 559ms
Migration completed.
```

Validation
Start the SQL prompt in your KVStore.

```
 java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Verify that the new table is created with the source data:

```
sql-> select * from course;
{"id":4,"name":"Telecommunication","location":"San
Francisco","fees":2500}
{"id":1,"name":"Computer Science","location":"San
Francisco","fees":2500}
{"id":2,"name":"Bio-Technology","location":"Los Angeles","fees":1200}
{"id":3,"name":"Journalism","location":"Las Vegas","fees":1500}

4 rows returned
```

Troubleshooting the Oracle NoSQL Database Migrator

Learn about the general challenges that you may face while using the , and how to resolve them.

Migration has failed. How can I resolve this?

A failure of the data migration can be because of multiple underlying reasons. The important causes are listed below:

Table 5-4 Migration Failure Causes

Error Message	Meaning	Resolution
`Failed to connect to Oracle NoSQL Database`	The migrator could not establish a connection with the NoSQL Database.	• Check if the values of the `storeName` and `helperHosts` attributes in the configuration JSON file are valid and that the hosts are reachable. • For a secured store, verify if the security file is valid with correct user name and password values.
`Failed to connect to Oracle NoSQL Database Cloud Service`	The migrator could not establish a connection with the Oracle NoSQL Database Cloud Service.	• Verify if the endpoint URL or region name specified in the configuration JSON file is correct. • Check if the OCI credentials file is available in the path specified in the configuration JSON file. • Ensure that the OCI credentials provided in the OCI credentials are valid.
`Table not found`	The table identified for the migration could not be located by the NoSQL Database Migrator.	**For the Source:** • Verify if the table is present in the source database. • Ensure that the table is qualified with its namespace in the configuration JSON file, if the table is created in a non-default namespace. • Verify if you have the required read/write authorization to access the table. • If the source is Oracle NoSQL Database Cloud Service, verify if the valid compartment name is specified in the configuration JSON file, and ensure that you have the required authorization to access the table. **For the Sink:** • Verify if the table is present in the Sink. If it does not exist, you must either create the table manually or use the `schemaInfo` config to create it through the migration.

Table 5-4 (Cont.) Migration Failure Causes

Error Message	Meaning	Resolution
`DDL Execution failed`	The DDL commands provided in the input schema definition file is invalid.	• Check the syntax of the DDL commands in the `schemaPath` file. • Ensure that there is only one DDL statement per line in the `schemaPath` file.
`failed to write record to the sink table with java.lang.IllegalArgument Exception`	The input record is not matching with the table schema of the sink.	• Check if the data types and column names specified in the target sink table are matching with sink table schema. • If you applied any transformation, check if the transformed records are matching with the sink table schema.
`Request timeout`	The source or sink's operation did not complete within the expected time.	• Verify the network connection. • Check if the NoSQL Database is up and running. • Try to increase `requestTimeout` value in the configuration JSON file.

What should I consider before restarting a failed migration?

When a data migration task fails, the sink will be at an intermediate state containing the imported data until the point of failure. You can identify the error and failure details from the logs and restart the migration after diagnosing and correcting the error. A restarted migration starts over, processing all data from the beginning. There is no way to checkpoint and restart the migration from the point of failure. Therefore, NoSQL Database Migrator overwrites any record that was migrated to the sink already.

Migration is too slow. How can I speed it up?

The time taken for the data migration depends on multiple factors such as volume of data being migrated, network speed, current load on the database. In case of a cloud service, the speed of migration also depends on the read throughput and the write throughput provisioned. So, to improve the migration speed, you can:

- Try to reduce the current workload on your Oracle NoSQL Database while migrating the data.
- Ensure that the machine that is running the migration, source, and sink all are located in the same data center and the network latencies are minimal.
- In case of Oracle NoSQL Database Cloud Service, provision high read/write throughput and verify if the storage allocated for table is sufficient or not. If the NoSQL Database Migrator is not creating the table, you can increase the write throughput. If the migrator is creating the table, consider specifying a higher value for the `schemaInfo.writeUnits` parameter in the sink configuration. Once the data migration completes, you can lower this value. Be aware of daily limits on throughput changes. see Cloud Limits and Sink Configuration Templates .

I have a long running migration involving huge datasets. How can I track the progress of the migration?

You can enable additional logging to track the progress of a long-running migration. To control the logging behavior of Oracle NoSQL Database Migrator, you must set the desired level of logging in the `logging.properties` file. This file is provided with the NoSQL Database Migrator package and available in the directory where the Oracle NoSQL Database Migrator was unpacked. The different levels of logging are `OFF`, `SEVERE`, `WARNING`, `INFO`, `FINE`, and `ALL` in the order of increasing verbosity. Setting the log level to `OFF` turns off all the logging information, whereas setting the log level to `ALL` provides the full log information. The default log level is `WARNING`. All the logging output is configured to go to the console by default. You can see comments in the `logging.properties` file to know about each log level.

Using Plugins and Extensions for Development

Get familiar with the plugins and extensions available for developing NoSQL applications in the Oracle NoSQL Database from external integrated development environments (IDEs) or code editors.

Topics

- About Oracle Enterprise Manager (OEM) Plugin
- About IntelliJ Plugin
- About Eclipse plugin
- About Oracle NoSQL Database Visual Studio Code Extension

About Oracle Enterprise Manager (OEM) Plugin

The monitoring of a store in Oracle NoSQL Database can be done through its native command-line interface (CLI). The monitoring data is available through Java Management Extensions (JMX) interfaces allowing customers to build their own monitoring solutions. For more information on monitoring data, see Standardized Monitoring Interfaces .

In this current release, the integration of Oracle's Enterprise Manager (OEM) with Oracle NoSQL Database provides a graphical management interface tool to discover and monitor a deployed store.

The integration of Oracle NoSQL Database with OEM primarily takes the form of an EM plugin. The plugin allows monitoring through Enterprise Manager of the Oracle NoSQL Database store components, their availability, performance metrics, and operational parameters. The current 12.1.0.9.0 version of the plugin is compatible with multiple versions of the *Oracle Enterprise Manager Cloud Control* (EM versions 13.3.0 or earlier). See *Oracle Enterprise Manager Cloud Control Administrator's Guide*.

> **Note:**
>
> For a Storage Node Agent (SNA) to be discovered and monitored, it must be configured for JMX. JMX is not enabled by default. You can tell whether JMX is enabled on a deployed SNA issuing the `show parameters` command and checking the reported value of the mgmtClass parameter. If the value is not `oracle.kv.impl.mgmt.jmx.JmxAgent`, then you need to issue the `change-parameters` plan command to enable JMX.
>
> For example:
>
> ```
> plan change-parameters -service sn1 -wait \
> -params mgmtClass=oracle.kv.impl.mgmt.jmx.JmxAgent
> ```

Also, the EM agent process must have read permission on the contents of $KVROOT.

Importing and Deploying the EM Plug-in

Follow the steps below to import and deploy the EM plug-in:

1. Import the file (.opar) into the Enterprise Manager before deploying it. The plug-in is delivered to the user as a file inside the release package: `lib/12.1.0.9.0;_oracle.nosql.snab;_2000_0.opar`

   See *Importing Plug-In Archives* in the *Oracle Enterprise Manager Cloud Control Administrator's Guide*.

2. Copy the `.opar` file to the host where Oracle Management Service (OMS) is running. Import the plugin into OEM and deploy the plugin on the server hosting OEM, via the following commands:

   ```
   $emcli import_update -file=/home/guy/
   12.1.0.9.0;_oracle.nosql.snab;_2000_0.opar -omslocal
   ```

3. Deploy the plug-in to the Oracle Management Service (OMS). You can deploy multiple plug-ins to an OMS instance in graphical interface or command line interface. See *Deploying Plug-Ins to Oracle Management Service* in the *Oracle Enterprise Manager Cloud Control Administrator's Guide*.

   CLI Example:

   ```
   $emcli deploy_plugin_on_server -plugin
   =oracle.nosql.snab:12.1.0.9.0 -sys_password=password
   ```

4. Deploy the agent on the server hosting Oracle NoSQL Database. See Deploying Agent .

5. Deploy the plug-in to the EM Agents where Oracle NoSQL Database components are running. See step 4 in *Deploying Plug-Ins on Oracle Management Agent* in the *Oracle Enterprise Manager Cloud Control Administrator's Guide*.

CLI Example:

```
$emcli deploy_plugin_on_agent -agent_names=agent1.example.com:3872;
agent2.example.com:3872 -plugin=oracle.nosql.snab:12.1.0.9.0
```

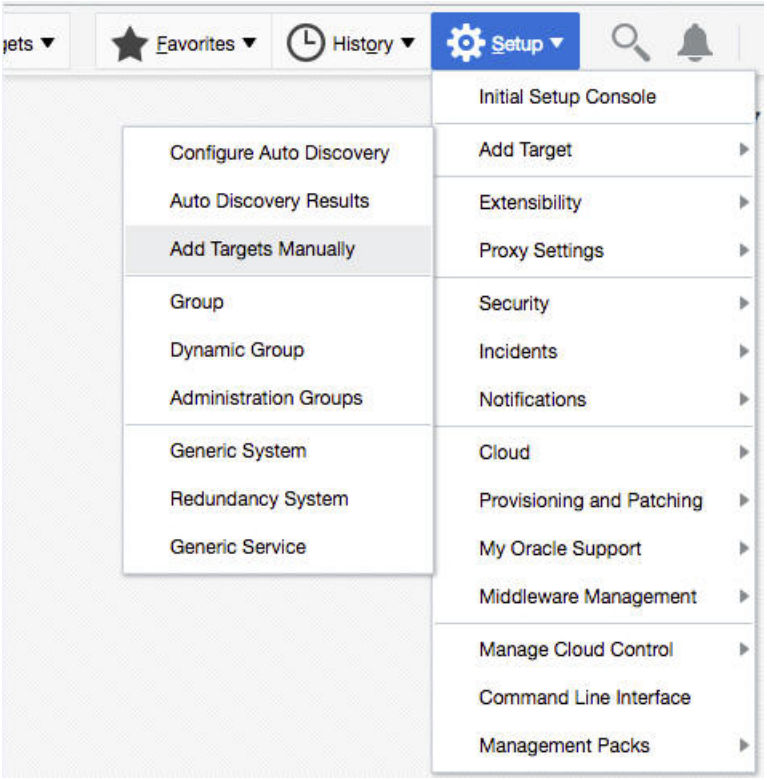
6. Add Oracle NoSQL Database targets. See Adding NoSQL Database Targets .

> **Note:**
>
> The plugin components are now installed and can be configured.

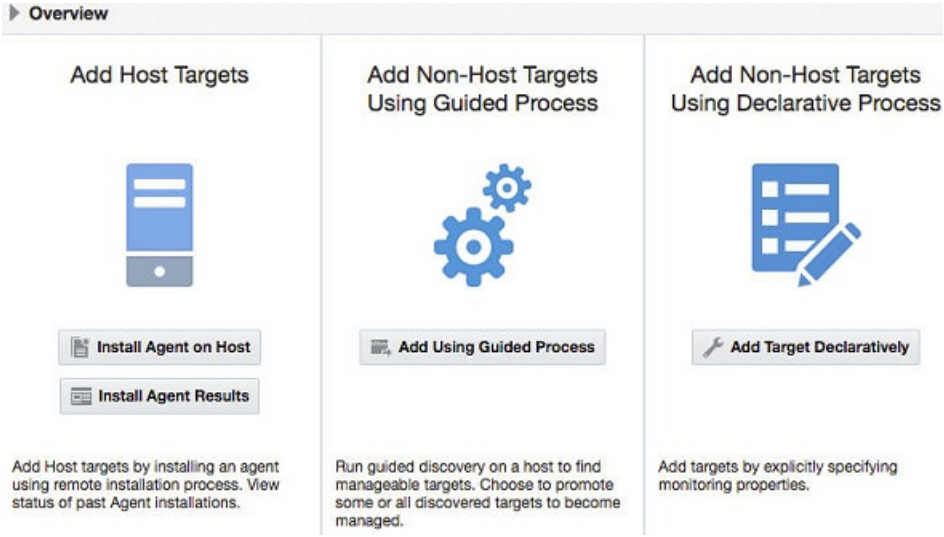
Deploying Agent

Follow the steps below to deploy agent on the server hosting Oracle NoSQL Database:

1. Click `Setup -> Add Target -> Add Targets Manually` on Oracle Enterprise Manager Cloud.

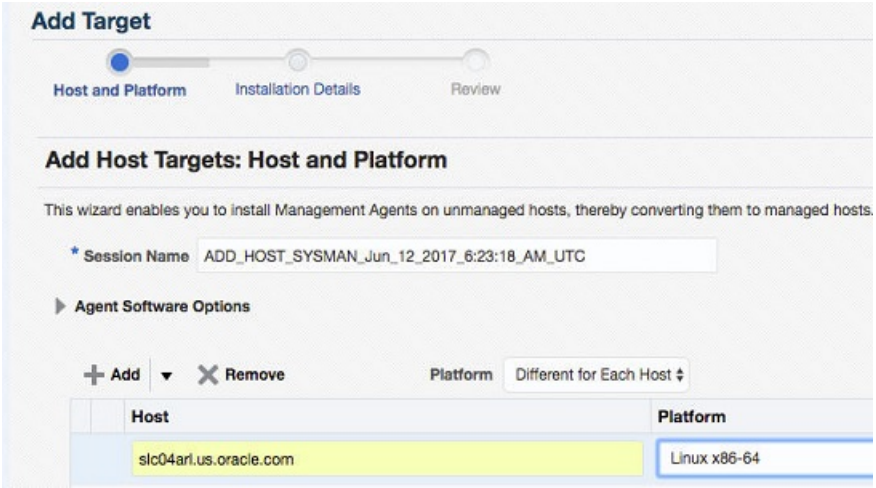


2. Click `Install Agent on Host`.

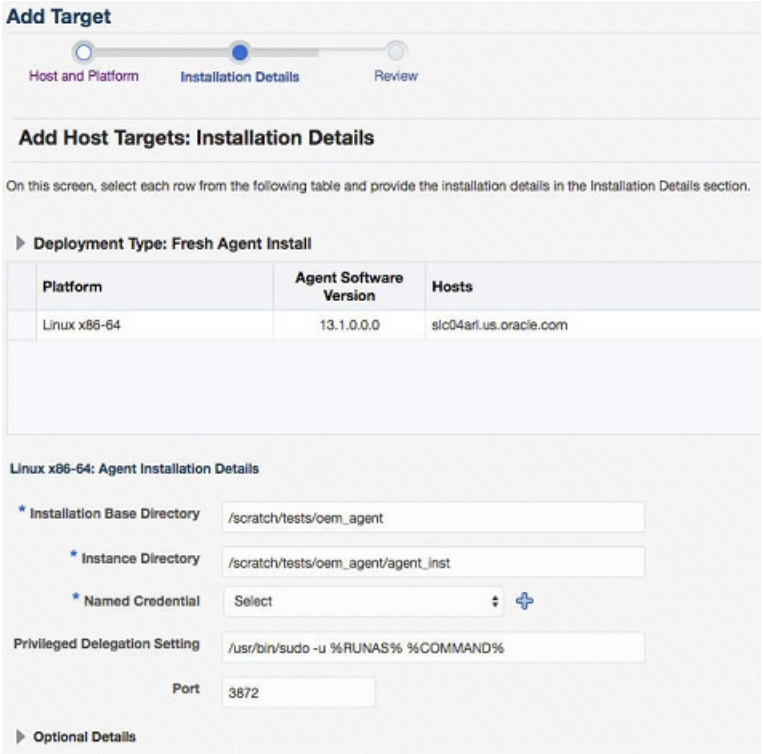


Installing Agent on Host

3. Add the host name of the machine running Oracle NoSQL Database and select the operating system type and click `Next`.



4. Enter the directory where agent files should be stored and the credential information to login to the machine and click `Next`.



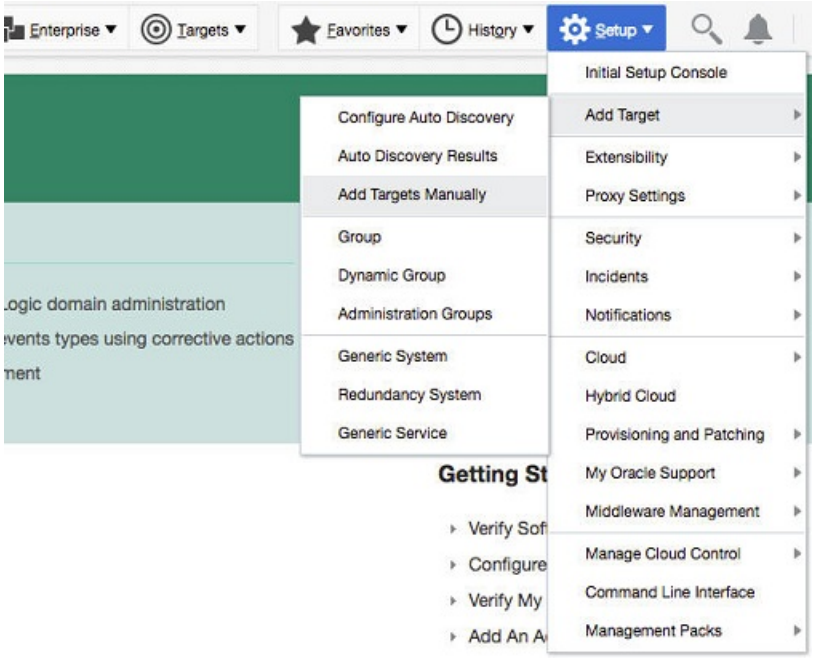
5. Click `Deploy`.

Adding NoSQL Database Targets

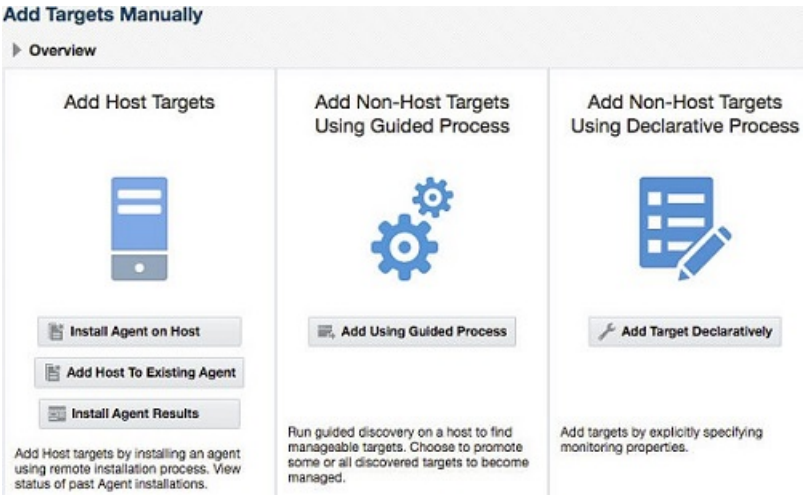
Run the plug-in's discovery program on each host where a Storage Node Agent (SNA) is running, to add the components of a store as monitoring targets.

Follow the steps below to add NoSQL Database targets:

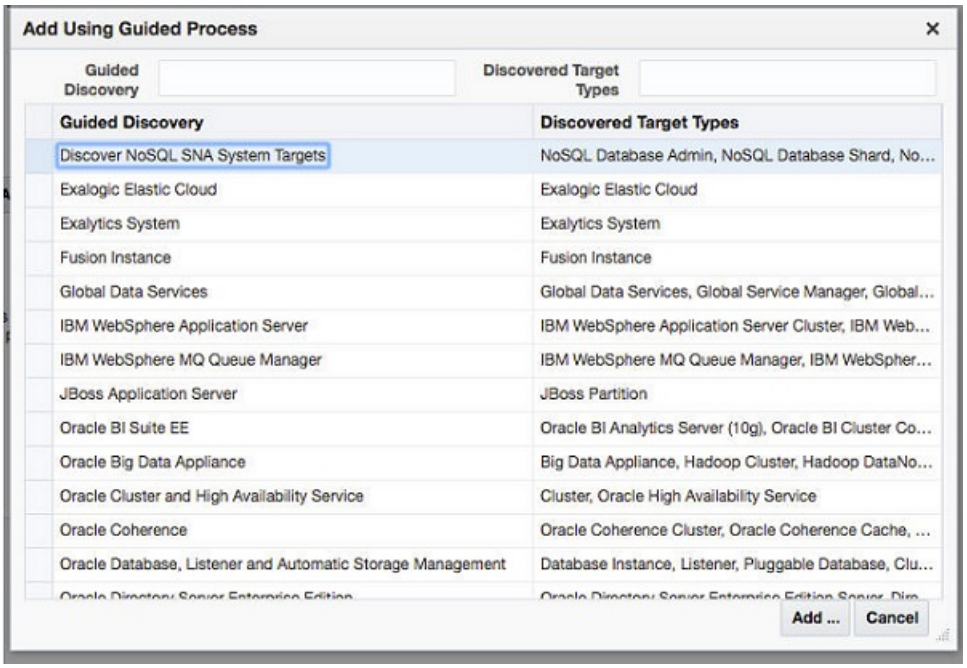
1. Select “Add Targets” from the “Setup” menu, then choose “Add Targets Manually”.



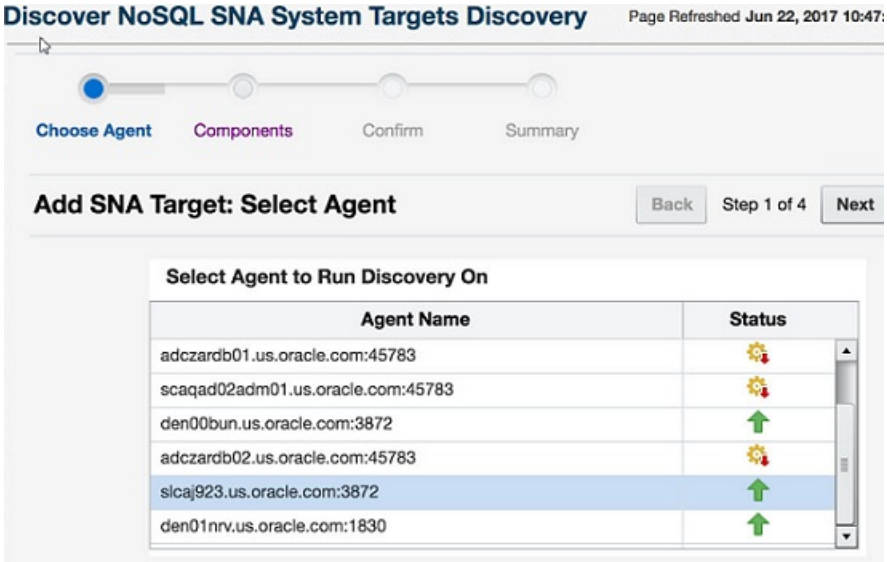
2. Select “Add Using Guided Process” on the “Add Targets Manually” page.



3. Select “Discover NoSQL SNA System Targets” in the “Target Types” drop-down list.

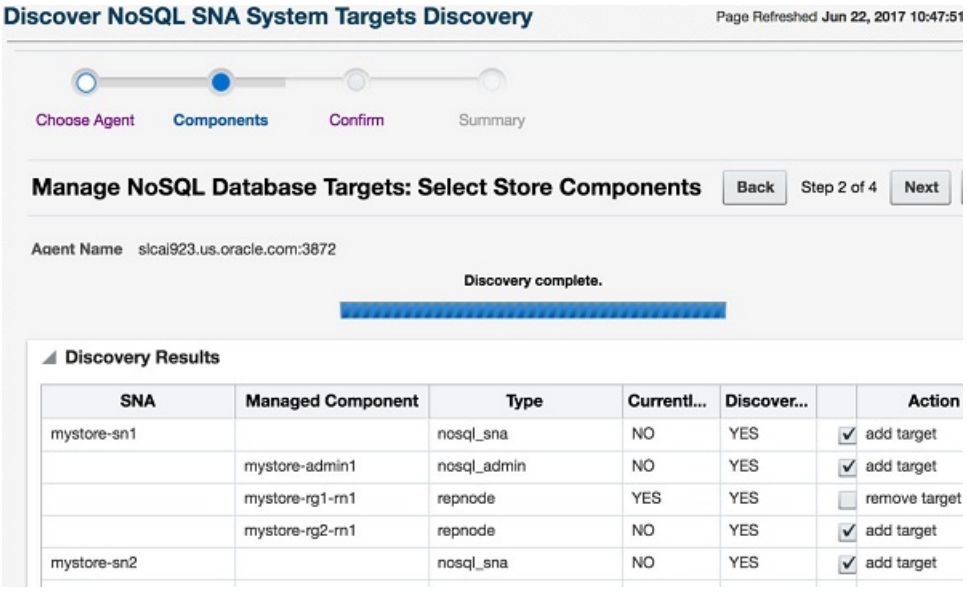


4. Select an agent on which you can run the discovery program. "Choose Agent" (Select Agent to Run Discovery On) in the first page of the program displays a list of all available EM agents.



Select Agent to Run Discovery On

5. Click “Next”. This takes you to the “Components” (Manage NoSQL Database Targets: Select Store Components) page. This shows all the NoSQL Database components that were found on the agent's host. To be found, an SNA must be running when the discovery program runs. The SNA's configuration reports the SNA's managed components, such as Replication Nodes and Admins.



For each component, two attributes are listed:

- Discovered
- Currently Managed

Each attribute can have a value of “Yes” or “No”. For each component found, one of two actions is available:

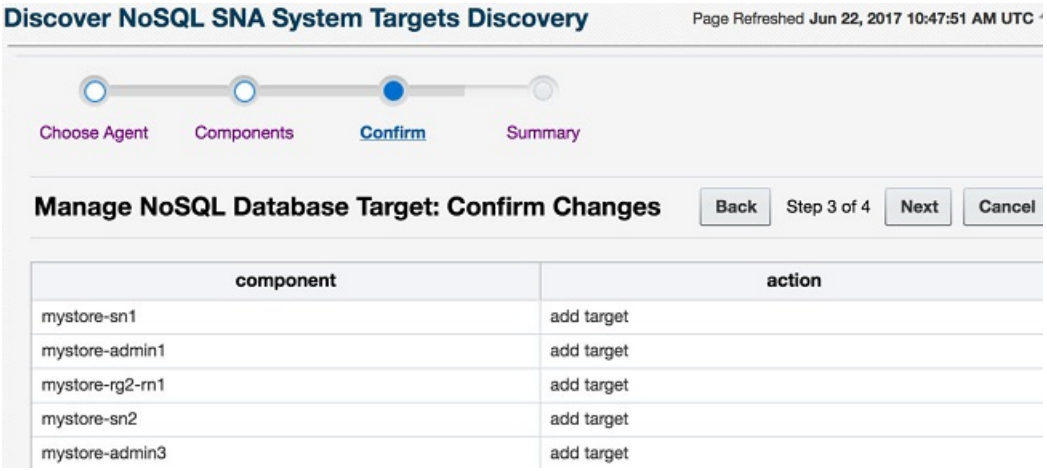
- add target
- remove target

The action is enabled by means of a check box. The recommended action for a component is shown by the state of its check box.

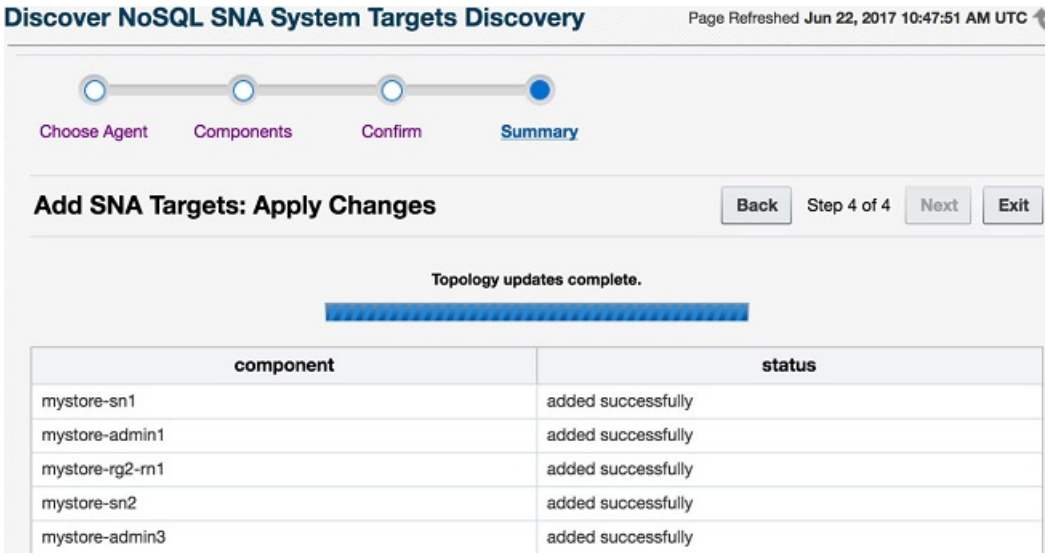
- If the box is checked, then the action is recommended. The user can override the recommended action by checking or un-checking the box.
- If a component has Discovered = YES, it means that an instance of that component was found on the host.
- If a component has Currently Managed = YES, it means that the component is already configured for monitoring by EM.
- If a component is Discovered and not Currently Managed, then it is a candidate for being added as a target. For such components, the available action is "add target", which is the recommended action.
- If a component is Discovered and Currently Managed, it means that the component has already been discovered and added as a monitoring target. For such components, the available action is "remove target", but the recommended action is to do nothing, because the discovery report is in sync with EM's configuration.
- If a component is Currently Managed and not Discovered, it means that EM is configured to monitor a component that was unexpectedly not found on the agent's host. This could be so because the component no longer resides on the host; or it could reflect a temporary unavailability of the Storage Node Agent. For such components, the recommended action is "remove target".

> **Note:**
>
> In most cases, the default recommended action is the correct action, and no checkboxes need to be altered.

6. Click “Next” once the desired configuration is set up on the “Components” page. This takes you to the “Confirm” (Manage NoSQL Database Targets: Confirm Changes) page, which shows a list of all the chosen actions.



7. Click “Next” to go to the “Summary” (Add SNA Targets: Apply Changes) page. This shows a report of success or failure of each action.



8. At this point, you may exit Discovery, or you may click on "Choose Agent", near the top of the page, to return to the first page of the program, to re-start and run discovery on a new agent.

   Once all of the components of a store have been discovered and added, EM's model of the store's topology is complete.

Components of a NoSQL Store

Components of a NoSQL Database Store include the Store itself, Storage Node Agents, Replication Nodes, Admins, and Shards. Of these, Stores and Shards are abstract components that do not correspond to a particular service running on a host. Shards are implied by the existence of Replication Nodes that implement them, and a Store is implied by the existence of the components that belong to it. These components are discovered when components that imply their existence are discovered.

For example, the first time discovery is run on an agent where components belonging to a Store are running, the Store itself will appear as a Discovered and not Managed component to be added. After the Store is added, subsequent runs of discovery on other agents where the existence of the Store is implied will show that the Store is Discovered and Managed, with a recommended action to do nothing. Newly discovered components belonging to the Store will be added as Members of the Store.

Likewise, Shards may appear to be discovered on multiple hosts, but a single Shard need be added only once.

Store Targets

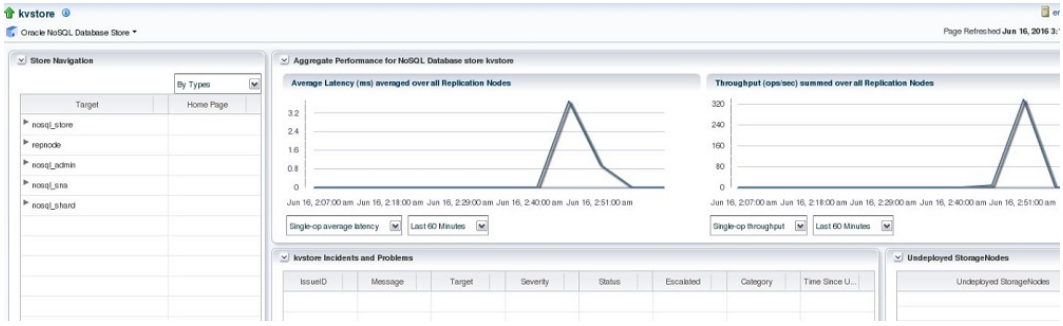
The name of the Store target will be the name of the Store as configured in the NoSQL Database CLI's "configure" command. For more information, see configure. This name must be unique across all instances of NoSQL Database that will be monitored by a given instance of EM.

Member components of the store have target names made up of their component IDs appended to the store name. For example, in a store named myStore, a Storage Node Agent with an id of "sn1" will have the target name "myStore-sn1", a Replication Node with an id of "rg1-rn1" will have the target name "myStore-rg1-rn1", and so forth. The undeployed StorageNodes will be "UNREGISTERED-hostname-port", for example, “UNREGISTERED-example1.example.com-5050". Once the components of a store have been added, you can find the page representing the store by searching for the store name in the “Search Target Name” box in the upper right part of EM's home page. You can also find it via `Targets->All Targets` or `Targets->Systems`.

Store Page

Clicking on the Store's name in any of the lists will take you to the Store's target page.

- The page has two large graphs showing the:
  - Average Latency Averaged over all Replication Nodes in the Store
  - Total Throughput for all Replication Nodes

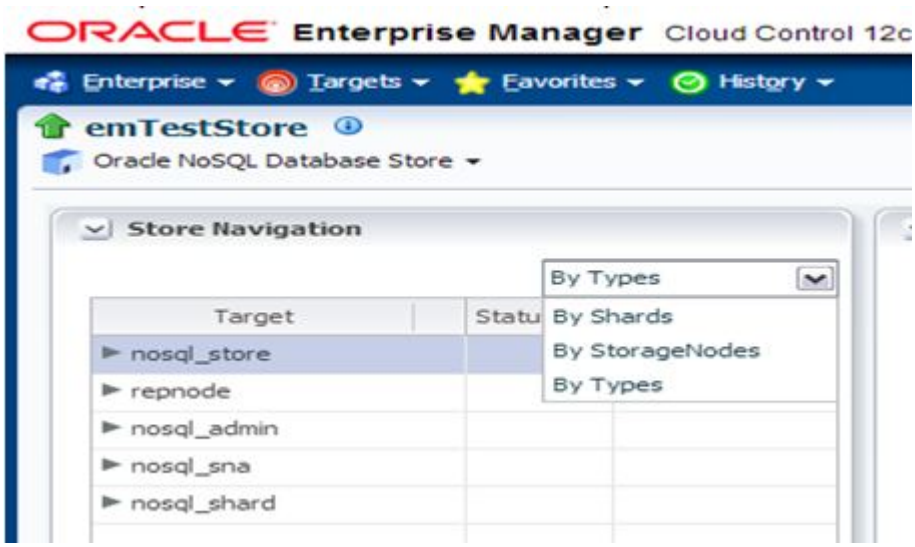


- In the lower right is a list of “Undeployed Storage Node” agents.
- In the lower middle is a list of “Incidents and Problems” related to the store.
- On the left side of the page is the “Store Navigation” panel. This panel presents the topology of the store in three different ways:
  - Types

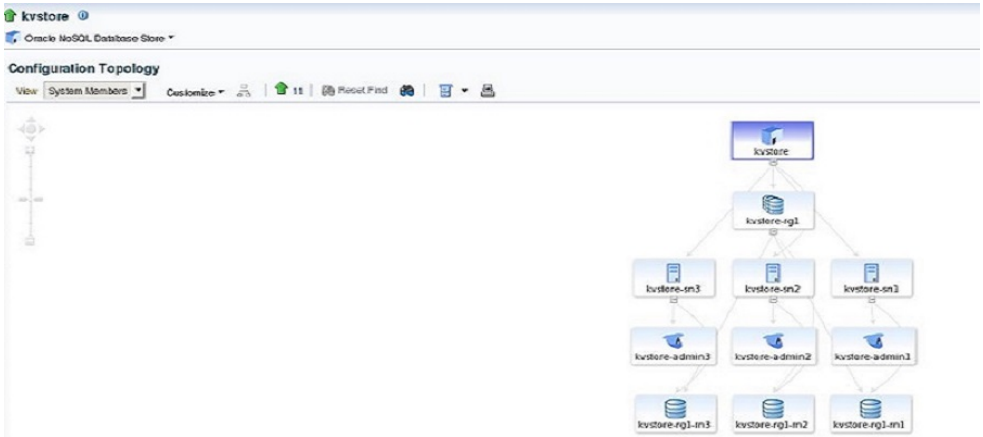
    “By Types” groups targets by their target types; so all Replication Nodes are listed together, all Storage nodes are together, and so forth.
  - StorageNodes

    “By StorageNodes” shows the hierarchy of `Store->Storage Node->Replication Node`. Each Replication Node is managed by a Storage Node Agent, and always resides on the same host. A Storage Node Agent may manage more than one Replication Node, and this is reflected in the tree layout of the navigation panel.
  - Shard

    “By Shards” shows the hierarchy of `Store->Shard->Replication Node`.



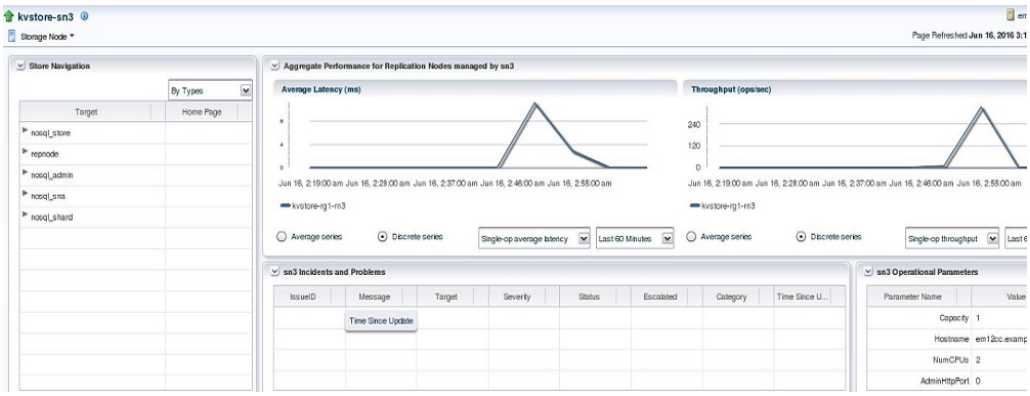
- Each component in the navigation panel has a status "up" or "down", or "unknown" and a link to the target page (labeled "Home Page") for that component. The status can be "unknown" if the targets have yet to be reported for the first time, or if OMS cannot contact the EM Agent.
- The “Store” page, (under menu item Members->Topology) shows the layout of the store as a graph, which is an overview of the "Configuration Topology".



Storage Node Page

On the “Storage Node” page, you see the same graphs of “Average Latency” and “Throughput”. Here, the graphs show aggregate information for the Replication Nodes that are managed by the Storage Node. The graphs display either discrete series, with one line per Replication Node; or it can combine the series into a single line. The choice is made using the radio buttons at the bottom of the graph.

This page also shows the operational parameters for the Storage Node Agent.

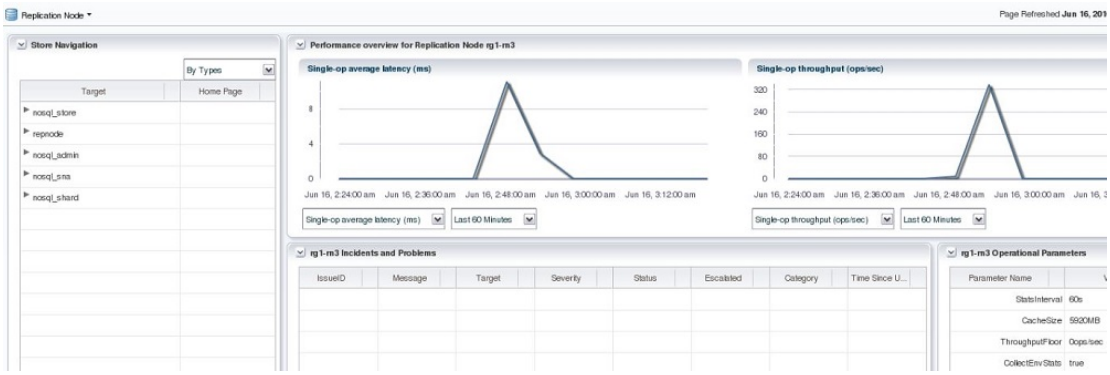


Shard Page

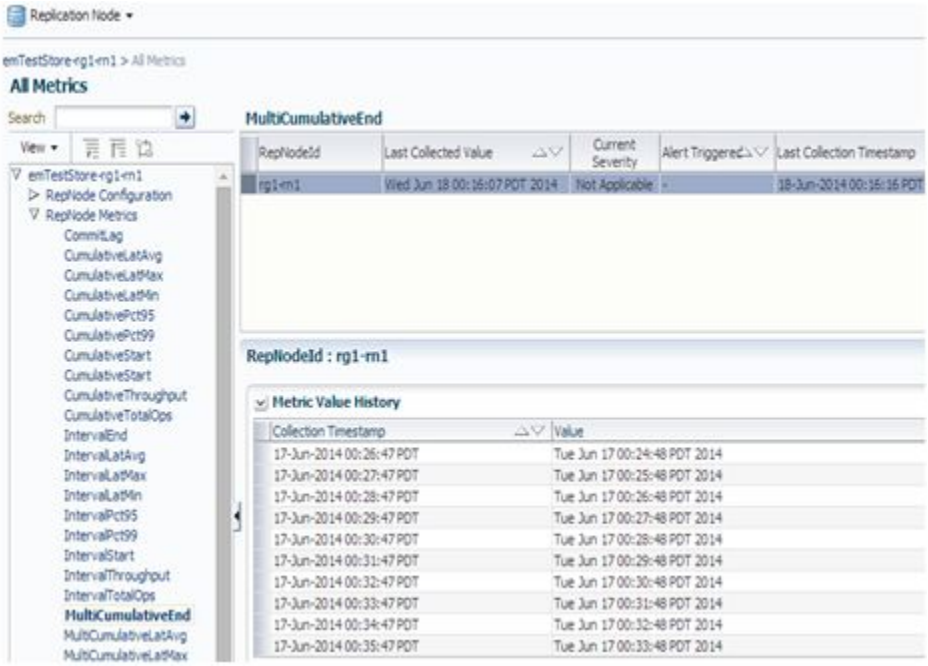
The “Shard” page is similar to the "Storage Node" page. It shows metrics collected from multiple Replication Nodes. These Replication Nodes are the ones that serve as the replication group for the Shard.

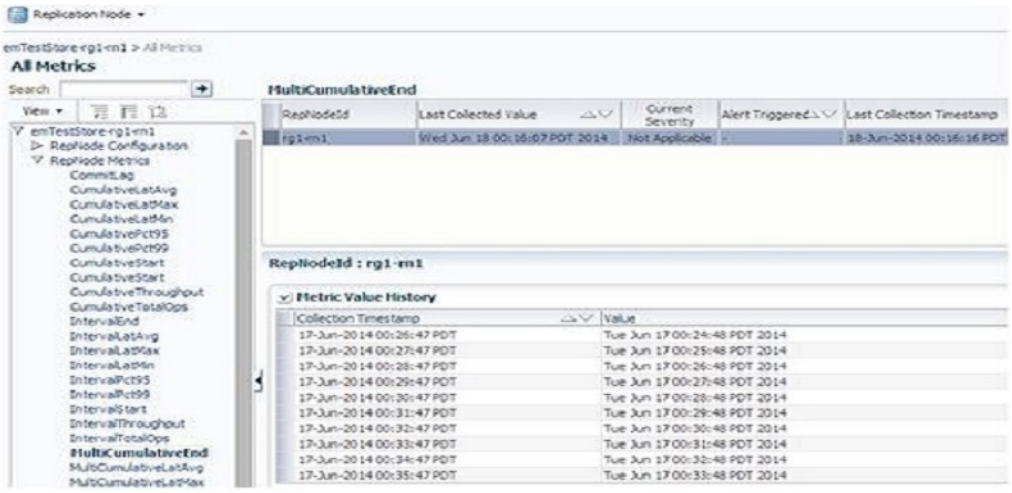
Replication Node Page

The “Replication Node” page shows metrics for a single replication node. This page also shows the Replication Node's operational parameters.



On this page you can view a variety of metrics by choosing `Replication Node->Monitoring->All Metrics`. It also shows different time ranges.





From here you can examine and graph any of the numerous metric values that are collected from the Replication Node.

About IntelliJ Plugin

Browse tables and execute queries on your Oracle NoSQL Database KVStore from IntelliJ.

The Oracle NoSQL Database IntelliJ plugin connects to a running instance of Oracle NoSQL Database KVStore and allows you to:

- View the tables in a well-defined tree structure with Table Explorer.
- View information on columns, indexes, primary key(s), and shard key(s) for a table.

- View column data in a well-formatted JSON Structure.
- Create tables using form-based schema entry or supply DDL statements.
- Drop tables.
- Add new columns using form-based entry or supply DDL statements.
- Drop Columns.
- Create Indexes.
- Drop Indexes.
- Execute SELECT SQL queries on a table and view query results in tabular format.
- Execute DML statements to update, insert, and delete data from a table.

Topics:

- Setting Up IntelliJ Plug-in
- Creating a NoSQL Project in IntelliJ
- Connecting to Oracle NoSQL Database from IntelliJ
- Managing Tables Using the IntelliJ Plugin

Setting Up IntelliJ Plug-in

Learn how to set up the IntelliJ plug-in for Oracle NoSQL Database KVStore.

Perform the following steps:

1. Download and extract Oracle NoSQL Java SDK. See About Oracle NoSQL Database SDK Drivers.
2. Install the IntelliJ plugin, and restart the IDE.

   You have two options to install the plugin:

   - Search the Oracle NoSQL Database Connector in the JetBrains plug-in repository, and install it, or
   - Download the IntelliJ plugin from Oracle Technology Network, and install the plugin from disk.

   > **Tip:**
   >
   > Don't extract the downloaded plugin zip file. Select the plugin in the zip format while installing it from disk.

After you successfully set up your IntelliJ plugin, create a NoSQL project, and connect it to your Oracle NoSQL Database KVStore.

Creating a NoSQL Project in IntelliJ

Learn how to create a NoSQL project in IntelliJ.

Perform the following steps:

1. Open IntelliJ IDEA. Click **File** > **New** > **Project**.
2. Enter a value for **Project Name** and **Project Location**, and click **Create**.

3. Once your NoSQL project is created, you can browse the example java files from the Project Explorer window.

After you successfully create a NoSQL project in IntelliJ, connect your project to your Oracle NoSQL Database KVStore.

Connecting to Oracle NoSQL Database from IntelliJ

Learn how to connect your NoSQL project to Oracle NoSQL Database KVStore using the IntelliJ plugin.

Prerequisites:
To create a successful connection to your Oracle NoSQL Database KVStore, ensure that:

- The KVStore is deployed and running.
- The Oracle NoSQL Database Proxy is started. See Configuring the Proxy. Starting the release 19.5, Proxy is bundled along with the Oracle NoSQL Database download package.

Perform the following steps:

1. Open your NoSQL project in IntelliJ.
2. Click the **wrench** icon in the **Schema Explorer** window to open the **Settings** dialog for the plugin.
3. Expand **Tools** > **Oracle NoSQL** in the Settings Explorer, and click **Connections**.
4. Select **Onprem** from the drop-down menu for the connection type.
5. Enter values for the following connection parameters, and click **OK**.

Table 5-5 Connection Parameters

Parameter	Description
Proxy URL	`http://<proxy_host>:<proxy_http_port>` or `https://<proxy_host>:<proxy_http_port>` where: • `http` or `https` indicates the store security. For a secure KVStore, the proxy URL begins with `https`. • `proxy_host` is the host name of the machine to host the proxy service. This should match the host you configured earlier. See Configuring the Proxy.
SDK Path	Complete path to the directory where you extracted the Oracle NoSQL Java SDK. For example, `D:\oracle-nosql-java-sdk-5.2.11`

Table 5-5 (Cont.) Connection Parameters

Parameter	Description
Security	Select **SSL** for secure KVStores. In case, you are creating connection to a non-secure KVStore, select **None**. The default value is **SSL**. **Note:** In case of secure KVStores, the proxy URL must begin with `https`.
Username	User name to connect to the secure store. This value is required only if you select **SSL** for the Security parameter.
Password	Password to connect to the secure store. This value is required only if you select `SSL` for the Security parameter.
TrustStore	Browse to the location where the certificate trust file is placed. See Using the Proxy in a secure data store.

6. The Intellij plugin connects your project to the Oracle NoSQL Database KVStore and displays its schema in the Schema Explorer window.

After you successfully connect your project to your Oracle NoSQL Database KVStore, you can manage the tables and data in your schema.

Managing Tables Using the IntelliJ Plugin

Learn how to create tables and view table data in Oracle NoSQL Database KVStore from IntelliJ.

After connecting to the Oracle NoSQL Database, you can execute the examples downloaded with Oracle NoSQL Java SDK to create a sample table. With the help of the IntelliJ Plugin, you can view the tables and their data in the Schema Explorer window.

Execute an example program:

1. Open the NoSQL project connected to your Oracle NoSQL Database.
2. Locate and click `BasicTableExample` in the Project Explorer window. You will find this in the **examples** folder under `oracle-nosql-java-sdk`. By looking at the code, you can notice that this program creates a table called `audienceData`, puts two rows into this table, queries the inserted rows, deletes the inserted rows, and finally drops the `audienceData` table.
3. To pass the required arguments, click **Run** > **Edit Configurations**. Enter the following program arguments, and click **OK**.

Table 5-6 Program Arguments

Program Arguments	More Information
`http://<proxy_host>:<proxy_http_port> -useKVProxy`	For example, if your Proxy URL is `http://<proxy_host>:8080`, the program argument must be `http://<proxy_host>:8080 -useKVProxy`.

4. To execute this program, click **Run** > **Run 'BasicExampleTable'** or press **Shift + 10**.
5. Verify the logs in the terminal to confirm that the code executed successfully. You can see the display messages that indicate table creation, rows insertion, and so on.

> **Tip:**
>
> As the `BasicExampleTable` deletes the inserted rows and drops the `audienceData` table, you can't view this table in the Schema Explorer. If you want to see the table in the Schema Explorer, comment the code that deletes the inserted rows and drops the table, and rerun the program.

6. To view the tables and their data:
   a. Locate the Schema Explorer, and click the **Refresh** icon to reload the schema.
   b. Locate the `audienceData` table under your tenant identifier, and expand it to view its columns, primary key, and shard key details.
   c. Double-click the table name to view its data. Alternatively, you can right-click the table and select **Browse Table**.
   d. A record viewer window appears in the main editor. Click **Execute** to run the query and display table data.
   e. To view individual cell data separately, double-click the cell.

Perform DDL operations using IntelliJ

You can use IntelliJ to perform DDL operations.

Some of the DDL operations that can be performed from inside the IntelliJ plugin are

- CREATE TABLE
- DROP TABLE
- CREATE INDEX
- DROP INDEX
- ADD COLUMN
- DROP COLUMN

CREATE TABLE

- Locate the Schema Explorer, and click the **Refresh** icon to reload the schema.
- Right click the connection name and choose **Create Table**.

- In the prompt, enter the details for your new table. You can create the Oracle NoSQL Database table in two modes:
  - **Simple DDL Input** : You can use this mode to create the table declaratively, that is, without writing a DDL statement.
  - **Advanced DDL Input** : You can use this mode to create the table using a DDL statement.
- You have the option to view the DDL statement before creating. Click **Show DDL** to view the DDL statement formed based on the values entered in the fields in the Simple DDL input mode. This DDL statement gets executed when you click Create.
- Click **Create** to create the table.

DROP TABLE

- Locate the Schema Explorer, and click the Refresh icon to reload the schema.
- Right click on the table that you want to drop. Choose **Drop Table**.
- A confirmation window appears, click **Ok** to confirm the drop action.

CREATE INDEX

- Locate the Schema Explorer, and click the Refresh icon to reload the schema.
- Right click on the table where index need to be created. Choose **Create Index**.
- In the Create Index panel, enter the details for creating an index without writing any DDL statement. Specify the name of the index and the columns to be part of the index.
- Click **Add Index**.

DROP INDEX

- Locate the Schema Explorer, and click the Refresh icon to reload the schema.
- Click on the target table to see the listed columns, Primary Keys, Indexes and Shard Keys.
- Locate the target-index which has to be dropped and right-click on it. Click **Drop Index**.
- A confirmation window appears, click **Ok** to confirm the drop action.

ADD COLUMN

- Locate the Schema Explorer, and click the Refresh icon to reload the schema.
- Right click on the table where column needs to be added. Choose **Add Column**.
- You can add new COLUMNs in two modes:
  - Simple DDL Input : You can use this mode to add new columns without writing a DDL statement.
  - Advanced DDL Input : You can use this mode to add new columns into the table by supplying a valid DDL statement.
- In both the modes, specify the name of the column and define the column with its properties - datatype, default value and whether it is nullable.
- Click **Add Column**.

DROP COLUMN

- Locate the Schema Explorer, and click the Refresh icon to reload the schema.
- Click on the target table to see the listed columns, Primary Keys, Indexes and Shard Keys.
- Locate the target-column which has to be dropped and right-click on it. Click **Drop Column.**
- A confirmation window appears, click **Ok** to confirm the drop action.

Perform DML operations using IntelliJ

You can add data, modify existing data and query data from tables using IntelliJ plugin.

Insert data

- Locate the Schema Explorer, and click the Refresh icon to reload the schema.
- Right click on the table where a row needs to be inserted. Choose **Insert Row**.
- In the Insert Row panel, enter the details for inserting a new row. You can INSERT a new ROW in two modes:
  - Simple Input : You can use this mode to insert the new row without writing a DML statement. Here a form based row fields entry is loaded, where you can enter the value of every field in the row.
  - Advanced JSON Input : You can use this mode to insert a new row into the table by supplying a JSON Object containing the column name and its corresponding value as key-value pairs.
- Click **Insert Row**.

Modify Data - UPDATE ROW/DELETE ROW:

- Locate the Schema Explorer, and click the Refresh icon to reload the schema.
- Right click on the table where a row needs to be inserted. Choose **Browse Table.**
- In the textbox on the left, enter the SQL statement to fetch data from your table. Click **Execute** to run the query.
- To view individual cell data separately, click the table cell.
- To perform DML operations like Update and Delete Row, right-click on the particular row. Pick your option from the context-menu that appears.
  - Delete Row : A confirmation window appears, click **Ok** to delete the row.
  - Update Row : A separate HTML panel opens below the listed rows, containing the column names and its corresponding value in a form-based entry and as a JSON key-pair object. You can choose either of the two methods and supply new values.

> **Note:**
>
> In any row, PRIMARY KEY and GENERATED ALWAYS AS IDENTITY columns cannot be updated.

Query tables

- Locate the Schema Explorer, and click the Refresh icon to reload the schema.
- Right click on the table and choose **Browse Table**.
- In the textbox on the left, enter the SELECT statement to fetch data from your table.
- Click **Execute** to run the query. The corresponding data is retrieved from the table.
- Right click on any row and click View JSON to view the entire row object in the JSON format.
- Click **Show Query Plan** to view the execution plan of the query.

About Eclipse plugin

Build and run your Oracle NoSQL Database applications quickly from the Eclipse IDE.

To enhance your experience of building an Oracle NoSQL Database application, a plugin is available in Eclipse. This plugin connects to a running instance Oracle NoSQL Database KVStore and allows you to:

- Quickly get started with Oracle NoSQL Database by using the examples available with the plugin.
- Explore development/test date from tables in your Oracle NoSQL Database KVStore.
- Build and test your queries.
- Retrieve columns, indexes, primary keys, and shard keys for each table.
- Build and test your SQL queries on a table and obtain results in a tabular format.
- View the date in each column in the JSON format.

To use the Eclipse plugin:

1. Download the eclipse plugin from Oracle Technology Network.
2. Follow the instructions given in the **README** file and install the plugin.
3. After installing the Eclipse plugin, you can connect to your Oracle NoSQL Database KVStore and execute the code to read/write the tables. For more details, you can access the help content embedded within Eclipse.
   To access the help content:
   a. Click **Help Contents** from the **Help** menu.
   b. Locate and expand the **Oracle NoSQL Plugin Help Contents** section. This lists all the help topics available for Oracle NoSQL Plugin.
   c. Refer the help topic as per your requirement.

> **Note:**
>
> The Oracle NoSQL Database Eclipse plugin works with Eclipse Neon 4.6 and later.

About Oracle NoSQL Database Visual Studio Code Extension

The Oracle NoSQL Database provides an extension for Microsoft Visual Studio Code which lets you connect to a running instance of Oracle NoSQL Database.

You can use Oracle NoSQL Database Visual Studio (VS) Code extension to:

- View the tables in a well-defined tree structure with Table Explorer.
- View information on columns, indexes, primary key(s), and shard key(s) for a table.
- View column data in a well-formatted JSON Structure.
- Create tables using form-based schema entry or supply DDL statements.
- Drop tables.
- Add new columns using form-based entry or supply DDL statements.
- Drop Columns.
- Create Indexes.
- Drop Indexes.
- Execute SELECT SQL queries on a table and view query results in tabular format.
- Execute DML statements to update, insert, and delete data from a table.
- Download the Query Result after running the SELECT query into a JSON file.
- Download each row of the result obtained after running the SELECT query into a JSON file.

Installing Oracle NoSQL Database Visual Studio Code Extension

You can install the Oracle NoSQL Database VS Code extension in two ways. Install from the Visual Studio Marketplace for online installation or install from the VSIX package using `*.vsix` file for offline installation.

Before you can install the Oracle NoSQL Database Visual Studio (VS) Code extension, you must install Visual Studio Code. You can download Visual Studio Code from here.

- Install from Visual Studio Marketplace
- Install from a VSIX

Install from Visual Studio Marketplace

1. In Visual Studio Code, click the **Extensions** icon in the left navigation.

Alternatively, you can open the **Extensions** view by pressing:

- (Windows and Linux) Control + Shift + X
- (macOS) Command + Shift + X.

2. Search Oracle NoSQL Database Connector in the extension marketplace.
3. Click Install on the Oracle NoSQL Database Connector extension

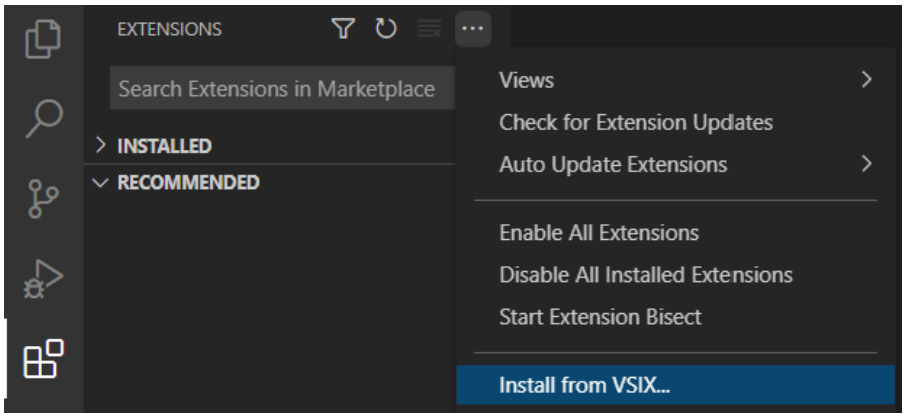
Install from a VSIX

1. Download the VSIX file for Oracle NoSQL Database from Oracle NoSQL Database Downloads site.
2. In Visual Studio Code, click the **Extensions** icon in the left navigation.

Alternatively, you can open the **Extensions** view by pressing:

- (Windows and Linux) Control + Shift + X
- (macOS) Command + Shift + X.

3. In the **Extensions** view, Click the **More Actions (...)** menu and then click **Install from VSIX...**.



4. Browse to the location where the `*.vsix` file is stored and click **Install**.

Connecting to Oracle NoSQL Database from Visual Studio Code

Oracle NoSQL Database Visual Studio (VS) Code extension provides two methods to connect to Oracle NoSQL Database. You can either provide a config file with the connection information or fill in the connection information in the specific fields.

- Fill in Individual Fields
- Connect via Config File

Fill in Individual Fields

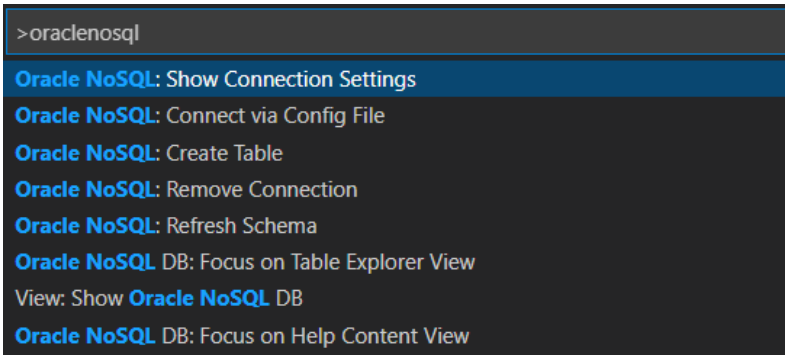
1. In Visual Studio Code, click the **Oracle NoSQL DB** view in the **Activity Bar**.

2. Open the Oracle NoSQL DB **Show Connection Settings** page from the Command Palette or the **Oracle NoSQL DB** view in the **Activity Bar**.
   - Open from Command Palette

a. Open the **Command Palette** by pressing:
   - (Windows and Linux) Control + Shift + P
   - (macOS) Command + Shift + P

b. From the Command Palette, select **OracleNoSQL: Show Connections Settings**.

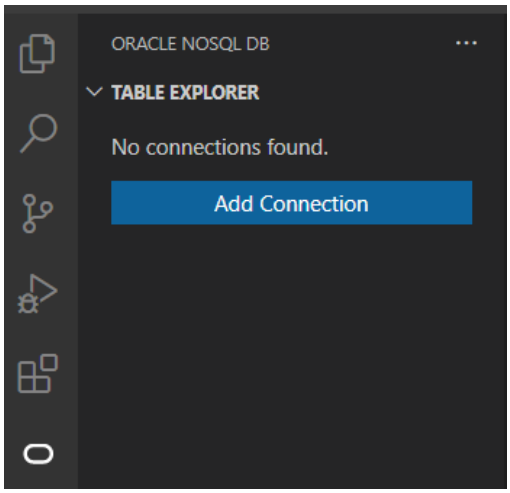
> **Tip:**
>
> Enter `oraclenosql` in the Command Palette to display all of the Oracle NoSQL DB commands you can use.



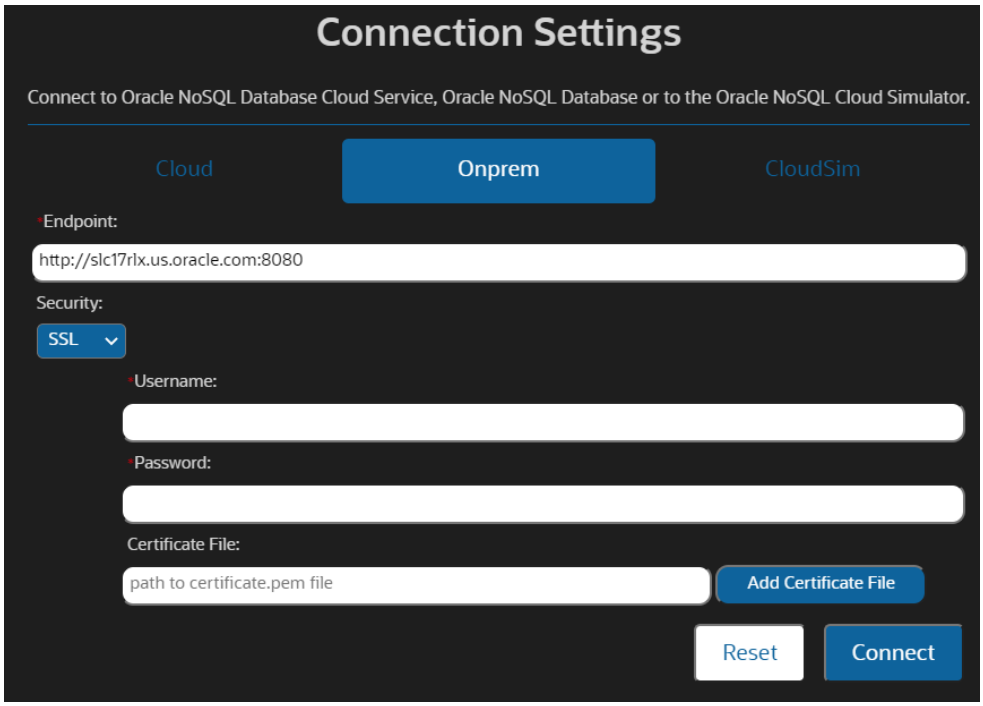
- Open from Oracle NoSQL DB View

  a. Expand the **Schema Explorer** pane in the left navigation if it's collapsed.

  b. Click **Add Connection** to open the Oracle NoSQL DB **Show Connection Settings** page.



3. In the **Show Connection Settings** page, click **Onprem** to connect to Oracle NoSQL Database.



4. Enter the connection information.

Table 5-7 Oracle NoSQL Database Connection Parameters

Field	Description
Endpoint:	Service URL of the Oracle NoSQL Database Proxy. • `http://<proxy_host>:<proxy_http_port>` • `https://<proxy_host>:<proxy_https_port>` where, • `http` or `https` indicates the store security. For a secure KVStore, the proxy URL begins with `https`. • *proxy_host* is the host name of the machine to host the Oracle NoSQL Database Proxy service. For more information, see Configuring the Proxy.
Security:	Select SSL for secure KVStores. In case you are creating the connection to a non-secure KVStore, select None. The default value is `None`.
Username:	User name to connect to the secure store. This value is required only if you select SSL for the Security parameter.

Table 5-7 (Cont.) Oracle NoSQL Database Connection Parameters

Field	Description
Password:	Password to connect to the secure store. This value is required only if you select `SSL` for the Security parameter.
Certificate File:	Browse to the location of the SSL certificate file (for example, `certificate.pem`). See Using the Proxy in a secure data store.

5. Click **Connect**.
6. Click **Reset** to clear the saved connection details from the workspace.

Connect via Config File

1. Create the config file, for example, config.json or a file with the JSON object. The config file format for connecting to Oracle NoSQL Database is as shown below.

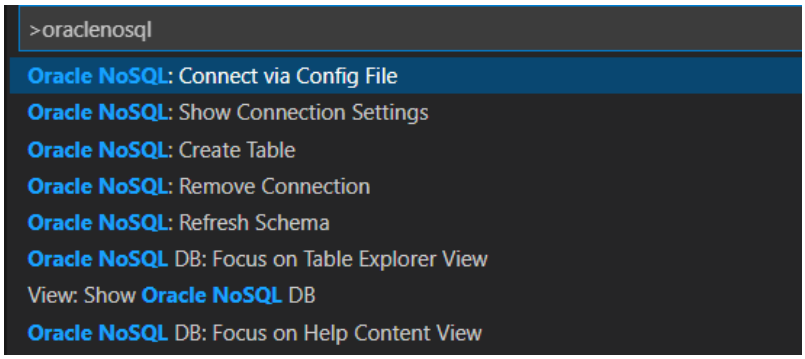
Table 5-8 Configuration Templates

Secure KVStore	Non-secure KVStore

Configuration template to connect using OCI configuration file
 	
`{ "serviceType": "KVSTORE", "endpoint": "<service-URL-of-Oracle-NoSQL-Database-Proxy>", "auth": { "kvstore": { "configFile": "<path-to-OCI-config-file>" } } }`
 	
Configuration template to connect using authentication credentials
 	
`{ "serviceType": "KVSTORE", "endpoint": "<service-URL-of-Oracle-NoSQL-Database-Proxy>", "auth": { "kvstore": { "user": "<username>", "password": "<password>" } } }`
	`{ "serviceType": "KVSTORE", "endpoint": "<service-URL-of-Oracle-NoSQL-Database-Proxy>" }`

2. Open the Command Palette by pressing:
   - (Windows and Linux) Control + Shift + X
   - (macOS) Command + Shift + X
3. From the Command Palette, select **Oracle NoSQL: Connect via Config File**.

> **Tip:**
>
> Enter `oraclenosql` in the Command Palette to display all of the Oracle NoSQL DB commands you can use.



4. Browse to the location where the `*.config` file is stored and click **Select**.

> **Note:**
>
> When you want to connect to a on-premise secure cluster using Visual Studio Code Extension, you need to do one of the following if you have self-signed certificates:
>
> - Add the self-signed certificate to the trusted root store in your Windows configuration (using Microsoft Management Console)).
> - Open Visual Studio Code. Go to File > Preferences > Settings > Application > Proxy. In the **Proxy Support** menu select **off**.

Managing Tables Using Visual Studio Code Extension

Once you connect to your deployment using Oracle NoSQL Database Visual Studio (VS) Code extension, use the **TABLE EXPLORER** located on the left navigation to:

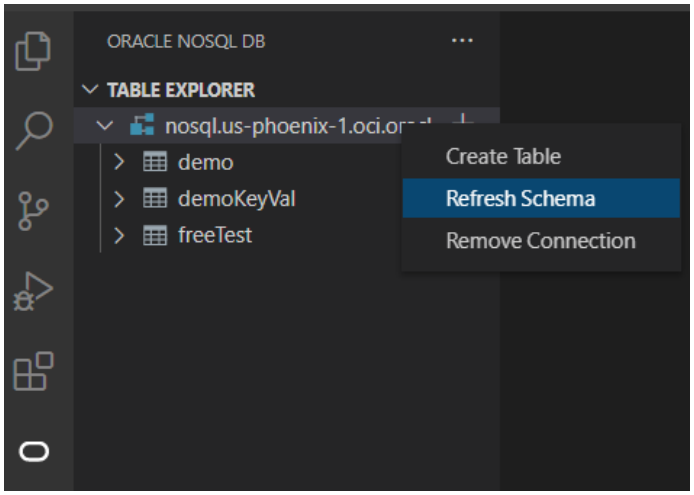
- Explore your tables, columns, indexes, primary keys, and shard keys.
- Create new tables.
- Drop existing tables.
- Create Indexes.
- Drop Indexes.
- Add columns.
- Drop Columns.
- Insert data into table.
- Execute SELECT SQL queries.

Explore tables, columns, indexes and keys

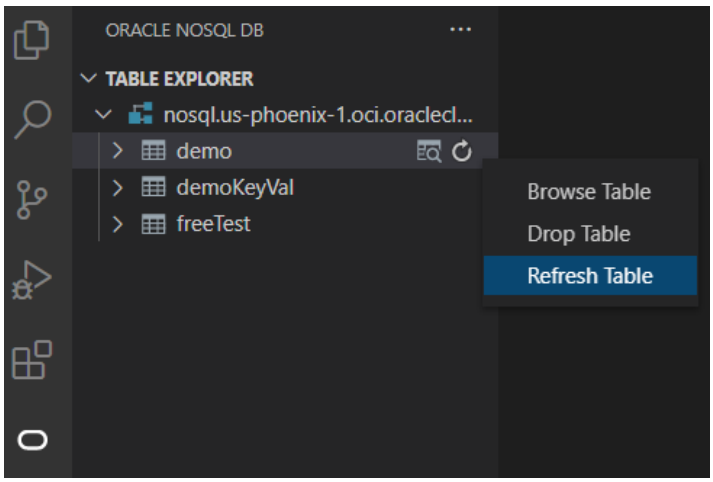
When you expand an active connection, Oracle NoSQL Database VS Code shows the tables in that deployment.

- Click the table name to view its columns, indexes, primary key(s), and shard key(s). The column name displays along with its data type.

- You can refresh the schema or table at any time to re-query your deployment and populate Oracle NoSQL Database with the most up-to-date data.
  - In the **TABLE EXPLORER**, locate the connection and click the Refresh icon to reload the schema. Alternatively, you can right-click the connection and select **Refresh Schema**.



  - In the **TABLE EXPLORER**, locate the table name and click the Refresh icon to reload the table. Alternatively, you can right-click the table name and select **Refresh Table**.



Perform DDL operations using Visual Studio Code

You can use Visual Studio Code to perform DDL operations.

Some of the DDL operations that can be performed from inside the Visual Studio Code plugin are:

- CREATE TABLE
- DROP TABLE
- CREATE INDEX

- DROP INDEX
- ADD COLUMN
- DROP COLUMN

CREATE TABLE

You can create the Oracle NoSQL Database table in two modes:

- **Simple DDL Input**: You can use this mode to create the Oracle NoSQL Database table declaratively, that is, without writing a DDL statement.
- **Advanced DDL Input**: You can use this mode to create the Oracle NoSQL Database table using a DDL statement.

1. Hover over the Oracle NoSQL Database connection to add the new table.
2. Click the Plus icon that appears.
3. In the **Create Table** page, select **Simple DDL Input**.

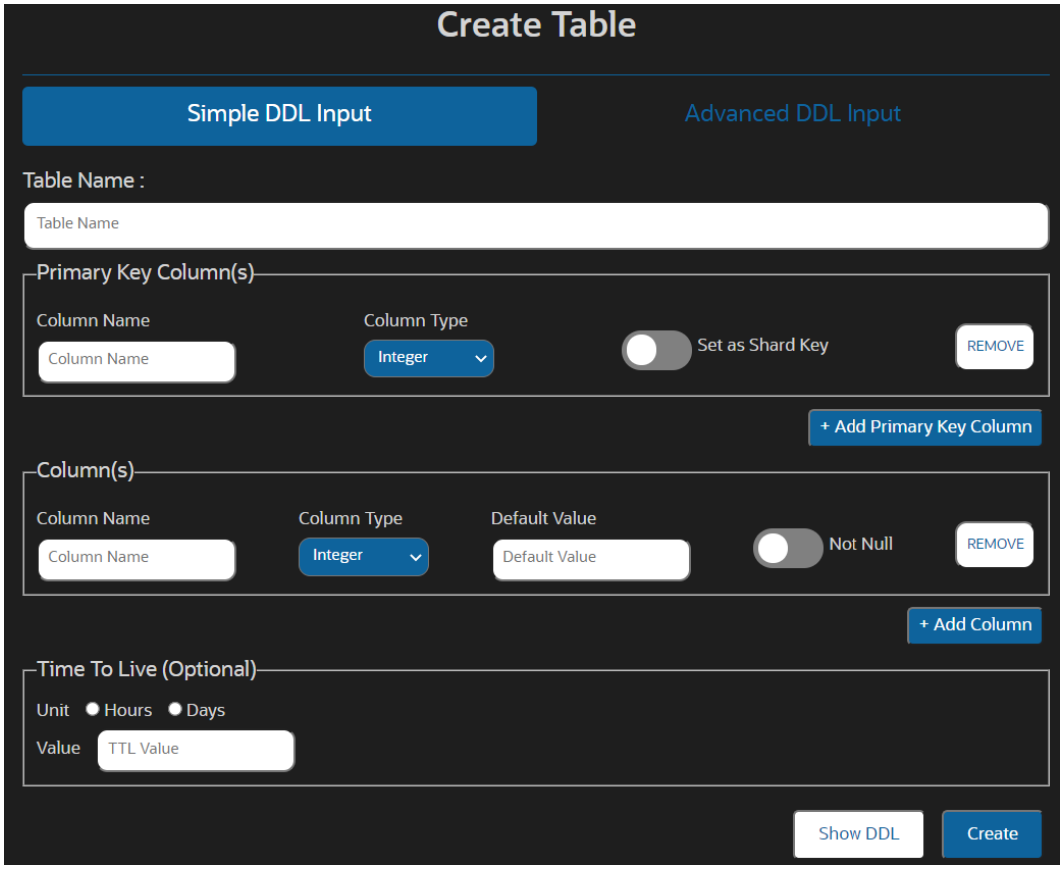


Table 5-9 Create an Oracle NoSQL Database Table

Field	Description
Table Name:	Specify a unique table name.
Column Name	Specify a column name for the primary key in your table.

Table 5-9 (Cont.) Create an Oracle NoSQL Database Table

Field	Description
Column Type	Select the data type for your primary key column.
Set as Shard Key	Select this option to set this primary key column as shard key. Shard key is to distribute data across the Oracle NoSQL Database cluster for increased efficiency, and to position records that share the shard key locally for easy reference and access. Records that share the shard key are stored in the same physical location and can be accessed atomically and efficiently.
Remove	Click this button to delete an existing column.
+ Add Primary Key Column	Click this button to add more columns while creating a composite (multi-column) primary key.
Column Name	Specify the column name.
Column Type	Select the data type for your column.
Default Value	(optional) Specify a default value for the column. **Note:** Default values can not be specified for binary and JSON data type columns.
Not Null	Select this option to specify that a column must always have a value.
Remove	Click this button to delete an existing column.
+ Add Column	Click this button to add more columns.
Unit	Select the unit (**Days** or **Hours**) to use for TTL value for the rows in the table.
Value	Specify expiration duration for the rows in the table. After the number of days or hours, the rows expire automatically, and are no longer available. The default value is zero, indicating no expiration time. **Note:** Updating Table Time to Live (TTL) does not change the TTL value of any existing data in the table. The new TTL value applies *only* to those rows that are added to the table after this value is modified and to the rows for which no overriding row-specific value has been supplied.

4. Click **Show DDL** to view the DDL statement formed based on the values entered in the fields in the **Simple DDL input** mode. This DDL statement gets executed when you click **Create**.
5. Click **Create**.

DROP TABLE

1. Right-click the target table.
2. Click **Drop Table**.
3. Click **Yes** to drop the table.

CREATE INDEX

- Locate the Table Explorer, and click the Refresh Schema to reload the schema.
- Right click on the table where index need to be created. Choose **Create Index**.
- Specify the name of the index and the columns to be part of the index.
- Click **Add Index**.

DROP INDEX

- Locate the Table Explorer, and click the Refresh Schema to reload the schema.
- Click on the table where the index needs to be removed. The list of indexes are displayed below the column names.
- Right click on the index to be dropped. Click **Drop Index**.
- A confirmation window appears, click **Ok** to confirm the drop action.

ADD COLUMN

- Locate the Table Explorer, and click the Refresh Schema to reload the schema.
- Right click on the table where column needs to be added. Click **Add columns**.
- Specify the name of the column and define the column with its properties - datatype, default value and whether it is nullable.
- Click **Add New Columns**.

DROP COLUMN

- Locate the Table Explorer, and click the Refresh Schema to reload the schema.
- Expand the table where column needs to be removed.
- Right click the column to be removed and choose **Drop Column**.
- A confirmation window appears, click **Ok** to confirm the drop action.

Perform DML operations using Visual Studio Code

You can add data, modify existing data and query data from tables usingVisual Studio Code plugin.

Insert Data

- Locate the Table Explorer, and click the Refresh Schema to reload the schema.

- Right click on the table where a row needs to be inserted. Choose **Insert Row**.
- In the Insert Row panel, enter the details for inserting a new row. You can INSERT a new ROW in two modes:
  - Simple Input : You can use this mode to insert the new row without writing a DML statement. Here a form based row fields entry is loaded, where you can enter the value of every field in the row.
  - Advanced JSON Input : You can use this mode to insert a new row into the table by supplying a JSON Object containing the column name and its corresponding value as key-value pairs.
- Click **Insert Row**.

Modify Data - UPDATE ROW/DELETE ROW:

- Locate the Table Explorer, and click the Refresh Schema to reload the schema.
- Click on the table where data needs to be modified.
- In the textbox on the right under SQL>, enter the SQL statement to fetch data from your table. Click **>** to run the query.
- To view individual cell data separately, click the table cell.
- To perform DML operations like Update and Delete Row, right-click on the particular row. Pick your option from the context-menu that appears.
  - Delete Row : A confirmation window appears, click **Ok** to delete the row.
  - Update Row : A separate HTML panel opens below the listed rows, containing the column names and its corresponding value in a form-based entry or provide the input as ON key-pair object. You can choose either of the two methods and supply new values.

> **Note:**
>
> In any row, PRIMARY KEY and GENERATED ALWAYS AS IDENTITY columns cannot be updated.

Executing SQL Queries for a Table

- Locate the Table Explorer, and click the Refresh Schema to reload the schema.
- Right click on the table and choose **Browse Table**.
- In the textbox on the right under SQL>, enter the SELECT statement to fetch data from your table.
- Click **>** to run the query. The corresponding data is retrieved from the table.
- Right click on any row and click Download row in to JSON file. The single row gets downloaded into a JSON file.
- Click **Download Query Result** to save the complete result of the SELECT statement as a JSON file.
- Click **Fetch All Records** to retrieve all data from the table.

Removing a Connection

Oracle NoSQL Database Connector provides two methods to remove a connection from Visual Studio (VS) Code.

You can:

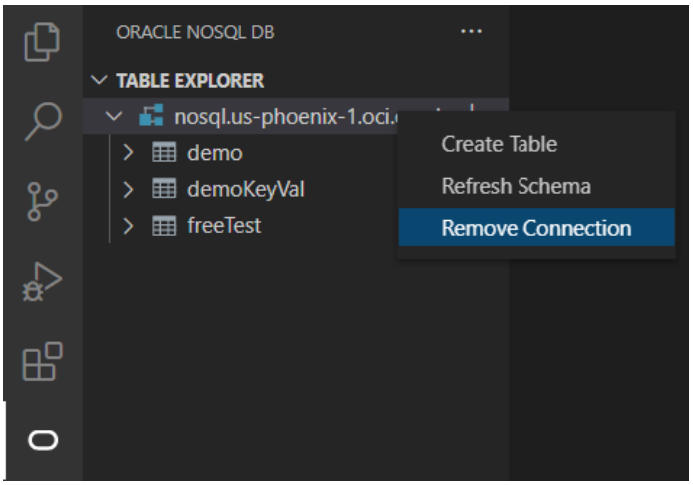
- Remove a connection with the Command Palette, or
- Remove a connection from the Oracle NoSQL DB view in the Activity Bar.

> **Note:**
>
> Removing a connection from Visual Studio Code deletes the persisted connection details from the current workspace.

- Remove Connection from Oracle NoSQL DB View
- Remove Connection with Command Palette

Remove Connection from Oracle NoSQL DB View

1. Expand the **TABLE EXPLORER** pane in the left navigation if it's collapsed.
2. Right-click the connection you want to remove, then click **Remove Connection**.

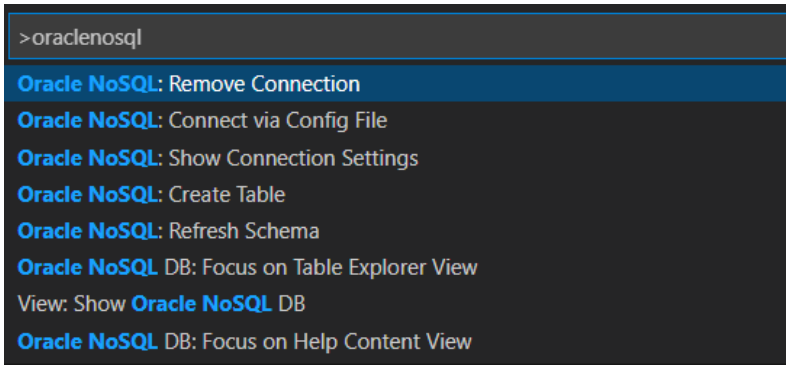


Remove Connection with Command Palette

1. Open the **Command Palette** by pressing:
   - (Windows and Linux) Control + Shift + P
   - (macOS) Command + Shift + P
2. From the Command Palette, select **OracleNoSQL: Remove Connection**.

Tip:

Enter `oraclenosql` in the Command Palette to display all of the Oracle NoSQL DB commands you can use.



6
Monitor

The articles in this section explain management and monitoring aspects of the Oracle NoSQL Database.

Monitoring the Store

You can obtain Information about the performance and availability of your store from both server side and client side perspectives:

- Your Oracle NoSQL Database applications can obtain performance statistics using the `oracle.kv.KVStore.getStats()` class. This provides a client side view of the complete round trip performance for Oracle NoSQL Database operations.
- Oracle NoSQL Database automatically captures Replication Node performance statistics into a log file that you can into into spreadsheet software for analysis. The store tracks, logs, and writes statistics at a user specified interval to a CSV file. The file is `je.stat.csv`, located in the Environment directory. Logging occurs per-Environment when the Environment is opened in read/write mode.

  Configuration parameters control the size and number of rotating log files to use (similar to java logging, see java.util.logging.FileHandler). For a rotating set of files, as each file reaches a given size limit, it is closed, rotated out, and a new file is opened. Successively older files are named with an incrementing numeric suffix to the file name. The name format is `je.stat[`*`version`*`].csv`.
- The Oracle NoSQL Database administrative service collects and aggregates status information, alerts, and performance statistics components that the store generates. This provides a detailed view of the behavior and performance of the Oracle NoSQL Database server.
- Each Oracle NoSQL Database Storage Node maintains detailed logs of trace information from the services that the node supports. The administrative service presents an aggregated, store-wide view of these component logs. Logs are available on each Storage Node if the administrative service is not available, or if it is more convenient to examine individual Storage Node logs. Additionally, you can compress these log files to store more logging output in the same disk space.
- Oracle NoSQL Database supports the optional Java Management Extensions (JMX) agents for monitoring. The JMX interfaces allow you to poll the Storage Nodes for information about the storage node and any replication nodes that it hosts. For more information on JMX monitoring, see Standardized Monitoring Interfaces . For information on using JMX securely, see Guidelines for using JMX securely in the *Security Guide*.

You can monitor the status of the store by verifying it from within the CLI. See Verifying the Store. You can also use the CLI to examine events.

Events

Events are special messages that inform you of the state of your system. As events are generated, they are routed through the monitoring system so that you can see them. There are four types of events that the store reports:

1. State Change events are issued when a service starts up or shuts down.
2. Performance events report statistics about the performance of various services.
3. Log events are records produced by the various system components to provide trace information about debugging. These records are produced by the standard `java.util.logging` package.
4. Plan Change events record the progress of plans as they execute, are interrupted, fail or are canceled.

> **Note:**
>
> - Some events are considered critical. These events are recorded in the administration service's database, and can be retrieved and viewed using the CLI.
> - You can compress the log event records that are produced by the standard `java.util.logging` package. For more information, see Log File Compression

You cannot view Plan Change events directly through Oracle NoSQL Database's administrative interfaces. However, State Change events, Performance events, and Log events are recorded using the EventRecorder facility, which is internal to the Admin. Only events considered *critical* are recorded, and the criteria for being designated as such varies with the event type. These are the events considered critical:

- All state changes.
- Log events classified as `SEVERE`.
- Any performance events reported as below a certain threshold.

You can view all of these critical events using the Admin CLI `show events` and `show event` commands.

Use the CLI `show events` command with no arguments to see all of the unexpired events in the database. Use the `-from` and `-to` arguments to limit the range of events that display. Use the `-type` or `-id` arguments to filter events by type or id, respectively.

For example, this is part of the output from a `show events` command:

```
kv-> show events
idarpdfbS STAT 2015-08-13 22:18:39.287 UTC sn1 RUNNING sev1
idarpeg0S STAT 2015-08-13 22:18:40.608 UTC sn2 RUNNING sev1
idarphmuS STAT 2015-08-13 22:18:44.742 UTC rg1-rn1 RUNNING sev1
idarpjLLS STAT 2015-08-13 22:18:47.289 UTC rg1-rn2 RUNNING sev1
```

```
idartfcuS STAT 2015-08-13 22:21:48.414 UTC rg1-rn2 UNREACHABLE sev2
                                                  (reported by admin1)
```

This result shows four service state change events (`sev1`) and one log event (`UNREACHABLE`), classified as `sev2`. Tags at the beginning of each line are individual event record identifiers. To see detailed information for a particular event, use the `show event` command, which takes an event record identifier, such as `idartfcuS` as its argument:

```
kv-> show event -id idartfcuS
idartfcuS STAT 2015-08-13 22:21:48.414 UTC rg1-rn2 UNREACHABLE sev2
  (reported by admin1)
```

Using this method of event identifiers, you can see a complete stack trace.

Events are removed from the system if the total number of events is greater than a set maximum number, or if the Event is older than a set period. The default maximum number of events is 10,000, while the default time period is 30 days.

Both `Sev1` and `Sev2` flags are associated with specific service state change events. `Sev1` flags report the current state. `Sev2` flags report errors during attempted state changes, as follows:

Sev1 Flags	Sev2 Flags
STARTING	ERROR_RESTARTING
WAITING_FOR_DEPLOY	ERROR_NO_RESTART
RUNNING	UNREACHABLE
STOPPING	
STOPPED	

Log File Compression

You can compress the log files by setting the `serviceLogFileCompression` Storage Node parameter. This helps to store significantly more logging output in the same amount of disk space. As a result, the logs can be retained for a longer period, rendering their availability for debugging purposes. The log files are compressed with the standard `gzip` algorithm. By default, the compression of log files stores approximately five times more data as compared with uncompressed files.

When you enable the log file compression, restart the associated Storage Node Agent for the new setting to take effect on the new files and compress the existing files. The size of log files may temporarily exceed the defined limits in certain cases. The rotated copies of the store-combined debug logs, statistics files, and performance files are compressed. Additionally, the service debug log files across all Storage Nodes, Replication Nodes, Admins, and Arbiters are compressed.

The compressed rotated log files are renamed with a `.gz` suffix and the file name is modified to use the creation time instead of the standard log rotation number.

For example:

Precompressed log file	Post compressed log file
rg1-rn1_1.log	rg1-rn1_20220720-201902.log.gz

Here, the standard rotated log file `rg1-rn1_1.log` is renamed to `rg1-rn1_20220720-201902.log.gz` after compression. The date part of the log file implies that the file was created on `2022-07-20 20:19:02 UTC`.

If multiple rotated log files are created in the same second, a unique suffix is added to the date portion of the file name.

For example: `rg1-rn1_20220720-201902-1.log.gz`.

> **Note:**
>
> - To avoid the usage of extra disk space in the standard log directory, you can use the `gzcat` command to view the contents without uncompressing the zipped files. Use the `zgrep` command to search the compressed log files. You can also uncompress the files into another directory. Manually uncompressed files are not deleted automatically.
> - The Bootstrap debug log files, GC log files, JE debug log files, and JE statistics files are not compressed.

Software Monitoring

Being a distributed system, the Oracle NoSQL Database is composed of several software components and each expose unique metrics that can be monitored, interpreted, and utilized to understand the general health, performance, and operational capability of the Oracle NoSQL Database cluster.

This section focuses on best practices for monitoring the Oracle NoSQL software components. While there are several software dependencies for the Oracle NoSQL Database itself (for example, Java virtual machine, operating system, NTP), this section focuses solely on the NoSQL components.

There are four basic mechanisms for monitoring the health of the Oracle NoSQL Database:

- System Log File Monitoring – Oracle NoSQL Database uses the java.util.logging package to write all trace, information, and error messages to the log files for each component of the store. These files can be parsed using the typical log file probing mechanism supported by the leading system management solutions.
- Application Monitoring – A good proxy for the “health” of the Oracle NoSQL Database rests with application level metrics. Metrics like average and 90th percentile response times, average and 90th percentile throughput, as well average number of timeout exceptions encountered from NoSQL API calls are all potential indicators that something may be wrong with a component in the NoSQL cluster. In fact, sampling these metrics and looking for deviations from mean values can be the best way to know that something may be wrong with your environment.
- Oracle Enterprise Manager (OEM) – The integration of Oracle NoSQL Database with OEM primarily takes the form of an EM plug-in. The plug-in allows monitoring store components, their availability, performance metrics, and operational parameters through Enterprise Manager. For more information on OEM, see About IntelliJ Plugin.

The following sections discuss details of each of these monitoring techniques (except OEM) and illustrate how each of them can be utilized to detect failures in Oracle NoSQL Database components.

System Log File Monitoring

The Oracle NoSQL Database is composed of the following components, and each component produces log files that can be monitored:

- Replication Nodes (RN) – Service read and write requests from API calls. Replication Nodes for a particular shard are laid out on different Storage Nodes (physical servers) by the topology manager, so the log files for the nodes in each shard are spread across multiple machines.
- Storage Node Agents (SNA) – Manage the Replication Nodes that are running on each Storage Node (SN). The Storage Node Agent maintains its own log regarding the state of each replication node it is managing. You can think of the Storage Node Agent log as a high level log of the Replication Node activity on a particular Storage Node.
- Administration (Admin) Nodes – Administrative Nodes handle the execution of commands from the administrative command line interface. Long running plans are also staged from the administrative nodes. Administrative Nodes also maintain a consolidated log of all the other logs in the Oracle NoSQL cluster.

All of the above mentioned log files can be found in the following directory structure `KVROOT/kvstore/log` on the machine where the component is running. It is possible to compress these log files to accommodate more files in the given disk space. For more details, see Log File Compression.

The following steps can be used to find the machines that are running the components of the cluster:

1. `java -Xmx64m -Xms64m -jar kvstore.jar ping -host <any machine in the cluster> -port <the port number used to initialize the KVStore>`
2. Each Storage Node (snXX) is listed in the output of the ping command, along with a list of Replication nodes (rgXX-rnXX) running on the host listed in the ping output. XX denotes the unique number assigned to that component by NoSQL Database. For Replication Nodes, rg denotes the shard number and stands for replication group, while rn denotes the Replication Node number within that shard.
3. Administration (Admin) Nodes – Identifying the nodes in the cluster that are running administrative services is a bit more challenging. To identify these nodes, a script would run ps axww on every host in the cluster and grep for kvstore.jar and -class Admin.

The Oracle NoSQL Database maintains a single consolidated log of every node in the cluster, and this can be found on any of the nodes running an administrative service. While this is a convenient and easy single place to monitor for errors, it is not 100% guaranteed. The single consolidated view is aggregated by getting log messages over the network, and transient network failures, packet loss, and high network utilization can cause this consolidated log to either be out of date, or have missing entries. Therefore, we recommend monitoring each host in the cluster as well as monitoring each type of log file on each host in the cluster.

Generally speaking, any log message with a level of SEVERE should be considered a potentially critical event and worthy of generating a systems management notification. The sections in the later part of this document illustrate how to correlate specific SEVERE exceptions with hardware component failure.

Java Management Extensions (JMX) Monitoring

Oracle NoSQL Database is also monitored through JMX based system management tools. For more information on JMX, see Standardized Monitoring Interfaces .

Monitoring for Storage Nodes (SN)

A Storage Node is a physical (or virtual) machine with its own local storage, which houses the Replication Node. For more information, see Architecture in the *Concepts Guide*.

See the following sections:

- Metrics for Storage Nodes
- Java Management Extensions (JMX) Notifications

Metrics for Storage Nodes

- snServiceStatus – The current status of the Storage Node Agent running on the host. The Storage Node Agent manages all the Replication Nodes running on the Storage Node (host). The textual representation along with the enumeration ID are shown below:
  - starting (1) – The Storage Node Agent is booting up.
  - waitingForDeploy (2) – The Storage Node Agent is waiting for the initial deploy-SN command to be run.
  - running(3) – The Storage Node Agent is running.
  - stopping(4) – The Storage Node Agent is in the process of shutting down. It may be in the process of shutting down Replication Nodes that it manages.
  - stopped(5) – An intentional clean shutdown.
  - errorRestarting(6) – Although this state exists in the category, it is typically never seen for storage node agents.
  - errorNoRestart(7) – Although this state exists in the category, it is typically never seen for storage node agents.
  - unreachable(8) – The Storage Node Agent is unreachable by the admin service.

    > **Note:**
    >
    > If a Storage Node is UNREACHABLE, or a Replication Node is having problems and its Storage Node is UNREACHABLE, first check the network connectivity between the Admin and the Storage Node. If the managing Storage Node Agent is reachable, but the managed Replication Node is not, the problem most likely lies somewhere other than the network.

  - expectedRestarting(9) – This state is rare for Storage Node Agents.
- snHostName – The name of the host where the Storage Node agent has been deployed.

- snRegistryPort – The TCP/IP port on which Oracle NoSQL Database should be contacted.
- snHAHostName – If the HA host name has been configured through the boot parameters then this is returned, otherwise the name of the host running the Storage Node agent is returned. This value represents the network interface name that the replication subsystem uses for internode communication. The HA host name is specified using the `-hahost` flag to the `makebootconfig` command, and it corresponds to the haHostname Storage Node parameter, in the Setting Store Parameters. If users encounter a problem indicating that the HA host name has been specified incorrectly, first check that they have used the correct value in the call to the `makebootconfig` command. The user can change the value later by modifying the `haHostname` parameter. For more information, see makebootconfig.
- snHaPortRange – The range of ports that replication nodes use to communicate among themselves.
- snStoreName – The name of the KVStore that this storage node agent is servicing.
- snRootDirPath – The fully qualified path to the root of the directory structure where the Oracle NoSQL Database installation files exist.
- snLogFileCount – A logging config parameter that represents the maximum number of log files that are retained by the Storage Node Agent.
- snLogFileLimit – A logging config parameter that represents the maximum size of a single log file in bytes.
- snCapacity – The current capacity of the Storage Node. This parameter essentially describes the number of persistent storage devices on the Storage Node and is typically set at store initialization time, but can be modified by the administrator if the hardware configuration is changed after the store is initialized.
- snMountPoints – A list of one or more fully qualified paths to the data files that reside on this storage node.
- snMemory – The current memory configuration for this Storage Node in megabytes. This parameter is typically set at store initialization time, but can be modified by the administrator if the hardware configuration is changed after the store is initialized.
- snCPUs – The current number of CPUs configured for this Storage Node. This parameter is typically set at store initialization time. The administrator can modify the number if the hardware configuration changes after the store is initialized.
- snCollectorInterval – The interval that all nodes are using for aggregate statistics.

Java Management Extensions (JMX) Notifications

Mbean Object Name: Oracle NoSQL Database:type=StorageNode

- New operation performance metrics are available as follows:
  - `Type: oracle.kv.repnode.opmetric`
  - `User Data`: Contains a full listing of performance metrics for a given RN. The statistics are a string in JSON form, and are obtained via Notification.getUserData().

    These metrics contain statistics of each type of API operation. And each operation statistics is calculated by interval and cumulative statistics. Interval statistics cover a

single measurement period, cumulative statistics cover the duration of this repNode's uptime. Statistics follows the following naming convention:

```
[Operation]_[Interval|Cumulative]_[Metric]
```

[Operation] has following user operations: Gets, Puts, PutIfAbsent, PutIfPresent, PutIfVersion, Deletes, DeleteIfVersion, MultiGets, MultiGetKeys, MultiGetIterator, MultiGetKeysIterator, StoreIterator, StoreKeysIterator, MultiDeletes, Executes, IndexIterator, IndexKeysIterator, QuerySinglePartition, QueryMultiPartition, QueryMultiShard, BulkPut, BulkGet, BulkGetKeys, BulkGetTable, BulkGetTableKeys

`AllSingleKeyOperations` are Gets, Puts, PutIfAbsent, PutIfPresent, PutIfVersion, Deletes, DeleteIfVersion

`AllMultiKeyOperations` are MultiGetKeys, MultiGetIterator, MultiGetKeysIterator, StoreIterator, StoreKeysIterator, MultiDeletes, Executes, IndexIterator, IndexKeysIterator, QuerySinglePartition, QueryMultiPartition, QueryMultiShard, BulkPut, BulkGet, BulkGetKeys, BulkGetTable, BulkGetTableKeys

Read operations are Gets, MultiGets, MultiGetKeys, MultiGetIterator, MultiGetKeysIterator, StoreIterator, StoreKeysIterator, IndexIterator, IndexKeysIterator, QuerySinglePartition, QueryMultiPartition, QueryMultiShard, BulkGet, BulkGetKeys, BulkGetTable, BulkGetTableKeys

Write operation are Puts, PutIfAbsent, PutIfPresent, PutIfVersion, Deletes, DeleteIfVersion, MultiDeletes, Executes, BulkPut

[Metric] has the following types:

- `TotalReq`: The total number of operation requests.
- `TotalOps`: The total number of records returned or processed. Single operation requests only apply to one record, but the multi, iterate, query, bulk or execute operation requests will work on multiple records.
- `PerSec`: Operation throughput per second, that is [TotalOps] / [Interval]
- `Min`: minimum latency
- `Max`: maximum latency
- `Avg`: average latency
- `95th`: The maximum value within the bottom 95% of latency values.
- `99th`: The maximum value within the bottom 99% of latency values.

The average latency tells users how long to expect calls to take when considering a large number of calls. The 95th and 99th percentile latency numbers provide information about how much call times vary in cases where calls took longer than the average amount of time to complete. 95% of calls completed within the time specified by the 95th percentile number; 5% of calls took at least that long to complete. 99% of calls completed within the time specified by the 99th percentile number; 1% of calls took at least that long to complete.

For example, consider the following latency values:

- Avg: 1 ms
- 95th: 3 ms
- 99th: 10 ms

If these were the measurements for 1000 calls to the store, then the average means that, overall, the 1000 calls took a total of 1000 ms (1000 x 1 ms), with a mix of call times, some less than 1 ms and some greater. The 95% and 99% values give some sense of how the call times varied over the set of calls. A 95% value of 3 ms means that, out of 1000 calls, 950 (95% of 1000) took less than 3 ms, and 50 (5% of 1000) took 3 ms or longer. A 99% value of 10 ms means that, out of 1000 calls, 990 (99% of 1000) took less than 10 ms, and 10 (1% of 1000) took 10 ms or longer.

The `MultiGets_Interval_TotalOps` stats tells how many records were read through MultiGets operations in the last interval. `MultiGets_Cumulative_TotalOps` stats tells how many records were read through MultiGets operations in the whole Replication Node lifetime.

A sample operation performance metrics:


```
{
  "resource": "rg1-rn1",
  "shard": "rg1",
  "reportTime": 1481031260001,
  "AllSingleKeyOperations_Interval_TotalOps": 154571,
  "AllSingleKeyOperations_Interval_TotalReq": 154571,
  "AllSingleKeyOperations_Interval_PerSec": 7728,
  "AllSingleKeyOperations_Interval_Min": 0,
  "AllSingleKeyOperations_Interval_Max": 72,
  "AllSingleKeyOperations_Interval_Avg": 0.09015835076570511,
  "AllSingleKeyOperations_Interval_95th": 0,
  "AllSingleKeyOperations_Interval_99th": 0,
  "AllSingleKeyOperations_Cumulative_TotalOps": 27916089,
  "AllSingleKeyOperations_Cumulative_TotalReq": 27916089,
  "AllSingleKeyOperations_Cumulative_PerSec": 854,
  "AllSingleKeyOperations_Cumulative_Min": 0,
  "AllSingleKeyOperations_Cumulative_Max": 5124,
  "AllSingleKeyOperations_Cumulative_Avg": 0.1090782955288887,
  "AllSingleKeyOperations_Cumulative_95th": 0,
  "AllSingleKeyOperations_Cumulative_99th": 0,
  "AllMultiKeyOperations_Interval_TotalOps": 6002,
  "AllMultiKeyOperations_Interval_TotalReq": 6002,
  "AllMultiKeyOperations_Interval_PerSec": 300,
  "AllMultiKeyOperations_Interval_Min": 0,
  "AllMultiKeyOperations_Interval_Max": 29,
  "AllMultiKeyOperations_Interval_Avg": 0.14758114516735077,
  "AllMultiKeyOperations_Interval_95th": 0,
  "AllMultiKeyOperations_Interval_99th": 1,
  "AllMultiKeyOperations_Cumulative_TotalOps": 1105133,
  "AllMultiKeyOperations_Cumulative_TotalReq": 1105133,
  "AllMultiKeyOperations_Cumulative_PerSec": 33,
  "AllMultiKeyOperations_Cumulative_Min": 0,
  "AllMultiKeyOperations_Cumulative_Max": 956,
  "AllMultiKeyOperations_Cumulative_Avg": 0.16301529109477997,
  "AllMultiKeyOperations_Cumulative_95th": 0,
  "AllMultiKeyOperations_Cumulative_99th": 1,
  "Gets_Interval_TotalOps": 154571,
  "Gets_Interval_TotalReq": 154571,
  "Gets_Interval_PerSec": 7728,
  "Gets_Interval_Min": 0,
  "Gets_Interval_Max": 72,
```

```
"Gets_Interval_Avg": 0.08909573405981064,
"Gets_Interval_95th": 0,
"Gets_Interval_99th": 0,
"Gets_Cumulative_TotalOps": 27916089,
"Gets_Cumulative_TotalReq": 27916089,
"Gets_Cumulative_PerSec": 854,
"Gets_Cumulative_Min": 0,
"Gets_Cumulative_Max": 5124,
"Gets_Cumulative_Avg": 0.10803056508302689,
"Gets_Cumulative_95th": 0,
"Gets_Cumulative_99th": 0,
"Puts_Interval_TotalOps": 0,
"Puts_Interval_TotalReq": 0,
"Puts_Interval_PerSec": 0,
"Puts_Interval_Min": 0,
"Puts_Interval_Max": 0,
"Puts_Interval_Avg": 0,
"Puts_Interval_95th": 0,
"Puts_Interval_99th": 0,
"PutIfAbsent_Interval_TotalOps": 0,
"PutIfAbsent_Interval_TotalReq": 0,
"PutIfAbsent_Interval_PerSec": 0,
"PutIfAbsent_Interval_Min": 0,
"PutIfAbsent_Interval_Max": 0,
"PutIfAbsent_Interval_Avg": 0,
"PutIfAbsent_Interval_95th": 0,
"PutIfAbsent_Interval_99th": 0,
"PutIfPresent_Interval_TotalOps": 0,
"PutIfPresent_Interval_TotalReq": 0,
"PutIfPresent_Interval_PerSec": 0,
"PutIfPresent_Interval_Min": 0,
"PutIfPresent_Interval_Max": 0,
"PutIfPresent_Interval_Avg": 0,
"PutIfPresent_Interval_95th": 0,
"PutIfPresent_Interval_99th": 0,
"PutIfVersion_Interval_TotalOps": 0,
"PutIfVersion_Interval_TotalReq": 0,
"PutIfVersion_Interval_PerSec": 0,
"PutIfVersion_Interval_Min": 0,
"PutIfVersion_Interval_Max": 0,
"PutIfVersion_Interval_Avg": 0,
"PutIfVersion_Interval_95th": 0,
"PutIfVersion_Interval_99th": 0,
"Puts_Cumulative_TotalOps": 0,
"Puts_Cumulative_TotalReq": 0,
"Puts_Cumulative_PerSec": 0,
"Puts_Cumulative_Min": 0,
"Puts_Cumulative_Max": 0,
"Puts_Cumulative_Avg": 0,
"Puts_Cumulative_95th": 0,
"Puts_Cumulative_99th": 0,
"PutIfAbsent_Cumulative_TotalOps": 0,
"PutIfAbsent_Cumulative_TotalReq": 0,
"PutIfAbsent_Cumulative_PerSec": 0,
"PutIfAbsent_Cumulative_Min": 0,
```

```
"PutIfAbsent_Cumulative_Max": 0,
"PutIfAbsent_Cumulative_Avg": 0,
"PutIfAbsent_Cumulative_95th": 0,
"PutIfAbsent_Cumulative_99th": 0,
"PutIfPresent_Cumulative_TotalOps": 0,
"PutIfPresent_Cumulative_TotalReq": 0,
"PutIfPresent_Cumulative_PerSec": 0,
"PutIfPresent_Cumulative_Min": 0,
"PutIfPresent_Cumulative_Max": 0,
"PutIfPresent_Cumulative_Avg": 0,
"PutIfPresent_Cumulative_95th": 0,
"PutIfPresent_Cumulative_99th": 0,
"PutIfVersion_Cumulative_TotalOps": 0,
"PutIfVersion_Cumulative_TotalReq": 0,
"PutIfVersion_Cumulative_PerSec": 0,
"PutIfVersion_Cumulative_Min": 0,
"PutIfVersion_Cumulative_Max": 0,
"PutIfVersion_Cumulative_Avg": 0,
"PutIfVersion_Cumulative_95th": 0,
"PutIfVersion_Cumulative_99th": 0,
"Deletes_Interval_TotalOps": 0,
"Deletes_Interval_TotalReq": 0,
"Deletes_Interval_PerSec": 0,
"Deletes_Interval_Min": 0,
"Deletes_Interval_Max": 0,
"Deletes_Interval_Avg": 0,
"Deletes_Interval_95th": 0,
"Deletes_Interval_99th": 0,
"DeleteIfVersion_Interval_TotalOps": 0,
"DeleteIfVersion_Interval_TotalReq": 0,
"DeleteIfVersion_Interval_PerSec": 0,
"DeleteIfVersion_Interval_Min": 0,
"DeleteIfVersion_Interval_Max": 0,
"DeleteIfVersion_Interval_Avg": 0,
"DeleteIfVersion_Interval_95th": 0,
"DeleteIfVersion_Interval_99th": 0,
"Deletes_Cumulative_TotalOps": 0,
"Deletes_Cumulative_TotalReq": 0,
"Deletes_Cumulative_PerSec": 0,
"Deletes_Cumulative_Min": 0,
"Deletes_Cumulative_Max": 0,
"Deletes_Cumulative_Avg": 0,
"Deletes_Cumulative_95th": 0,
"Deletes_Cumulative_99th": 0,
"DeleteIfVersion_Cumulative_TotalOps": 0,
"DeleteIfVersion_Cumulative_TotalReq": 0,
"DeleteIfVersion_Cumulative_PerSec": 0,
"DeleteIfVersion_Cumulative_Min": 0,
"DeleteIfVersion_Cumulative_Max": 0,
"DeleteIfVersion_Cumulative_Avg": 0,
"DeleteIfVersion_Cumulative_95th": 0,
"DeleteIfVersion_Cumulative_99th": 0,
"MultiGets_Interval_TotalOps": 0,
"MultiGets_Interval_TotalReq": 0,
"MultiGets_Interval_PerSec": 0,
```

```
"MultiGets_Interval_Min": 0,
"MultiGets_Interval_Max": 0,
"MultiGets_Interval_Avg": 0,
"MultiGets_Interval_95th": 0,
"MultiGets_Interval_99th": 0,
"MultiGetKeys_Interval_TotalOps": 0,
"MultiGetKeys_Interval_TotalReq": 0,
"MultiGetKeys_Interval_PerSec": 0,
"MultiGetKeys_Interval_Min": 0,
"MultiGetKeys_Interval_Max": 0,
"MultiGetKeys_Interval_Avg": 0,
"MultiGetKeys_Interval_95th": 0,
"MultiGetKeys_Interval_99th": 0,
"MultiGetIterator_Interval_TotalOps": 0,
"MultiGetIterator_Interval_TotalReq": 0,
"MultiGetIterator_Interval_PerSec": 0,
"MultiGetIterator_Interval_Min": 0,
"MultiGetIterator_Interval_Max": 0,
"MultiGetIterator_Interval_Avg": 0,
"MultiGetIterator_Interval_95th": 0,
"MultiGetIterator_Interval_99th": 0,
"MultiGetKeysIterator_Interval_TotalOps": 0,
"MultiGetKeysIterator_Interval_TotalReq": 0,
"MultiGetKeysIterator_Interval_PerSec": 0,
"MultiGetKeysIterator_Interval_Min": 0,
"MultiGetKeysIterator_Interval_Max": 0,
"MultiGetKeysIterator_Interval_Avg": 0,
"MultiGetKeysIterator_Interval_95th": 0,
"MultiGetKeysIterator_Interval_99th": 0,
"MultiGets_Cumulative_TotalOps": 0,
"MultiGets_Cumulative_TotalReq": 0,
"MultiGets_Cumulative_PerSec": 0,
"MultiGets_Cumulative_Min": 0,
"MultiGets_Cumulative_Max": 0,
"MultiGets_Cumulative_Avg": 0,
"MultiGets_Cumulative_95th": 0,
"MultiGets_Cumulative_99th": 0,
"MultiGetKeys_Cumulative_TotalOps": 0,
"MultiGetKeys_Cumulative_TotalReq": 0,
"MultiGetKeys_Cumulative_PerSec": 0,
"MultiGetKeys_Cumulative_Min": 0,
"MultiGetKeys_Cumulative_Max": 0,
"MultiGetKeys_Cumulative_Avg": 0,
"MultiGetKeys_Cumulative_95th": 0,
"MultiGetKeys_Cumulative_99th": 0,
"MultiGetIterator_Cumulative_TotalOps": 0,
"MultiGetIterator_Cumulative_TotalReq": 0,
"MultiGetIterator_Cumulative_PerSec": 0,
"MultiGetIterator_Cumulative_Min": 0,
"MultiGetIterator_Cumulative_Max": 0,
"MultiGetIterator_Cumulative_Avg": 0,
"MultiGetIterator_Cumulative_95th": 0,
"MultiGetIterator_Cumulative_99th": 0,
"MultiGetKeysIterator_Cumulative_TotalOps": 0,
"MultiGetKeysIterator_Cumulative_TotalReq": 0,
```

```
"MultiGetKeysIterator_Cumulative_PerSec": 0,
"MultiGetKeysIterator_Cumulative_Min": 0,
"MultiGetKeysIterator_Cumulative_Max": 0,
"MultiGetKeysIterator_Cumulative_Avg": 0,
"MultiGetKeysIterator_Cumulative_95th": 0,
"MultiGetKeysIterator_Cumulative_99th": 0,
"StoreIterator_Interval_TotalOps": 0,
"StoreIterator_Interval_TotalReq": 0,
"StoreIterator_Interval_PerSec": 0,
"StoreIterator_Interval_Min": 0,
"StoreIterator_Interval_Max": 0,
"StoreIterator_Interval_Avg": 0,
"StoreIterator_Interval_95th": 0,
"StoreIterator_Interval_99th": 0,
"StoreKeysIterator_Interval_TotalOps": 0,
"StoreKeysIterator_Interval_TotalReq": 0,
"StoreKeysIterator_Interval_PerSec": 0,
"StoreKeysIterator_Interval_Min": 0,
"StoreKeysIterator_Interval_Max": 0,
"StoreKeysIterator_Interval_Avg": 0,
"StoreKeysIterator_Interval_95th": 0,
"StoreKeysIterator_Interval_99th": 0,
"StoreIterator_Cumulative_TotalOps": 0,
"StoreIterator_Cumulative_TotalReq": 0,
"StoreIterator_Cumulative_PerSec": 0,
"StoreIterator_Cumulative_Min": 0,
"StoreIterator_Cumulative_Max": 0,
"StoreIterator_Cumulative_Avg": 0,
"StoreIterator_Cumulative_95th": 0,
"StoreIterator_Cumulative_99th": 0,
"StoreKeysIterator_Cumulative_TotalOps": 0,
"StoreKeysIterator_Cumulative_TotalReq": 0,
"StoreKeysIterator_Cumulative_PerSec": 0,
"StoreKeysIterator_Cumulative_Min": 0,
"StoreKeysIterator_Cumulative_Max": 0,
"StoreKeysIterator_Cumulative_Avg": 0,
"StoreKeysIterator_Cumulative_95th": 0,
"StoreKeysIterator_Cumulative_99th": 0,
"MultiDeletes_Interval_TotalOps": 0,
"MultiDeletes_Interval_TotalReq": 0,
"MultiDeletes_Interval_PerSec": 0,
"MultiDeletes_Interval_Min": 0,
"MultiDeletes_Interval_Max": 0,
"MultiDeletes_Interval_Avg": 0,
"MultiDeletes_Interval_95th": 0,
"MultiDeletes_Interval_99th": 0,
"MultiDeletes_Cumulative_TotalOps": 0,
"MultiDeletes_Cumulative_TotalReq": 0,
"MultiDeletes_Cumulative_PerSec": 0,
"MultiDeletes_Cumulative_Min": 0,
"MultiDeletes_Cumulative_Max": 0,
"MultiDeletes_Cumulative_Avg": 0,
"MultiDeletes_Cumulative_95th": 0,
"MultiDeletes_Cumulative_99th": 0,
"Executes_Interval_TotalOps": 0,
```

```
"Executes_Interval_TotalReq": 0,
"Executes_Interval_PerSec": 0,
"Executes_Interval_Min": 0,
"Executes_Interval_Max": 0,
"Executes_Interval_Avg": 0,
"Executes_Interval_95th": 0,
"Executes_Interval_99th": 0,
"Executes_Cumulative_TotalOps": 0,
"Executes_Cumulative_TotalReq": 0,
"Executes_Cumulative_PerSec": 0,
"Executes_Cumulative_Min": 0,
"Executes_Cumulative_Max": 0,
"Executes_Cumulative_Avg": 0,
"Executes_Cumulative_95th": 0,
"Executes_Cumulative_99th": 0,
"NOPs_Interval_TotalOps": 0,
"NOPs_Interval_TotalReq": 0,
"NOPs_Interval_PerSec": 0,
"NOPs_Interval_Min": 0,
"NOPs_Interval_Max": 0,
"NOPs_Interval_Avg": 0,
"NOPs_Interval_95th": 0,
"NOPs_Interval_99th": 0,
"NOPs_Cumulative_TotalOps": 0,
"NOPs_Cumulative_TotalReq": 0,
"NOPs_Cumulative_PerSec": 0,
"NOPs_Cumulative_Min": 0,
"NOPs_Cumulative_Max": 0,
"NOPs_Cumulative_Avg": 0,
"NOPs_Cumulative_95th": 0,
"NOPs_Cumulative_99th": 0,
"IndexIterator_Interval_TotalOps": 6002,
"IndexIterator_Interval_TotalReq": 6002,
"IndexIterator_Interval_PerSec": 300,
"IndexIterator_Interval_Min": 0,
"IndexIterator_Interval_Max": 29,
"IndexIterator_Interval_Avg": 0.14662425220012665,
"IndexIterator_Interval_95th": 0,
"IndexIterator_Interval_99th": 1,
"IndexKeysIterator_Interval_TotalOps": 0,
"IndexKeysIterator_Interval_TotalReq": 0,
"IndexKeysIterator_Interval_PerSec": 0,
"IndexKeysIterator_Interval_Min": 0,
"IndexKeysIterator_Interval_Max": 0,
"IndexKeysIterator_Interval_Avg": 0,
"IndexKeysIterator_Interval_95th": 0,
"IndexKeysIterator_Interval_99th": 0,
"IndexIterator_Cumulative_TotalOps": 1105133,
"IndexIterator_Cumulative_TotalReq": 1105133,
"IndexIterator_Cumulative_PerSec": 33,
"IndexIterator_Cumulative_Min": 0,
"IndexIterator_Cumulative_Max": 956,
"IndexIterator_Cumulative_Avg": 0.1620502769947052,
"IndexIterator_Cumulative_95th": 0,
"IndexIterator_Cumulative_99th": 1,
```

```
"IndexKeysIterator_Cumulative_TotalOps": 0,
"IndexKeysIterator_Cumulative_TotalReq": 0,
"IndexKeysIterator_Cumulative_PerSec": 0,
"IndexKeysIterator_Cumulative_Min": 0,
"IndexKeysIterator_Cumulative_Max": 0,
"IndexKeysIterator_Cumulative_Avg": 0,
"IndexKeysIterator_Cumulative_95th": 0,
"IndexKeysIterator_Cumulative_99th": 0,
"QuerySinglePartition_Interval_TotalOps": 0,
"QuerySinglePartition_Interval_TotalReq": 0,
"QuerySinglePartition_Interval_PerSec": 0,
"QuerySinglePartition_Interval_Min": 0,
"QuerySinglePartition_Interval_Max": 0,
"QuerySinglePartition_Interval_Avg": 0,
"QuerySinglePartition_Interval_95th": 0,
"QuerySinglePartition_Interval_99th": 0,
"QueryMultiPartition_Interval_TotalOps": 0,
"QueryMultiPartition_Interval_TotalReq": 0,
"QueryMultiPartition_Interval_PerSec": 0,
"QueryMultiPartition_Interval_Min": 0,
"QueryMultiPartition_Interval_Max": 0,
"QueryMultiPartition_Interval_Avg": 0,
"QueryMultiPartition_Interval_95th": 0,
"QueryMultiPartition_Interval_99th": 0,
"QueryMultiShard_Interval_TotalOps": 0,
"QueryMultiShard_Interval_TotalReq": 0,
"QueryMultiShard_Interval_PerSec": 0,
"QueryMultiShard_Interval_Min": 0,
"QueryMultiShard_Interval_Max": 0,
"QueryMultiShard_Interval_Avg": 0,
"QueryMultiShard_Interval_95th": 0,
"QueryMultiShard_Interval_99th": 0,
"QuerySinglePartition_Cumulative_TotalOps": 0,
"QuerySinglePartition_Cumulative_TotalReq": 0,
"QuerySinglePartition_Cumulative_PerSec": 0,
"QuerySinglePartition_Cumulative_Min": 0,
"QuerySinglePartition_Cumulative_Max": 0,
"QuerySinglePartition_Cumulative_Avg": 0,
"QuerySinglePartition_Cumulative_95th": 0,
"QuerySinglePartition_Cumulative_99th": 0,
"QueryMultiPartition_Cumulative_TotalOps": 0,
"QueryMultiPartition_Cumulative_TotalReq": 0,
"QueryMultiPartition_Cumulative_PerSec": 0,
"QueryMultiPartition_Cumulative_Min": 0,
"QueryMultiPartition_Cumulative_Max": 0,
"QueryMultiPartition_Cumulative_Avg": 0,
"QueryMultiPartition_Cumulative_95th": 0,
"QueryMultiPartition_Cumulative_99th": 0,
"QueryMultiShard_Cumulative_TotalOps": 0,
"QueryMultiShard_Cumulative_TotalReq": 0,
"QueryMultiShard_Cumulative_PerSec": 0,
"QueryMultiShard_Cumulative_Min": 0,
"QueryMultiShard_Cumulative_Max": 0,
"QueryMultiShard_Cumulative_Avg": 0,
"QueryMultiShard_Cumulative_95th": 0,
```

```
"QueryMultiShard_Cumulative_99th": 0,
"BulkPut_Interval_TotalOps": 0,
"BulkPut_Interval_TotalReq": 0,
"BulkPut_Interval_PerSec": 0,
"BulkPut_Interval_Min": 0,
"BulkPut_Interval_Max": 0,
"BulkPut_Interval_Avg": 0,
"BulkPut_Interval_95th": 0,
"BulkPut_Interval_99th": 0,
"BulkPut_Cumulative_TotalOps": 0,
"BulkPut_Cumulative_TotalReq": 0,
"BulkPut_Cumulative_PerSec": 0,
"BulkPut_Cumulative_Min": 0,
"BulkPut_Cumulative_Max": 0,
"BulkPut_Cumulative_Avg": 0,
"BulkPut_Cumulative_95th": 0,
"BulkPut_Cumulative_99th": 0,
"BulkGet_Interval_TotalOps": 0,
"BulkGet_Interval_TotalReq": 0,
"BulkGet_Interval_PerSec": 0,
"BulkGet_Interval_Min": 0,
"BulkGet_Interval_Max": 0,
"BulkGet_Interval_Avg": 0,
"BulkGet_Interval_95th": 0,
"BulkGet_Interval_99th": 0,
"BulkGetKeys_Interval_TotalOps": 0,
"BulkGetKeys_Interval_TotalReq": 0,
"BulkGetKeys_Interval_PerSec": 0,
"BulkGetKeys_Interval_Min": 0,
"BulkGetKeys_Interval_Max": 0,
"BulkGetKeys_Interval_Avg": 0,
"BulkGetKeys_Interval_95th": 0,
"BulkGetKeys_Interval_99th": 0,
"BulkGetTable_Interval_TotalOps": 0,
"BulkGetTable_Interval_TotalReq": 0,
"BulkGetTable_Interval_PerSec": 0,
"BulkGetTable_Interval_Min": 0,
"BulkGetTable_Interval_Max": 0,
"BulkGetTable_Interval_Avg": 0,
"BulkGetTable_Interval_95th": 0,
"BulkGetTable_Interval_99th": 0,
"BulkGetTableKeys_Interval_TotalOps": 0,
"BulkGetTableKeys_Interval_TotalReq": 0,
"BulkGetTableKeys_Interval_PerSec": 0,
"BulkGetTableKeys_Interval_Min": 0,
"BulkGetTableKeys_Interval_Max": 0,
"BulkGetTableKeys_Interval_Avg": 0,
"BulkGetTableKeys_Interval_95th": 0,
"BulkGetTableKeys_Interval_99th": 0,
"BulkGet_Cumulative_TotalOps": 0,
"BulkGet_Cumulative_TotalReq": 0,
"BulkGet_Cumulative_PerSec": 0,
"BulkGet_Cumulative_Min": 0,
"BulkGet_Cumulative_Max": 0,
"BulkGet_Cumulative_Avg": 0,
```

```
  "BulkGet_Cumulative_95th": 0,
  "BulkGet_Cumulative_99th": 0,
  "BulkGetKeys_Cumulative_TotalOps": 0,
  "BulkGetKeys_Cumulative_TotalReq": 0,
  "BulkGetKeys_Cumulative_PerSec": 0,
  "BulkGetKeys_Cumulative_Min": 0,
  "BulkGetKeys_Cumulative_Max": 0,
  "BulkGetKeys_Cumulative_Avg": 0,
  "BulkGetKeys_Cumulative_95th": 0,
  "BulkGetKeys_Cumulative_99th": 0,
  "BulkGetTable_Cumulative_TotalOps": 0,
  "BulkGetTable_Cumulative_TotalReq": 0,
  "BulkGetTable_Cumulative_PerSec": 0,
  "BulkGetTable_Cumulative_Min": 0,
  "BulkGetTable_Cumulative_Max": 0,
  "BulkGetTable_Cumulative_Avg": 0,
  "BulkGetTable_Cumulative_95th": 0,
  "BulkGetTable_Cumulative_99th": 0,
  "BulkGetTableKeys_Cumulative_TotalOps": 0,
  "BulkGetTableKeys_Cumulative_TotalReq": 0,
  "BulkGetTableKeys_Cumulative_PerSec": 0,
  "BulkGetTableKeys_Cumulative_Min": 0,
  "BulkGetTableKeys_Cumulative_Max": 0,
  "BulkGetTableKeys_Cumulative_Avg": 0,
  "BulkGetTableKeys_Cumulative_95th": 0,
  "BulkGetTableKeys_Cumulative_99th": 0
}
```

- New detailed statistics of single environment and replicated environment are available as follows:
  - `Type: oracle.kv.repnode.envmetric`
  - `User Data`: contains a full listing of detailed statistics for a given RN. The statistics are a string in JSON form, and are obtained through Notification.getUserData(). See the javadoc for EnvironmentStats and ReplicatedEnvironmentStats for more information about the meaning of the statistics.

    An example stat is: `getReplicaVLSNLagMap()` – Returns a map from replica node name to the lag, in VLSNs, between the replication state of the replica and the master, if known. Returns an empty map if this node is not the master.

  A sample statistics of single environment and replicated environment:

```
{
  "resource": "rg1-rn1",
  "shard": "rg1",
  "reportTime": 1498021100001,
  "FeederManager_nMaxReplicaLag": -1,
  "FeederManager_replicaLastCommitTimestampMap":
  "rg1-rn2=1498021098996;rg1-rn3=1498021096989",
  "FeederManager_nFeedersShutdown": 0,
  "FeederManager_nFeedersCreated": 2,
  "FeederManager_nMaxReplicaLagName": "rg1-rn2",
  "FeederManager_replicaVLSNLagMap": "rg1-rn2=0;rg1-rn3=2",
  "FeederManager_replicaVLSNRateMap": "rg1-rn2=472;rg1-rn3=472",
  "FeederManager_replicaDelayMap": "rg1-rn2=0;rg1-rn3=2007",
```

```
  "FeederManager_replicaLastCommitVLSNMap": "rg1-rn2=836;rg1-
rn3=834",
  "FeederTxns_txnsAcked": 77,
  "FeederTxns_lastCommitVLSN": 848,
  "FeederTxns_totalTxnMS": 228,
  "FeederTxns_lastCommitTimestamp": 1498021099030,
  "FeederTxns_vlsnRate": 439,
  "FeederTxns_txnsNotAcked": 0,
  "FeederTxns_ackWaitMS": 115,
  "Replay_nAborts": 0,
  "Replay_nGroupCommits": 0,
  "Replay_nNameLNs": 0,
  "Replay_nElapsedTxnTime": 0,
  "Replay_nMessageQueueOverflows": 0,
  "Replay_nGroupCommitMaxExceeded": 0,
  "Replay_nCommitSyncs": 0,
  "Replay_nCommitNoSyncs": 0,
  "Replay_maxCommitProcessingNanos": 0,
  "Replay_nGroupCommitTxns": 0,
  "Replay_nCommitWriteNoSyncs": 0,
  "Replay_minCommitProcessingNanos": 0,
  "Replay_nCommitAcks": 0,
  "Replay_nLNs": 0,
  "Replay_nCommits": 0,
  "Replay_latestCommitLagMs": 0,
  "Replay_totalCommitLagMs": 0,
  "Replay_totalCommitProcessingNanos": 0,
  "Replay_nGroupCommitTimeouts": 0,
  "ConsistencyTracker_nVLSNConsistencyWaitMS": 0,
  "ConsistencyTracker_nLagConsistencyWaits": 0,
  "ConsistencyTracker_nLagConsistencyWaitMS": 0,
  "ConsistencyTracker_nVLSNConsistencyWaits": 0,
  "BinaryProtocol_nMaxGroupedAcks": 0,
  "BinaryProtocol_messagesWrittenPerSecond": 19646,
  "BinaryProtocol_nEntriesOldVersion": 0,
  "BinaryProtocol_bytesReadPerSecond": 0,
  "BinaryProtocol_bytesWrittenPerSecond": 1057385,
  "BinaryProtocol_nMessagesWritten": 344,
  "BinaryProtocol_nGroupAckMessages": 0,
  "BinaryProtocol_nMessagesRead": 0,
  "BinaryProtocol_nReadNanos": 0,
  "BinaryProtocol_nMessageBatches": 24,
  "BinaryProtocol_nAckMessages": 0,
  "BinaryProtocol_nWriteNanos": 17509221,
  "BinaryProtocol_nBytesRead": 0,
  "BinaryProtocol_nGroupedAcks": 0,
  "BinaryProtocol_nMessagesBatched": 48,
  "BinaryProtocol_messagesReadPerSecond": 0,
  "BinaryProtocol_nBytesWritten": 18514,
  "VLSNIndex_nHeadBucketsDeleted": 0,
  "VLSNIndex_nBucketsCreated": 0,
  "VLSNIndex_nMisses": 0,
  "VLSNIndex_nTailBucketsDeleted": 0,
  "VLSNIndex_nHits": 19,
  "I/O_nRepeatFaultReads": 0,
```

```
"I/O_nRandomReads": 0,
"I/O_nLogIntervalExceeded": 0,
"I/O_nTempBufferWrites": 0,
"I/O_nWriteQueueOverflowFailures": 0,
"I/O_nGroupCommitWaits": 0,
"I/O_nGroupCommitRequests": 0,
"I/O_nWritesFromWriteQueue": 0,
"I/O_nSequentialWrites": 4,
"I/O_nGrpCommitTimeouts": 0,
"I/O_nFileOpens": 0,
"I/O_nRandomWrites": 0,
"I/O_bufferBytes": 4404016,
"I/O_nSequentialReadBytes": 0,
"I/O_endOfLog": 135573,
"I/O_nSequentialWriteBytes": 16693,
"I/O_nFSyncTime": 114,
"I/O_nSequentialReads": 0,
"I/O_nLogFSyncs": 1,
"I/O_nNoFreeBuffer": 0,
"I/O_nFSyncs": 0,
"I/O_nCacheMiss": 0,
"I/O_nWriteQueueOverflow": 0,
"I/O_nRandomWriteBytes": 0,
"I/O_nReadsFromWriteQueue": 0,
"I/O_nBytesReadFromWriteQueue": 0,
"I/O_nBytesWrittenFromWriteQueue": 0,
"I/O_nNotResident": 21,
"I/O_nFSyncRequests": 0,
"I/O_nRandomReadBytes": 0,
"I/O_nOpenFiles": 1,
"I/O_nLogBuffers": 16,
"I/O_nLogMaxGroupCommitThreshold": 0,
"I/O_nFSyncMaxTime": 114,
"Cache_nBytesEvictedCACHEMODE": 0,
"Cache_nINSparseTarget": 85,
"Cache_nINNoTarget": 81,
"Cache_dataAdminBytes": 48800,
"Cache_nBINsFetchMiss": 0,
"Cache_nNodesEvicted": 0,
"Cache_cacheTotalBytes": 5125248,
"Cache_nSharedCacheEnvironments": 0,
"Cache_nEvictionRuns": 0,
"Cache_lruMixedSize": 90,
"Cache_nLNsEvicted": 0,
"Cache_nBINsFetch": 92,
"Cache_nNodesMovedToDirtyLRU": 0,
"Cache_nLNsFetch": 1761,
"Cache_nBytesEvictedDAEMON": 0,
"Cache_nDirtyNodesEvicted": 0,
"Cache_nUpperINsFetchMiss": 0,
"Cache_nCachedBINs": 90,
"Cache_nNodesMutated": 0,
"Cache_nNodesStripped": 0,
"Cache_dataBytes": 686704,
"Cache_nFullBINsMiss": 0,
```

```
"Cache_nBINsFetchMissRatio": 0,
"Cache_nRootNodesEvicted": 0,
"Cache_nCachedBINDeltas": 0,
"Cache_nBytesEvictedMANUAL": 0,
"Cache_nNodesSkipped": 0,
"Cache_nUpperINsFetch": 0,
"Cache_nBinDeltaBlindOps": 0,
"Cache_nBytesEvictedCRITICAL": 0,
"Cache_lruDirtySize": 0,
"Cache_nLNsFetchMiss": 21,
"Cache_adminBytes": 589,
"Cache_nBINDeltasFetchMiss": 0,
"Cache_nThreadUnavailable": 0,
"Cache_nCachedUpperINs": 84,
"Cache_sharedCacheTotalBytes": 0,
"Cache_nNodesPutBack": 0,
"Cache_nBytesEvictedEVICTORTHREAD": 0,
"Cache_DOSBytes": 0,
"Cache_lockBytes": 33936,
"Cache_nNodesTargeted": 0,
"Cache_nINCompactKey": 7,
"OffHeap_offHeapCriticalNodesTargeted": 0,
"OffHeap_offHeapDirtyNodesEvicted": 0,
"OffHeap_offHeapNodesSkipped": 4,
"OffHeap_offHeapLNsEvicted": 44,
"OffHeap_offHeapAllocOverflow": 0,
"OffHeap_offHeapCachedLNs": 0,
"OffHeap_offHeapNodesStripped": 44,
"OffHeap_offHeapLruSize": 0,
"OffHeap_offHeapLNsStored": 44,
"OffHeap_offHeapLNsLoaded": 22,
"OffHeap_offHeapTotalBytes": 0,
"OffHeap_offHeapTotalBlocks": 0,
"OffHeap_offHeapNodesEvicted": 0,
"OffHeap_offHeapCachedBINDeltas": 0,
"OffHeap_offHeapNodesMutated": 0,
"OffHeap_offHeapNodesTargeted": 48,
"OffHeap_offHeapCachedBINs": 0,
"OffHeap_offHeapAllocFailure": 0,
"OffHeap_offHeapBINsLoaded": 0,
"OffHeap_offHeapThreadUnavailable": 63,
"OffHeap_offHeapBINsStored": 0,
"Cleaning_nBINDeltasMigrated": 0,
"Cleaning_minUtilization": 68,
"Cleaning_nLNsMigrated": 0,
"Cleaning_nINsCleaned": 0,
"Cleaning_nPendingLNsProcessed": 0,
"Cleaning_nToBeCleanedLNsProcessed": 0,
"Cleaning_nLNsLocked": 0,
"Cleaning_nRevisalRuns": 0,
"Cleaning_nPendingLNsLocked": 0,
"Cleaning_nTwoPassRuns": 0,
"Cleaning_nBINDeltasObsolete": 0,
"Cleaning_maxUtilization": 68,
"Cleaning_nLNsMarked": 0,
```

```
"Cleaning_pendingLNQueueSize": 0,
"Cleaning_nMarkLNsProcessed": 0,
"Cleaning_nRepeatIteratorReads": 0,
"Cleaning_nLNsExpired": 0,
"Cleaning_nCleanerRuns": 0,
"Cleaning_nBINDeltasDead": 0,
"Cleaning_nCleanerDisksReads": 0,
"Cleaning_protectedLogSizeMap": "",
"Cleaning_nCleanerDeletions": 0,
"Cleaning_nCleanerEntriesRead": 0,
"Cleaning_availableLogSize": 48942137344,
"Cleaning_nLNsDead": 0,
"Cleaning_nINsObsolete": 0,
"Cleaning_activeLogSize": 112716,
"Cleaning_nINsDead": 0,
"Cleaning_nINsMigrated": 0,
"Cleaning_totalLogSize": 112716,
"Cleaning_nBINDeltasCleaned": 0,
"Cleaning_nLNsObsolete": 0,
"Cleaning_nLNsCleaned": 0,
"Cleaning_nLNQueueHits": 0,
"Cleaning_reservedLogSize": 0,
"Cleaning_protectedLogSize": 0,
"Cleaning_nClusterLNsProcessed": 0,
"Node Compression_processedBins": 0,
"Node Compression_splitBins": 0,
"Node Compression_dbClosedBins": 0,
"Node Compression_cursorsBins": 0,
"Node Compression_nonEmptyBins": 0,
"Node Compression_inCompQueueSize": 0,
"Checkpoints_lastCheckpointInterval": 670,
"Checkpoints_nDeltaINFlush": 0,
"Checkpoints_lastCheckpointStart": 670,
"Checkpoints_lastCheckpointEnd": 1342,
"Checkpoints_nFullBINFlush": 0,
"Checkpoints_lastCheckpointId": 1,
"Checkpoints_nFullINFlush": 0,
"Checkpoints_nCheckpoints": 0,
"Environment_nBinDeltaInsert": 0,
"Environment_nBinDeltaUpdate": 0,
"Environment_nBinDeltaGet": 0,
"Environment_btreeRelatchesRequired": 0,
"Environment_nBinDeltaDelete": 0,
"Environment_environmentCreationTime": 1498021055255,
"Locks_nWaiters": 0,
"Locks_nRequests": 142,
"Locks_nLatchAcquiresSelfOwned": 0,
"Locks_nWriteLocks": 0,
"Locks_nTotalLocks": 303,
"Locks_nReadLocks": 303,
"Locks_nLatchAcquiresNoWaitSuccessful": 0,
"Locks_nOwners": 303,
"Locks_nLatchAcquiresWithContention": 0,
"Locks_nLatchAcquireNoWaitUnsuccessful": 0,
"Locks_nLatchReleases": 0,
```

```
  "Locks_nLatchAcquiresNoWaiters": 0,
  "Locks_nWaits": 0,
  "Op_secSearchFail": 0,
  "Op_priDelete": 0,
  "Op_priSearchFail": 14,
  "Op_secPosition": 0,
  "Op_priInsertFail": 0,
  "Op_priDeleteFail": 0,
  "Op_secSearch": 0,
  "Op_priSearch": 54,
  "Op_priPosition": 2,
  "Op_secDelete": 0,
  "Op_secUpdate": 0,
  "Op_secInsert": 0,
  "Op_priUpdate": 11,
  "Op_priInsert": 66
}
```

- Announce a change in this RepNode's replication state.
  - `Type: oracle.kv.repnode.replicationstate`
  - `User Data`: RN replication state change event. The event is a string in JSON form, and is obtained via Notification.getUserData().

  For example:

```
{"resource":"rg1-rn3","shard":"rg1","reportTime":1476980297641,
"replication_state":"MASTER"}
```

- Announce a change in this RepNode's service status.
  - `Type: oracle.kv.repnode.status`
  - `User Data`: RN service status change event. The event is a string in JSON form, and is obtained via Notification.getUserData().

  For example:

```
{"resource":"rg3-rn3","shard":"rg3","reportTime":1476981010202,
"service_status":"ERROR_RESTARTING"}
```

- Announce a plan state change.
  - `Type: oracle.kv.plan.status`
  - `User Data`: Plan status change event. The event is a string in JSON form, and is obtained via Notification.getUserData().

  For example:

```
{"planId":7,"planName":"Change Global Params
(7)","reportTime":1477272558763,"state":"SUCCEEDED","attemptNumber":
1,
"message":"Plan finished."}
```

Monitoring for Replication Nodes (RN)

Each Storage Node hosts one or more Replication Nodes which stores the data in key-value pairs. For more information, see Replication Nodes and Shards in the *Concepts Guide*.

See the following section:

- Metrics for Replication Node

Metrics for Replication Node

- repNodeServiceStatus – The current status of the Replication Node. They are as follows:
  - starting (1) – The storage node agent is booting up.
  - waitingForDeploy (2) – The Replication Node is waiting to be registered with the Storage Node Agent.
  - running(3) – The replication node is running.
  - stopping(4) – The replication node is in the process of shutting down.
  - stopped(5) – An intentional clean shutdown.
  - errorRestarting(6) – The Replication Node is restarting after encountering an error.
  - errorNoRestart(7) – Service is in an error state, will not restart automatically, and the service requires Administrative intervention. The user can search for SEVERE entries in both the log file for the Replication Node and the log file of the SNA controlling the failed service. The service's log in Monitoring for RN section is RN log:

    ```
    <kvroot>/<storename>/log/rg*-rn*_*.log
    ```

    where, <kvroot> and <storename> are user inputs and * represents the number of the log. For example: rg3-rn2_0.log is the latest log, rg3-rn2_1.log is previous log.

    Note that the kvroot and storename will be different for every installation. Similarly, to find the log file for SNA, use:

    ```
    <kvroot>/<storename>/log/sn*_*.log
    ```

    Examples of SN logs can be: sn1_0.log, sn1_1.log.
    You can search SEVERE keyword in these log files, and then read the searched messages to fix the errors, or you may require help from Oracle NoSQL Database support. The action to take depends on the nature of the failure and can vary from stopping and restarting the service explicitly (easy) to the need to replace the service instance entirely (not easy and slow). The issues can be any of the following:

    * Resource issue – Some type of necessary resource for example, disk space, memory, or network is not available.
    * Configuration problem – Some configuration-related issues which needs a fix.
    * Software bug – Bugs in the code which needs Oracle NoSQL Database support.
    * On disk corruption – Something in persistent storage has been corrupted.

    Note that the corruption situations are difficult to handle, but this is rare and require help from Oracle NoSQL Database support.

- unreachable(8) – The Replication Node is unreachable by the admin service.

> **Note:**
>
> If a Storage Node is UNREACHABLE, or a Replication Node is having problems and its Storage Node is UNREACHABLE, the first thing to check is the network connectivity between the Admin and the Storage Node. However, if the managing Storage Node Agent is reachable and the managed Replication Node is not, we can guess that the network is OK and the problem lies elsewhere.

- expectedRestarting(9) – The Replication Node is executing an expected restart as some plan CLI commands causes a component to restart. This is an expected restart, that is different from errorRestarting(6) (which is a restart after encountering an error).

The following metrics can be monitored to get a sense for the performance of each Replication Node in the cluster. There are two flavors of metric granularity:

- Interval – By default, each node in the cluster will sample performance data every 20 seconds and aggregate the metrics to this interval. This interval may be changed using the admin plan change-parameters - global and supplying the collectorInterval parameter with a new value (see Changing Parameters).
- Cumulative – Metrics that have been collected and aggregated since the node has started.

The metrics are further broken down into measurements for operations over single keys versus operations over multiple keys.

> **Note:**
>
> All timestamp metrics are in UTC, therefore appropriate conversion to a time zone relevant to where the store is deployed is necessary.

- repNodeIntervalStart – The start timestamp of when this sample of single key operation measurements were collected.
- repNodeIntervalEnd –The start timestamp of when this sample of single key operation measurements were collected.
- repNodeIntervalTotalOps – Total number of single key operations (get, put, delete) processed by the Replication Node in the interval being measured.
- repNodeIntervalThroughput – Number of single key operations (get, put, delete) per second completed during the interval being measured.
- repNodeIntervalLatMin – The minimum latency sample of single key operations (get, put, delete) during the interval being measured.
- repNodeIntervalLatMax – The maximum latency sample of single key operations (get, put, delete) during the interval being measured.
- repNodeIntervalLatAvg – The average latency sample of single key operations (get, put, delete) during the interval being measured (returned as a float).

- repNodeIntervalPct95 – The 95th percentile of the latency sample of single key operations (get, put, delete) during the interval being measured.
- repNodeIntervalPct99 – The 95th percentile of the latency sample of single key operations (get, put, delete) during the interval being measured.
- repNodeCumulativeStart – The start timestamp of when the replication started collecting cumulative performance metrics (all the below metrics that are cumulative).
- repNodeCumulativeEnd – The end timestamp of when the replication ended collecting cumulative performance metrics (all the below metrics that are cumulative).
- repNodeCumulativeTotalOps – The total number of single key operations that have been processed by the Replication Node.
- repNodeCumulativeThroughput – The sustained operations per second of single key operations measured by this node since it has started.
- repNodeCumulativeLatMin – The minimum latency of single key operations measured by this node since it has started.
- repNodeCumulativeLatMax – The maximum latency of single key operations measured by this node since it has started.
- repNodeCumulativeLatAvg – The average latency of single key operations measured by this node since it has started (returned as a float).
- repNodeCumulativePct95 – The 95th percentile of the latency of single key operations (get, put, delete) since it has started.
- repNodeCumulativePct99 – The 99th percentile of the latency of single key operations (get, put, delete) since it has started.
- repNodeMultiIntervalStart – The start timestamp of when this sample of multiple key operation measurements were collected.
- repNodeMultiIntervalEnd – The end timestamp of when this sample of multiple key operation measurements were collected.
- repNodeMultiIntervalTotalOps – Total number of multiple key operations (execute) processed by the replication node in the interval being measured.
- repNodeMultiIntervalThroughput – Number of multiple key operations (execute) per second completed during the interval being measured.
- repNodeMultiIntervalLatMin – The minimum latency sample of multiple key operations (execute) during the interval being measured.
- repNodeMultiIntervalLatMax – The maximum latency sample of multiple key operations (execute) during the interval being measured.
- repNodeMultiIntervalLatAvg – The average latency sample of multiple key operations (execute) during the interval being measured (returned as a float).
- repNodeMultiIntervalPct95 – The 95th percentile of the latency sample of multiple key operations (execute) during the interval being measured.
- repNodeMultiIntervalPct99 – The 95th percentile of the latency sample of multiple key operations (execute) during the interval being measured.
- repNodeMultiIntervalTotalRequests – The total number of multiple key operations (execute) during the interval being measured.
- repNodeMultiCumulativeStart – The start timestamp of when the Replication Node started collecting cumulative multiple key performance metrics (all the below metrics that are cumulative).

- repNodeMultiCumulativeEnd – The end timestamp of when the Replication Node started collecting cumulative multiple key performance metrics (all the below metrics that are cumulative).
- repNodeMultiCumulativeTotalOps – The total number of single multiple operations that have been processed by the Replication Node since it has started.
- repNodeMultiCumulativeThroughput – The sustained operations per second of multiple key operations measured by this node since it has started.
- repNodeMultiCumulativeLatMin – The minimum latency of multiple key operations (execute) measured by this node since it has started.
- repNodeMultiCumulativeLatMax – The maximum latency of multiple key operations (execute) measured by this node since it has started.
- repNodeMultiCumulativeLatAvg – The average latency of multiple key operations (execute) measured by this node since it has started (returned as a float).
- repNodeMultiCumulativePct95 – The 95th percentile of the latency of multiple key operations (execute) since it has started.
- repNodeMultiCumulativePct99 – The 99th percentile of the latency of multiple key operations (execute) since it has started.
- repNodeMultiCumulativeTotalRequests – The total number of multiple key operations measured by this node since it has started.
- repNodeCommitLag – The average commit lag (in milliseconds) for a given Replication Node's update operations during a given time interval.
- repNodeCacheSize – The size in bytes of the replication node's cache of B-tree nodes, which is calculated using the DBCacheSize utility.
- repNodeConfigProperties – The set of configuration name/value pairs that the Replication Node is currently running with. Each parameter is a constant which has a string value. The string value is used to set the parameter in KVSTORE. For example, the parameter CHECKPOINTER_BYTES_INTERVAL has je.checkpointer.bytesInterval string value in the javadoc (see, here). The document also details on the data type, minimum, maximum time, etc.
- repNodeCollectEnvStats – True or false depending on whether the Replication Node is currently collecting performance statistics.
- repNodeStatsInterval – The interval (in seconds) that the Replication Node is utilizing for aggregate statistics.
- repNodeMaxTrackedLatency – The maximum number of milliseconds for which latency statistics will be tracked. For example, if this parameter is set to 1000, then any operation at the repnode that exhibits a latency of 1000 or greater milliseconds is not put into the array of metric samples for subsequent reporting.
- repNodeJavaMiscParams – The value of the -Xms, -Xmx, and -XX:ParallelGCThreads= as encountered when the Java VM running this Replication Node was booted.
- repNodeLoggingConfigProps – The value of the loggingConfigProps parameter as encountered when the Java VM running this Replication Node was booted.
- repNodeHeapMB – The size of the Java heap for this Replication Node, in MB.
- repNodeMountPoint – The path to the file system mount point where this Replication Node's files are stored.

- repNodeMountPointSize – The size of the file system mount point where this Replication Node's files are stored.
- repNodeHeapSize – The current value of –Xmx for this Replication Node.
- repNodeLatencyCeiling – The upper bound (in milliseconds) at which latency samples may be gathered at this Replication Node before an alert is generated. For example, if this is set to 3, then any latency sample above 3 generates an alert.
- repNodeCommitLagThreshold – If the average commit lag (in milliseconds) for a given Replication Node during a given time interval exceeds the value returned by this method, an alert is generated.
- repNodeReplicationState – The replication state of the node.
- repNodeThroughputFloor – The lower bound (in operations per second) at which throughput samples may be gathered at this Replication Node before an alert is generated. For example, if this is set to 300,000, then any throughput calculation at this Replication Node that is lower than 300,000 operations per seconds generates an alert.

Monitoring for Arbiter Nodes

An Arbiter Node is a lightweight process that participates in electing a new master when the old master becomes unavailable. For more information, see Arbiter Nodes in the *Concepts Guide*.

See the following section:

- Metrics for Arbiter Nodes

Metrics for Arbiter Nodes

- arbNodeServiceStatus – The current status of the Arbiter Node. They are as follows:
  - starting (1) – The Storage Node Agent is booting up.
  - waitingForDeploy (2) – The Arbiter Node is waiting to be registered with the Storage Node Agent.
  - running(3) – The Arbiter Node is running.
  - stopping(4) – The Arbiter Node is in the process of shutting down.
  - stopped(5) – An intentional clean shutdown.
  - errorRestarting(6) – The Arbiter Node is restarting after encountering an error.
  - errorNoRestart(7) – Service is in an error state and will not be automatically restarted. Administrative intervention is required. The user can search for SEVERE entries in both the service's log file and the log file of the SNA controlling the failed service. The service's log in Monitoring for Arbiter section is Arbiter log:

    ```
    <kvroot>/<storename>/log/rg*_an1_*.log
    ```

    where, <kvroot> and <storename> are user inputs and * represents the number of the log.

    Note that the kvroot and storename will be different for every installation. Similarly, to find the log file for SNA, use:

    ```
    <kvroot>/<storename>/log/sn*_*.log
    ```

Examples of SN logs can be: sn1_0.log, sn1_1.log.

You can search SEVERE keyword in these log files, and then read the searched messages to fix the errors, or you may require help from Oracle NoSQL Database support. The action to take depends on the nature of the failure and can vary from stopping and restarting the service explicitly (easy) to the need to replace the service instance entirely (not easy and slow). The issues can be any of the following:

* Resource issue – Some type of necessary resource for example, disk space, memory, or network is not available.
* Configuration problem – Some configuration-related issues which needs a fix.
* Software bug – Bugs in the code which needs Oracle NoSQL Database support.
* On disk corruption – Something in persistent storage has been corrupted.

Note that the corruption situations are difficult to handle, but this is rare and require help from Oracle NoSQL Database support.

– unreachable(8) – The Arbiter Node is unreachable by the admin service.

> **Note:**
>
> If a Storage Node is UNREACHABLE, or an Admin Node is having problems and its Storage Node is UNREACHABLE, the first thing to check is the network connectivity between the Admin and the Storage Node. However, if the managing Storage Node Agent is reachable and the managed Arbiter Node is not, we can guess that the network is OK and the problem lies elsewhere.

– expectedRestarting(9) – The Arbiter Node is executing an expected restart as some plan CLI commands causes a component to restart. This is an expected restart, that is different from errorRestarting(6) (which is a restart after encountering an error).

> **Note:**
>
> All timestamp metrics are in UTC, therefore appropriate conversion to a time zone relevant to where the store is deployed is necessary.

- arbNodeConfigProperties – The set of configuration name/value pairs that the Arbiter Node is currently running with. This is analogous to the Replication Node.
- arbNodeJavaMiscParams – The value of the -Xms, -Xmx, and -XX:ParallelGCThreads= as encountered when the Java VM running this Arbiter Node was booted.
- arbNodeLoggingConfigProps – The value of the loggingConfigProps parameter as encountered when the Java VM running this Arbiter Node was booted.
- arbNodeCollectEnvStats – True or false depending on whether the Arbiter Node is currently collecting performance statistics.

- arbNodeStatsInterval – The interval (in seconds) that the Arbiter Node is utilizing for aggregate statistics.
- arbNodeHeapMB – The size of the Java heap for this Arbiter Node, in MB.
- arbNodeAcks – The number of transactions acked.
- arbNodeMaster – The current master.
- arbNodeState – The replication state of the node. An Arbiter has an associated replication state (analogous to the replication node state). The state diagram is UNKNOWN <-> REPLICA -> DETACHED.
- arbNodeVLSN – The current acked VLSN. Arbiters track the VLSN/DTVLSN of the transaction commit that the Arbiter acknowledges. This is the highest VLSN value that the Arbiter acknowledged.
- arbNodeReplayQueueOverflow – The current replayQueueOverflow value. The arbNodeReplayQueueOverflow statistic is incremented when the Arbiter is not able to process acknowledgement requests fast enough to prevent the thread reading from the network to wait for free space in the queue. The RepParms.REPLICA_MESSAGE_QUEUE_SIZE is used to specify the maximum number of entries that the queue can hold. The default is 1000 entries. A high arbNodeReplayQueueOverflow value may indicate that the queue size is too small or that the Arbiter is not able to process requests as fast as the system load requires.

Monitoring for Administration (Admin) Nodes

The Administrative (Admin) Node is a process running in the Storage Node, that is used to configure, deploy, monitor, and change store components. The Administrative Node handles the execution of commands from the Administrative Command Line Interface (CLI). For more information, see Administration in the *Concepts Guide*.

See the following section:

- Metrics for Admin Nodes

Metrics for Admin Nodes

The following metrics are accessible through JMX for monitoring Administrative Nodes in the Oracle NoSQL Database cluster.

- adminId – The unique ID for the Admin Node.
- adminServiceStatus – The status of the administrative service. It can be one of the follows:
  - unreachable(0) – The Admin Node is unreachable. This can be due to a network error or the Admin Node maybe down.
  - starting (1) – The Admin Node agent is booting up.
  - waitingForDeploy (2) – Indicates a bootstrap admin that has not been configured, that is, it has not been given a store name. Configuring the admin triggers the creation of the Admin database, and changes its status from "WAITING_FOR_DEPLOY" to "RUNNING".
  - running(3) – The Admin Node is running.
  - stopping(4) – The Admin Node in the process of shutting down.
  - stopped(5) – An intentional clean shutdown of the Admin Node.

- errorRestarting(6) – The Storage Node tried to start the admin several times without success and gave up.
- errorNoRestart(7) – Service is in an error state and will not be automatically restarted. Administrative intervention is required. The user can start looking for SEVERE entries in both the service's log file and the log file of the SNA controlling the failed service. The service's log in Monitoring for Admin section is Admin log:

  ```
  <kvroot>/<storename>/log/admin*_*.log
  ```

  where, <kvroot> and <storename> are user inputs and * represents the number of the log.

  Note that the kvroot and storename will be different for every installation. Similarly, to find the log file for SNA, use:

  ```
  <kvroot>/<storename>/log/sn*_*.log
  ```

  Examples of SN logs can be: sn1_0.log, sn1_1.log.
  You can search SEVERE keyword in these log files, and then read the searched messages to fix the errors, or you may require help from Oracle NoSQL Database support. The action to take depends on the nature of the failure and can vary from stopping and restarting the service explicitly (easy) to the need to replace the service instance entirely (not easy and slow). The issues can be any of the following:

  * Resource issue – Some type of necessary resource for example, disk space, memory, or network is not available.
  * Configuration problem – Some configuration-related issues which needs a fix.
  * Software bug – Bugs in the code which needs Oracle NoSQL Database support.
  * On disk corruption – Something in persistent storage has been corrupted.

  In the rare case that you discover disk corruption, you must get help from Oracle NoSQL Database support.
- expectedRestarting(9) – The Admin Node is executing an expected restart as some plan CLI commands causes a component to restart. This is an expected restart, that is different from errorRestarting(6) (which is a restart after encountering an error).

- adminLogFileCount – A logging config parameter that represents the maximum number of log files that are retained by the Admin Node. Users can change the value of this parameter, and also the `adminLogFileLimit` parameter, if they want to reduce the amount of disk space used by debug log files. Note that reducing the amount of debug log data saved may make it harder to debug problems if debug information is deleted before the problem is noticed. For more information on `adminLogFileCount`, see Admin Parameters and Admin Restart.
- adminLogFileLimit – A logging config parameter that represents the maximum size of a single log file in bytes. For more information on `adminLogFileLimit`, see Admin Parameters and Admin Restart.

- adminPollPeriod – The frequency by which the Admin polls agents (Replication Node and Storage Node Agent) for statistics. This polling receives service status changes, performance metrics, and log messages. This period is reported in units of milliseconds.
- adminEventExpiryAge – Tells how long critical events are saved in the admin database. This value is reported in units of hours.
- adminIsMaster – A Boolean value which indicates whether or not this Admin Node is the master node for the admin group.

Hardware Monitoring

While software component monitoring is central to insuring that high availability service levels are met, hardware monitoring, fault isolation, and ultimately the replacement of a failed component and how to recover from that failure are equally important. The following sections cover guidelines on what to monitor and how to detect potential hardware failures. It also discusses the replacement procedures of replacing failed hardware components and how to bring the Oracle NoSQL Database components (that were utilizing the components that were replaced) back online.

Monitoring for Hardware Faults

There are several different hardware scenarios/failures that are considered when monitoring the environment for Oracle NoSQL Database. The sections below cover the monitoring of network, disk, and machine failures as well as the correlation of these failures with log events in the Oracle NoSQL Database. Finally, it discusses how to recover from these failure scenarios.

The Network

Monitoring packet loss, round trip average latencies, and network utilization provides a glimpse into critical network activity that can affect the performance as well as the ongoing functioning of the Oracle NoSQL Database. There are two critical types of network activity in the Oracle NoSQL Database. The client driver will utilize Java RMI over TCP/IP to communicate between the machine running the application, and the machines running the nodes of the NoSQL Database cluster. Secondly, each node in the cluster must be able to communicate with each other. Replication Nodes will utilize Java RMI over TCP/IP and will also utilize streams based communication over TCP/IP. Administrative nodes and Storage Node agents will only utilize RMI over TCP/IP. The key issue in insuring an operational store that is able to maintain predictable latencies and throughput is to monitor the health of the network through which all of these nodes communicate.

The following tools are recommended for monitoring the health of the network interfaces that the Oracle NoSQL Database relies on:

- **Sar, ping, iptraf** – These operating system tools display critical network statistics such as # of packets lost, round trip latency, and network utilization. It is recommended to use `ping` in a scripted fashion to monitor round trip latency as well as packet loss and use either `sar` or `iptraf` in a scripted fashion to monitor network utilization. A good rule of thumb is to raise an alert if network utilization goes above 80%.
- **Oracle NoSQL Ping** command – The ping command attempts to contact each node of the cluster. Directions on how to run and script this command can be found here: CLI Command Reference.

Correlating Network Failure to NoSQL Log Events

Network failures that affect the runtime operation of NoSQL Database is ultimately logged as instances of Java runtime exceptions. Using log file monitoring, the following exception strings are added to a list of regular expressions that are recognized as critical events. Correlating the timestamps of these events with the timestamps of whatever network monitoring tool is being utilized.

> **Note:**
>
> While searching the log file for any of the exceptions stated below, the log level must also be checked such that only log levels of SEVERE is considered. These exceptions are logged at a level of INFO which indicates no errors will be encountered by the application.

- **UnknownHostException** – A DNS lookup of a node in the NoSQL Database failed due to either a misconfigured NoSQL Database or a DNS error. Encountering this error after a NoSQL cluster has been operational for some time indicates a network failure between the application and the DNS server.
- **ConnectException** – The client driver cannot open a connection to the NoSQL Database node. Either the node is not listening on the port being contacted or the port is blocked by a firewall.
- **ConnectIOException** – Indicates a possible handshake error between the client and the server or an I/O error from the network layer.
- **MarshalException** – Indicates a possible I/O error from the network layer.
- **UnmarshalException** – Indicates a possible I/O error from the network layer.
- **NoSuchObjectException** – Indicates a possible I/O error from the network layer.
- **RemoteException** – Indicates a possible I/O error from the network layer.

Recovering from Network Failure

In general, the NoSQL Database will retry and recover from network failures and no intervention at the database level is necessary. It is possible that a degraded level of service is encountered due to the network failure; however, the failure of network partitions will not cause the NoSQL Database to fail.

Persistent Storage

One of the most common failure scenarios you can expect to encounter while managing a deployed Oracle NoSQL Database instance (sometimes referred to as KVStore) is a disk that fails and needs to be replaced; where the disk is typically a hard disk drive (HDD), or a solid state drive (SSD). Because HDDs employ many moving parts that are continuously in action when the store performs numerous writes and reads, moving huge numbers of bytes on and off the disk, parts of the disk can easily wear out and fail. With respect to SSDs, although the absence of moving parts makes SSDs a bit less failure prone than HDDs, when placed under very heavy load, SSDs will also generally fail with regularity. As a matter of fact, when such stores scale to a very large number of nodes (machines), a point can be reached where disk failure is virtually guaranteed; much more than other hardware components making up a

node. For example, disks associated with such systems generally fail much more frequently than the system's mother board, memory chips, or even the network interface cards (NICs).

Since disk failures are so common, a well-defined procedure is provided for replacing a failed disk while the store continues to run; providing data availability.

Detecting and Correlating Persistent Storage Failures to NoSQL Log Events

There are many vendor specific tools for detecting the failure of persistent storage devices. It is beyond the scope of this book to recommend any vendor specific mechanism. There are however, some general things that can be done to identify a failed persistent storage device;

> **Note:**
>
> Using log file monitoring, the following exception string is to a list of regular expressions that should be recognized as critical events. Correlating the timestamps of these events with the timestamps of whatever storage device monitoring tool is being utilized. When searching the log file for any of the exception stated below, the log level must also be checked such that only log levels of SEVERE is considered.

- **I/O errors in /var/log/messages** – Monitoring /var/log/messages for I/O errors indicate that something is wrong with the device and it may be failing.
- **Smartctl** – If available, the smartctl tool detects a failure with a persistent storage device and displays the serial number of the specific device that is failing.
- **EnvironmentFailureException** – The storage layer of NoSQL Database (Berkeley DB Java Edition) converts Java IOExceptions detected from the storage device into an EnvironmentFailureException and this exception is written to the log file.

Resolving Storage Device Failures

The sections below describe that procedure for two common machine configurations.

In order to understand how a failed disk can be replaced while the KVStore is running, review what and where data is stored by the KVStore; which is dependent on each machine's disk configuration, as well as how the store's capacity and storage directory location is configured. Suppose a KVStore is distributed among 3 machines – or Storage Nodes (SNs) — and is configured with replication factor (RF) equal to 3, each SN's capacity equal to 2, KVROOT equal to /opt/ondb/var/kvroot, and store name equal to "store-name". Since the capacity or each SN is 2, each machine will host 2 Replication Nodes (RNs). That is, each SN will execute 2 Java VMs and each run a software service (an RN service) responsible for storing and retrieving a replicated instance of the key/value data maintained by the store.

Suppose in one deployment, the machines themselves (the SNs) are each configured with 3 disks; whereas in another deployment, the SNs each have only a single disk on which to write and read data. Although the second (single disk) scenario is fine for experimentation and "tire kicking", that configuration is strongly discouraged for production environments, where it is likely to have disk failure and replacement. In particular, one rule deployers are encouraged to follow in production environments is that multiple RN services should never be configured to write data to the same disk. That said, there may be some uncommon circumstances in which a deployer may choose to violate this rule. For example, in addition to being extremely reliable (for example, a RAID device), the disk may be a device with such high performance and large capacity that a single RN service would never be able to make

use of the disk without exceeding the recommended 32GB heap limit. Thus, unless the environment consists of disks that satisfy such uncommon criteria, deployers always prefer environments that allow them to configure each RN service with its own disk; separate from all configuration and administration information, as well as the data stored by any other RN services running on the system.

As explained below, to configure a KVStore use multiple disks on each SN, the storagedir parameter must be employed to exploit the separate media that is available. In addition to encouraging deployers to use the storagedir parameter in the multi-disk scenario, this note is also biased toward the use of that parameter when discussing the single disk scenario; even though the use of that parameter in the single disk case provides no substantial benefit over using the default location (other than the ability to develop common deployment scripts). To understand this, first compare the implications of using the default storage location with a non-default location specified with the storagedir parameter.

Thus, suppose the KVStore is deployed – in either the multi-disk scenario or the single disk scenario – using the default location; that is, the storagedir parameter is left unspecified. This means that data will be stored in either scenario under the KVROOT; which is `/opt/ondb/var/kvroot` in the examples below. For either scenario, a directory structure like the following is created and populated:

```
 - Machine 1 (SN1) -     - Machine 2 (SN2) -    - Machine 3 (SN3) -
/opt/ondb/var/kvroot   /opt/ondb/var/kvroot  /opt/ondb/var/kvroot
  log files             log files             log files
  /store-name           /store-name           /store-name
    /log                   /log                  /log
    /sn1                   /sn2                  /sn3
      config.xml             config.xml            config.xml
      /admin1                /admin2               /admin3
        /env                   /env                  /env

  /rg1-rn1                 /rg1-rn2                /rg1-rn3
    /env                     /env                    /env

  /rg2-rn1                 /rg2-rn2                /rg2-rn3
    /env                     /env                    /env
```

Compare this with the structure that is created when a KVStore is deployed to the multi-disk machines; where each machine's 3 disks are named /opt, /disk1, and/disk2. Assume that the makebootconfig utility (described in Chapter 2 of the Oracle NoSQL Database Administrator's Guide, section, "Installation Configuration") is used to create an initial boot config with parameters such as the following:

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar makebootconfig \
       -root /opt/ondb/var/kvroot \
       -port 5000  \
       -host <host-ip>
       -harange 5010,5020 \
       -num_cpus 0  \
       -memory_mb 0 \
       -capacity 2  \
       -admindir /opt/ondb/var/admin \
       -storagedir /disk1/ondb/data \
```

```
-storagedir /disk2/ondb/data \
-rnlogdir /disk1/ondb/rnlog \
-storagedir /disk2/ondb/rnlog
```

With a boot config such as that shown above, the directory structure that is created and populated on each machine would then be:

```
 - Machine 1 (SN1) -     - Machine 2 (SN2) -    - Machine 3 (SN3) -
/opt/ondb/var/kvroot   /opt/ondb/var/kvroot  /opt/ondb/var/kvroot
  log files             log files             log files
  /store-name           /store-name           /store-name
    /log                   /log                  /log
    /sn1                   /sn2                  /sn3
      config.xml             config.xml            config.xml
      /admin1                /admin2               /admin3
        /env                   /env                  /env

/disk1/ondb/data         /disk1/ondb/data        /disk1/ondb/data
  /rg1-rn1                 /rg1-rn2                /rg1-rn3
    /env                     /env                    /env

/disk2/ondb/data         /disk2/ondb/data        /disk2/ondb/data
  /rg2-rn1                 /rg2-rn2                /rg2-rn3
    /env                     /env                    /env
```

In this case, the configuration information and administrative data is stored in a location that is separate from all of the replication data. Furthermore, the replication data itself is stored by each distinct RN service on separate, physical media as well. That is, the data stored by a given member of each replication group (or shard) is stored on a disk that separate from the disks employed by the other members of the group.

> **Note:**
>
> Storing the data in these different locations as described above, provides for failure isolation and will typically make disk replacement less complicated and less time consuming. That is, by using a larger number of smaller disks, it is possible to recover much more quickly from a single disk failure because of the reduced amount of time it will take to repopulate the smaller disk. This is why both this note and Chapter 2 of the Oracle NoSQL Database Administrator's Guide, section, "Installation Configuration" strongly encourage configurations like that shown above; configurations that exploit separate physical media or disk partitions.

Even when a machine has only a single disk, nothing prevents the deployer from using the storagedir parameter in a manner similar to the multi-disk case; storing the configuration and administrative data under a parent directory that is different than the parent(s) under which the replicated data is stored. Since this non-default strategy may allow to create deployment scripts that can be more easily shared between single disk and multi-disk systems, some may prefer this strategy over using the default location (KVROOT); or may simply view it as a good habit to follow. Employing this non-default strategy is simply a matter of taste, and provides no additional benefit other than uniformity with the multi-disk case.

Hence, such a strategy applied to a single disk system will not necessarily make disk replacement less complicated; because, if that single disk fails and needs to be replaced, not only is all the data written by the RN(s) unavailable, but the configuration (and admin) data is also unavailable. As a result, since the configuration information is needed during the (RN) recovery process after the disk has been replaced, that data must be restored from a previously captured backup; which can make the disk replacement process much more complicated. This is why multi-disk systems are generally preferred in production environments; where, because of sheer use, the data disks are far more likely to fail than the disk holding only the configuration and other system data.

Procedure for Replacing a Failed Persistent Storage Device

Suppose a KVStore has been deployed to a set of machines, each with 3 disks, using the 'storagedir' parameter as described above. Suppose that disk2 on SN3 fails and needs to be replaced. In this case, the administrator would do the following:

1. Execute the KVStore administrative command line interface (CLI), connecting via one of the healthy admin services.
2. From the CLI, execute the following command:

   ```
   kv-> plan stop-service-service rg2-rn3
   ```

   This stops the service so that attempts by the system to communicate with that particular service are no longer necessary; resulting in a reduction in the amount of error output related to a failure the administrator is already aware of.
3. Remove disk2, using whatever procedure is dictated by the OS, the disk manufacture, and/or the hardware platform.
4. Install a new disk using the appropriate procedures.
5. Format the new disk to have the same storage directory as before; that is, `/disk2/ondb/var/kvroot`
6. From the CLI, execute the following commands; where the `verify configuration` command simply verifies that the desired RN is now up and running:

   ```
   kv-> plan start-service -service rg2-rn3 -wait
   kv-> verify configuration
   ```
7. Verify that the recovered RN data file(s) have the expected content; that is, `/disk2/ondb/var/kvroot/rg2-rn3/env/*.jdb`

In step 2, the RN service with id equal to 3, belonging to the replication group with id2, is stopped (rg2-rn3). To determine which specific RN service to stop when using the procedure outlined above, the administrator combines knowledge of which disk has failed on which machine with knowledge about the directory structure created during deployment of the KVStore. For this particular case, the administrator has first used standard system monitoring and management mechanisms to determine that disk2 has failed on the machine corresponding to the SN with id equal to 3 and needs to be replaced. Then, given the directory structure shown previously, the administrator knows that – for this deployment – the store writes replicated data to disk2 on the SN3 machine using files located under, `/disk2b/data/rg2-rn3/en`. As a result, the administrator determined that the RN service with name equal to rg2-rn3 must be stopped before replacing the failed disk.

In step 6, if the RN service that was previously stopped has successfully restarted when the `verify configuration` command is executed, and although the command's output indicates that the service is up and healthy, it is not necessary that the restarted RN has completely repopulated the new disk with that RN's data. This is because, it could take a considerable amount of time for the disk to recover all its data; depending on the amount of data that previously resided on the disk before failure. The system may encounter additional network traffic and load while the new disk is being repopulated.

Finally, it should be noted that step 7 is just a sanity check, and therefore optional. That is, if the RN service is successfully restarted and the `verify configuration` command reports RN as healthy, the results of that command is viewed as sufficient evidence for declaring the disk replacement a success. As indicated above, even if some data is not yet available on the new disk, that data will continue to be available via the other members of the recovering RN's replication group (shard), and will eventually be replicated to, and available from, the new disk as expected.

Example

Below, an example is presented that allows you to gain some practical experience with the disk replacement steps presented above. This example is intended to simulate the multi-disk scenario using a single machine with a single disk. Thus, no disks will actually fail or be physically replaced. But you should still feel how the data is automatically recovered when a disk is replaced.

For simplicity, assume that the KVStore is installed under `/opt/ondb/kv`; that is, `KVHOME=/opt/ondb/kv`, and that `KVROOT=/opt/ondb/var/kvroot`; that is, if you have not done so already, create the directory:

```
> mkdir -p /opt/ondb/var/kvroot
```

To simulate the data disks, create the following directories:

```
> mkdir -p /tmp/sn1/disk1/ondb/data
> mkdir -p /tmp/sn1/disk2/ondb/data

> mkdir -p /tmp/sn2/disk1/ondb/data
> mkdir -p /tmp/sn2/disk2/ondb/data

> mkdir -p /tmp/sn3/disk1/ondb/data
> mkdir -p /tmp/sn3/disk2/ondb/data
```

Next, open 3 windows; Win_A, Win_B, and Win_C, which will represent the 3 machines (SNs). In each window, execute the `makebootconfig` command, creating a different, but similar, boot config for each SN that will be configured.

On Win_A

```
java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar makebootconfig \
     -root /opt/ondb/var/kvroot \
     -host <host-ip> \
     -config config1.xml \
     -port 13230 \
     -harange 13232,13235 \
```

```
     -memory_mb 100 \
     -capacity 2 \
     -admindir /opt/ondb/var/admin \
     -storagedir /tmp/sn1/disk1/ondb/data \
     -storagedir /tmp/sn1/disk2/ondb/data
```

On Win_B

```
java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar makebootconfig \
     -root /opt/ondb/var/kvroot \
     -host <host-ip> \
     -config config2.xml \
     -port 13240 \
     -harange 13242,13245 \
     -memory_mb 100 \
     -capacity 2 \
     -admindir /opt/ondb/var/admin \
     -storagedir /tmp/sn2/disk1/ondb/data \
     -storagedir /tmp/sn2/disk2/ondb/data
```

On Win_C

```
java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar makebootconfig \
     -root /opt/ondb/var/kvroot \
     -host <host-ip> \
     -config config3.xml \
     -port 13250 \
     -harange 13252,13255 \
     -memory_mb 100 \
     -capacity 2    \
     -admindir /opt/ondb/var/admin \
     -storagedir /tmp/sn3/disk1/ondb/data \
     -storagedir /tmp/sn3/disk2/ondb/data
```

This will produce 3 configuration files:

```
/opt/ondb/var/kvroot
     /config1.xml
     /config2.xml
     /config3.xml
```

Using the different configurations just generated, start a corresponding instance of the KVStore Storage Node Agent (SNA) from each window.

> **Note:**
>
> Before starting the SNA, set the environment variable `MALLOC_ARENA_MAX` to `1`. Setting `MALLOC_ARENA_MAX` to `1` ensures that the memory usage is restricted to the specified heap size.

On Win_A

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot -config config1.xml &
```

On Win_B

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot -config config2.xml &
```

On Win_C

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot -config config3.xml &
```

Finally, from any window (Win_A, Win_B, Win_C, or a new window), use the KVStore administrative CLI to configure and deploy the store.

To start the administrative CLI, execute the following command:

```
> java -Xmx64m -Xms64m \
       -jar /opt/ondb/kv/lib/kvstore.jar runadmin \
       -host <host-ip> -port 13230
```

To configure and deploy the store, type the following commands from the administrative CLI prompt (remembering to substitute the actual IP address or hostname for the string <host-ip>):

```
configure -name store-name
plan deploy-zone -name Zone1 -rf 3 -wait
plan deploy-sn -zn 1 -host <host-ip> -port 13230 -wait
plan deploy-admin -sn 1 -port 13231 -wait
pool create -name snpool
pool join -name snpool -sn sn1
plan deploy-sn -zn 1 -host <host-ip> -port 13240 -wait
plan deploy-admin -sn 2 -port 13241 -wait
pool join -name snpool -sn sn2
plan deploy-sn -zn 1 -host <host-ip> -port 13250 -wait
plan deploy-admin -sn 3 -port 13251 -wait
pool join -name snpool -sn sn3
change-policy -params "loggingConfigProps=oracle.kv.level=INFO;"
```

```
change-policy -params cacheSize=10000000
topology create -name store-layout -pool snpool -partitions 100
plan deploy-topology -name store-layout -plan-name RepNode-Deploy -wait
```

> **Note:**
>
> The CLI command prompt (`kv->`) was excluded from the list of commands above to facilitate cutting and pasting the commands into a CLI load script.

When the above commands complete (use `show plans`), the store is up and running and ready for data to be written to it. Before proceeding, verify that a directory like that shown above for the multi-disk scenario has been laid out. That is:

```
   - Win_A -                 - Win_B -                - Win_C -

/opt/ondb/var/kvroot      /opt/ondb/var/kvroot      /opt/ondb/var/
kvroot
  log files                 log files                 log files
  /example-store            /example-store            /example-store
    /log                      /log                      /log
    /sn1                      /sn2                      /sn3
      config.xml                config.xml                config.xml
      /admin1                   /admin2                   /admin3
        /env                      /env                      /env
/tmp/sn1/disk1/ondb/data  /tmp/sn2/disk1/ondb/data /tmp/sn3/disk1/ondb/
data
  /rg1-rn1                  /rg1-rn2                  /rg1-rn3
    /env                      /env                      /env
      00000000.jdb              00000000.jdb              00000000.jdb
```

When a key/value pair is written to the store, it is stored in each of the (rf=3) files named, 00000000.jdb that belong to a given replication group (shard); for example, when a single key/value pair is written to the store, that pair would be stored in either these files:

```
/tmp/sn1/disk2/ondb/data/rg2-rn1/env/00000000.jdb
/tmp/sn2/disk2/ondb/data/rg2-rn2/env/00000000.jdb
/tmp/sn3/disk2/ondb/data/rg2-rn3/env/00000000.jdb
```

Or in these files:

```
/tmp/sn1/disk1/ondb/data/rg1-rn1/env/00000000.jdb
/tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb
/tmp/sn3/disk1/ondb/data/rg1-rn3/env/00000000.jdb
```

At this point, each file should contain no key/value pairs. Data can be written to the store in the most convenient way. But a utility that is quite handy for doing this is the KVStore client shell utility; which is a process that connects to the desired store and then presents a command line interface that takes interactive commands for putting

and getting key/value pairs. To start the KVStore shell, type the following from a window:

```
> java -Xmx64m -Xms64m \
       -jar /opt/ondb/kv/lib/kvstore.jar runadmin\
       -host <host-ip> -port 13230 -store store-name

kv-> get -all
  0 Record returned.

kv-> put -key /FIRST_KEY -value "HELLO WORLD"
  Put OK, inserted.

kv-> get -all
  /FIRST_KEY
  HELLO WORLD
```

A quick way to determine which files the key/value pair was stored in is to simply grep for the string "HELLO WORLD"; which should work with binary files on most linux systems. Using the `grep` command in this way is practical for examples that consist of only a small amount of data.

```
> grep "HELLO WORLD" /tmp/sn1/disk1/ondb/data/rg1-rn1/env/00000000.jdb
> grep "HELLO WORLD" /tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb
> grep "HELLO WORLD" /tmp/sn3/disk1/ondb/data/rg1-rn3/env/00000000.jdb

> grep "HELLO WORLD" /tmp/sn1/disk2/ondb/data/rg2-rn1/env/00000000.jdb
  Binary file /tmp/sn1/disk2/ondb/data/rg2-rn1/env/00000000.jdb matches
> grep "HELLO WORLD" /tmp/sn2/disk2/ondb/data/rg2-rn2/env/00000000.jdb
  Binary file /tmp/sn2/disk2/ondb/data/rg2-rn2/env/00000000.jdb matches
> grep "HELLO WORLD" /tmp/sn3/disk2/ondb/data/rg2-rn3/env/00000000.jdb
  Binary file /tmp/sn3/disk2/ondb/data/rg2-rn3/env/00000000.jdb matches
```

In the example above, the key/value pair that was written to the store was stored by each RN belonging to the second shard; that is, each RN is a member of the replication group with id equal to 2 (rg2-rn1, rg2-rn2, and rg2-rn3).

> **Note:**
>
> With which shard a particular key is associated depends on the key's value (specifically, the hash of the key's value) as well as the number of shards maintained by the store. It is also worth noting that although this example shows log files with the name 00000000.jdb, those files are only the first of possibly many such log files containing data written by the corresponding RN service.

As the current log file reaches its maximum capacity, a new file is created to receive all new data written. That new file's name is derived from the previous file by incrementing the prefix of the previous file. For example, you might see files with names such as, "..., 00000997.jdb, 00000998.jdb, 00000999.jdb, 00001000.jdb,00001001.jdb, ...".

After the data has been written to the store, a failed disk can be simulated, and the disk replacement process can be performed. To simulate a failed disk, pick one of the storage

directories where the key/value pair was written and, from a command window, delete the storage directory. For example:

```
> rm -rf /tmp/sn3/disk2
```

At this point, if the log file for SN3 is examined, you should see repeated exceptions being logged. That is:

```
> tail /opt/ondb/var/kvroot/store-name/log/sn3_0.log

rg2-rn3: ProcessMonitor: java.lang.IllegalStateException: Error
occurred
accessing statistic log file
  /tmp/sn3/disk2/ondb/data/rg2-rn3/env/je.stat.csv.
.......
```

But if the client shell is used to retrieve the previously stored key/value pair, the store is still operational, and the data that was written is still available. That is:

```
kvshell-> get -all
  /FIRST_KEY
  HELLO WORLD
```

The disk replacement process can now be performed. From the command window in which the KVStore administrative CLI is running, execute the following (step 2 from above):

```
kv-> plan stop-service -service rg2-rn3
  Executed plan 9, waiting for completion...
  Plan 9 ended successfully

kv-> verify configuration
  .........
  Rep Node [rg2-rn3] Status: UNREACHABLE
```

If you attempt to restart the RN service that was just stopped, the attempt would not succeed. This can be seen via the contents of SN3's log `file/opt/ondb/var/kvroot/store-name/log/sn3_0.log`. The contents of that file indicate repeated attempts to restart the service, but due to the missing directory – that is, because of the "failed" disk – each attempt to start the service fails, until the process reaches an ERROR state; for example:

```
kv-> show plans
  1 Deploy Zone (1) SUCCEEDED
  ......
  9 Stop RepNodes (9) SUCCEEDED
  10 Start RepNodes (10) ERROR
```

Now, the disk should be "replaced". To simulate disk replacement, we must create the original parent directory of rg2-rn3; which is intended to be analogous to installing and formatting the replacement disk:

```
> mkdir -p /tmp/sn3/disk2/ondb/data
```

From the administrative CLI, attempt to restart the RN service should succeed since the disk has been "replaced".

```
kv-> plan start-service -service rg2-rn3 -wait
  Executed plan 11, waiting for completion...
  Plan 11 ended successfully

kv-> verify configuration
  .........
  Rep Node [rg2-rn3] Status: RUNNING,REPLICA at sequence
  number 327 haPort:13254
```

To verify that the data has been recovered as expected, grep for "HELLO WORLD" again.

```
> grep "HELLO WORLD" /tmp/sn3/disk2/ondb/data/rg2-rn3/env/00000000.jdb
  Binary file /tmp/sn3/disk2/ondb/data/rg2-rn3/env/00000000.jdb matches
```

To see why the disk replacement process outlined above might be more complicated for the default – and by extension, the single disk – case than it is for the multi-disk case, try running the example above using default storage directories; that is, remove the storagedir parameters from the invocation of the `makebootconfig` command above. This will result in a directory structure such as:

```
/opt/ondb/var/kvroot   /opt/ondb/var/kvroot  /opt/ondb/var/kvroot
  log files              log files             log files
  /store-name            /store-name           /store-name
    /log                   /log                  /log
    /sn1                   /sn2                  /sn3
      config.xml             config.xml            config.xml
      /rg1-rn1               /rg1-rn2              /rg1-rn3
      /rg2-rn1               /rg2-rn2              /rg2-rn3
```

In a similar example, to simulate a failed disk in this case, you would delete the directory `/opt/ondb/var/kvroot/sn3`; which is the parent of the `/admin3` database, the `/rg1-rn3` database, and the `/rg2-rn3` database.

It is important to note that the directory also contains the configuration for SN3. Since SN3's configuration information is contained under the same parent – which is analogous to that information stored on the same disk – as the replication node databases; when the "failed" disk is "replaced" as it was in the previous example, the step where the RN service(s) are restarted will fail because SN3's configuration is no longer available. While the replicated data can be automatically recovered from the other nodes in the system when a disk is replaced, the SN's configuration information cannot. That data must be manually restored from a previously backed up copy. This extends to the non-default, single disk case in which different storagedir parameters are used to separate the KVROOT location from the location of each RN database. In that case, even though the replicated data is stored in separate locations,

that data is still stored on the same physical disk. Therefore, if that disk fails, the configuration information is still not available on restart, unless it has been manually reinstalled on the replacement disk.

Servers

Although not as common as a failed disk, it is not unusual for an administrator to need to replace one of the machines hosting services making up a given KVStore deployment (an SN). There are two common scenarios where a whole machine replacement may occur. The first is when one or more hardware components fail and it is more convenient or cost effective to simply replace the whole machine than it is to replace the failed components. The second is when a working, healthy machine is to be upgraded to a machine that is bigger and robust; for example, a machine with larger disks and better performance characteristics. The procedures presented in this section are intended to describe the steps for preparing a new machine to replace an existing machine, and the steps for retiring the existing machine.

Detecting and Correlating Server Failures to NoSQL Log Events

In a distributed such as Oracle NoSQL Database, it is generally difficult to distinguish between network outages and machine failure. The HA components of the NoSQL Database detects when a replication node is unreachable and logs this as an event in the admin log - however grepping for this log event produces false positives. Therefore it is recommended to utilize a systems monitoring package like JMX to detect machine/server failure.

> **Note:**
>
> If the log files are compressed, you can search the compressed log files using the `zgrep` command.

Resolving Server Failures

Two replacement procedures are presented below. Both procedures essentially achieve the same results, and both will result in one or more network restore processes being performed (see below).

The first procedure presented replaces the old machine with a machine that – to all interested parties – looks exactly like the original machine. That is, the new machine has the same hostname, IP address, port, and SN id. Compare this with the second procedure; where the old machine is removed from the store's topology and replaced with a machine that appears to be a different machine - different hostname, IP address, SN id – but the behavior is identical to the behavior of the replaced machine. That is, the new machine runs the same services, and manages the exact same data, as the original machine; it just happens to have a different network and SN identity. Thus, the first case can be viewed as a replacement of only the hardware; that is, from the point of view of the store, the original SN was temporarily taken down and then restarted. The new hardware is not reflected in the store's topology. In the other case, the original SN is removed, and a different SN takes over the original's duties. Although the store's content and behavior hasn't changed, the change in hardware is reflected in the store's new topology.

When determining which procedure to use when replacing a Storage Node, the decision is left to the discretion of the store administrator. Some administrators prefer to always use only one of the procedures, never the other. And some administrators establish a policy that is based on some preferred criteria. For example, you might imagine a policy where the first procedure is employed whenever SN replacement must be performed because the hardware has failed; whereas the second procedure is employed whenever healthy hardware is to be upgraded with newer/better hardware. In the first case, the failed SN is down and unavailable during the replacement process. In the second case, the machine to be replaced can remain up and available while the new machine is being prepared for migration; after which the old machine can be shut down and removed from the topology.

Terminology Review

It may be useful to review some of the terminology introduced in the Oracle NoSQL Database Getting Started Guide as well as the Oracle NoSQL Database Administrator's Guide. Recall from those documents that the physical machine on which the processes of the KVStore run is referred to as a Storage Node, or SN; where a typical KVStore deployment generally consists of a number of machines – that is, a number of SNs – that execute the processes and software services provided by the Oracle NoSQL Database KVStore. Recall also that when the KVStore software is initially started on a given SN machine, a process referred to as the "Storage Node Agent" (or the SNA) is started. Then, once the SNA is started, the KVStore administrative CLI is used to configure the store and deploy a "topology"; which results in the SNA executing and managing the lifecycle of one or more "services" referred to as "replication nodes" (or RN services). Finally, in addition to starting and managing RN services, the SNA also optionally (depending on the configuration) starts and manages another service type referred to as the "admin" service.

Because of the 1-to-1 correspondence between the machines making up a given KVStore deployment and the SNA process initially started on each machine when installing and deploying a store, the terms "Storage Node", "SN", or "SNx" (where x is a positive integer) are often used interchangeably in the Oracle NoSQL Database documents – including this note – when referring to either the machine on which the SNA is running, or the SNA process itself. Which concept is intended should be clear from the context in which the term is used in a given discussion. For example, when the terms SN3 or sn3 are used below as part of a discussion about hardware issues such as machine failure and recovery, that term refers to the physical host machine running an SNA process that has been assigned the id value 3 and is identified in the store's topology with the string "sn3". In other contexts, for example when the behavior of the store's software is being discussed, the term SN3 and/or sn3 would refer to the actual SNA process running on the associated machine.

Although not directly pertinent to the discussion below, the following terms are presented not only for completeness, but also because it may be useful to understand their implications when trying to determine which SN replacement procedure to employ.

First, recall from the Oracle NoSQL Database documents that the RN service(s) that are started and managed by each SNA are represented in the store's topology by their service identification number (a positive integer), in conjunction with the identification number of the replication group – or "shard" – in which the service is a member. For example, a given store's topology may reference a particular RN service with the string, "rg3-rn2"; which represents the RN service having id equal to 2 that is a member of the replication group (that is, the shard) with id 3. The capacity then, of a given SN machine that is operating as part of a given KVStore cluster is the number of RN services that will be started and managed by the SNA process deployed to that SN host. Thus, if the capacity of a given SN is 1, only a single RN service will be started and managed by that SN. On the other hand, if the capacity is 3 (for example), then 3 RN services will be started and managed by that SN, and each RN will typically belong to a different replication group (share).

With respect to the SN host machines and resident SNA processes that are deployed to a given KVStore, two concepts to understand are the concept of a "zone", and the concept of a "pool" of Storage Nodes. Both concepts correspond to mechanisms that are used to organize the SNs of the store into groups. As a result, the distinction between the two concepts is presented below.

When configuring a KVStore for deployment, it is a requirement that at least one "zone" be deployed to the store before deploying any Storage Nodes. Then, when deploying each SNA process, in addition to specifying the desired host, one of the previously deployed zones must also be specified; which, with respect to the store's topology, will "contain" that SNA, as well as the services managed by that SNA. Thus, the KVStore deployment process produces a a store consisting of one or more zones, where a distinct set of storage nodes belongs to (is a member of) one – and only one – of those zones.

In contrast to a zone, rather than being "deployed" to the store, one or more Storage Node "pools" can be (optionally) "created" within the store. Once such a "pool" is created, any deployed Storage Node can then be configured to "join" that pool, as well as any other pool that has been created. This means that, unlike zones, where the store consists of one or more zones containing disjoint sets of the deployed SNs, the store can also consist of one or more "pools", where there is no restriction on which, or how many, pools a given SN joins. Every store is automatically configured with a default pool named, "AllStorageNodes"; which all deployed Storage Nodes join. The creation of any additional pools is optional, and left to the discretion of the deployer; as is the decision about which pools a given Storage Node joins.

Besides the differences described above, there are additional conceptual differences to understand when using zones and pools to group sets of Storage Nodes. Although zones can be used to represent logical groupings of a store's nodes, crossing physical boundaries, deployers generally map them to real, physical locations. For example, although there is nothing to prevent the deployment of multiple SNA processes to a single machine, where each SNA is deployed to a different zone, more likely than not, a single SNA will be deployed to a single machine, and the store's zones along with the SN machines within each zone will generally be defined to correspond to physical locations that provide some form of fault isolation. For example, each zone may be defined to correspond to a separate floor of a building; or to separate buildings, a few miles apart (or even across the country).

Compare how zones are used with how pools are generally used. A single pool may represent all of the Storage Nodes across all zones; where the default pool is one such pool. On the other hand, multiple pools may be specified; in some cases with no relation between the pools and zones, and in other cases with each pool corresponding to a zone and containing only the nodes belonging to that zone. Although there may be reasons to map a set of Storage Node pools directly to the store's zones, this is not the primary intent of pools. Whereas the intent of zones is to enable better fault isolation and geographic availability via physical location of the storage nodes, the primary purpose of a pool is to provide a convenient mechanism for referring to a group of storage nodes when applying a given administrative operation. That is, the administrative store operations that take a pool argument can be called once to apply the desired operation to all Storage Nodes belonging to the specified pool, avoiding the need to execute the operation multiple times; once for each Storage Node.

Associated with zones, another term to understand is "replication factor" (or "rf"). Whenever a zone is deployed to a KVStore, the "replication factor" of that zone must be specified; which represents the number of copies (or "replicas") of each key/value pair to write and maintain on the nodes of the associated zone. Note that whereas

"capacity" is a per/SN concept that specifies the number of RN services to manage on a given machine, the "replication factor" is a concept whose scope is per/zone, and is used to determine the number of RN services that belong to each shard (or "replication group") created and managed within the associated zone.

Finally, a "network restore" is a process whereby the store automatically recovers all data previously written by a given RN service; retrieving replicas of the data from one or more RN services running on different SNs and then transferring that data (across the network) to the RN whose database is being restored. It is important to understand the implications this process may have on system performance; as the process can be quite time consuming, and can add significant network traffic and load while the data store of the restored RN is being repopulated. Additionally, with respect to SN replacement, these implications can be magnified when the capacity of the SN to be replaced is greater than 1; as this will result in multiple network restorations being performed.

Assumptions

When presenting the two procedures below, for simplicity, assume that a KVStore is initially deployed to 3 machines, resulting in a cluster of 3 Storage Nodes; sn1, sn2, sn3 on hosts with names, host-sn1, host-sn2, and host-sn3 respectively. Assume that:

- Each machine has a disk named `/opt` and a disk named `/disk1`; where each SN will store its configuration and admin database under `/opt/ondb/var/kvroot`, but will store the data that is written on the other, separate disk under `/disk1/ondb/data`.
- The KVStore is installed on each machine under `/opt/ondb/kv`; that is,`KVHOME=/opt/ondb/kv`.
- The KVStore is deployed with `KVROOT=/opt/ondb/var/kvroot`.
- The KVStore is named "example-store".
- One zone – named "Zone1" and configured with `rf=3` – is deployed to the store.
- Each SN is configured with `capacity=1`.
- After deploying each SN to the zone named "Zone1", each SN joins the `pool` named "snpool".
- In addition to the SNA and RN services, an admin service is also deployed to each machine; that is, `admin1` is deployed to `host-sn1` , `admin2` is deployed to `host-sn2`, and `admin3` is deployed to `host-sn3`, each listening on port 13230.

Using specific values such as those reflected in the Assumptions described above enables to follow the steps of each procedure. Using this administrators can generalize and extend those steps to their own particular deployment scenario, substituting the values specific to the given environment where necessary.

Replacement Procedure 1: Replace SN with Identical SN

The procedure presented in this section describes how to replace the desired SN with a machine having an identical network and SN identity. A number of requirements must be satisfied before executing this procedure; which are:

- An admin service must be running and accessible somewhere in the system.
- The `id` of the SN to be replaced must be known.
- The SN to be replaced must be taken down – either administratively or via failure – before starting the new SN.

An admin service is necessary so that the current configuration of the SN to be replaced can be retrieved from the admin service's database and packaged for installation on the new SN. Thus, before proceeding, the administrator must know the location (hostname or IP address) of the admin service, along with the port on which that service is listening for requests. Additionally, since this process requires the id of the SN to be replaced, the administrator must also know that value before initiating the procedure below; for example, something like, sn1, sn2, sn3, etc.

Finally, if the SN to be replaced has failed, and is down, the last requirement above is automatically satisfied. On the other hand, if the SN to be replaced is up, then at some point before starting the new SN, the old SN must be down so that that SN and the replacement SN do not conflict.

With respect to the requirement related to the admin service, if the system is running multiple instances of the admin, it is not important which instance is used in the steps below; just that the admin is currently running and accessible. This means that if the SN to be replaced is not only up but is also running an admin service, then that admin service can be used to retrieve and package that SN's current configuration. But if that SN has failed or is down or inaccessible for some reason, then any admin service on that SN is also down and/or inaccessible - which means an admin service running on one of the other SNs in the system must be used in the procedure below. This is why the Oracle NoSQL Database documents strongly encourage administrators to deploy multiple admin services; where the number deployed should make quorum loss less likely.

For example, it is obvious that if only 1 admin service was specified when deploying the store, and that service was deployed to the SN to be replaced, and that SN has failed or is otherwise inaccessible, then the loss of that single admin service makes it very difficult to replace the failed SN using the procedure presented here. Even if multiple admins are deployed – for example, 2 admins – and the failure of the SN causes just one of those admins to also fail and thus lose quorum, even though a working admin remains, it will still require additional work to recover quorum so that the admin service can perform the necessary duties to replace the failed SN.

Suppose a KVStore has been deployed as described in the section Assumptions. Also, suppose that the sn2 machine (whose hostname is, "host-sn2") has failed in some way and needs to be replaced. If the administrator wishes to replace the failed SN with an identical but healthy machine, then the administrator would do the following:

1. If, for some reason, host-sn2 is running, shut it down.
2. Log into host-sn1 (or host-sn3).
3. From the command line, execute the `generateconfig` utility to produce a ZIP file named "sn2-config.zip" that contains the current configuration of the failed SN (sn2):

   ```
   > java -Xmx64m -Xms64m \
          -jar /opt/ondb/kv/lib/kvstore.jar generateconfig \
          -host host-sn1 -port 13230 \
          -sn sn2 -target /tmp/sn2-config
   ```

   which creates and populates the file, `/tmp/sn2-config.zip`.
4. Install and provision a new machine with the same network configuration as the machine to be replaced; specifically, the same hostname and IP address.

5. Install the KVStore software on the new machine under `KVHOME=/opt/ondb/kv`.
6. If the directory `KVROOT=/opt/ondb/var/kvroot` exists, then make sure it's empty; otherwise, create it:

```
> rm -rf /opt/ondb/var/kvroot
> mkdir -p /opt/ondb/var/kvroot
```

7. Copy the ZIP file from `host-sn1` to the new `host-sn`.

```
> scp /tmp/sn2-config.zip host-sn2:/tmp
```

8. On the new host-sn2, install the contents of the ZIP file just copied.

```
> unzip /tmp/sn2-config.zip -d /opt/ondb/var/kvroot
```

9. Restart the sn2 Storage Node on the new host-sn2 machine, using the old sn2 configuration that was just installed:

> **Note:**
>
> Before starting the SNA, set the environment variable `MALLOC_ARENA_MAX` to `1`. Setting `MALLOC_ARENA_MAX` to `1` ensures that the memory usage is restricted to the specified heap size.

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config.xml&
```

which, after starting the SNA, RN, and admin services, will initiate a (possibly time-consuming) network restore, to repopulate the data stores managed by this new sn2.

Replacement Procedure 2: New SN Takes Over Duties of Removed SN

The procedure presented in this section describes how to deploy a new SN, having a network and SN identity different than all current SNs in the store, that will effectively replace one of the current SNs by taking over that SN's duties and data. Unlike the previous procedure, the only prerequisite that must be satisfied when executing this second procedure is the existence of a working quorum of admin service(s). Also, whereas in the previous procedure the SN to be replaced must be down prior to powering up the replacement SN (because the two SNs share an identity), in this case, the SN to be replaced can remain up and running until the migration step of the process; where the replacement SN finally takes over the duties of the SN being replaced. Thus, although the SN to be replaced can be down throughout the whole procedure if desired, that SN can also be left up so that it can continue to service requests while the replacement SN is being prepared.

Suppose a KVStore has been deployed as described in the section Assumptions. Also, suppose that the sn2 machine is currently up, but needs to be upgraded to a new machine with more memory, larger disks, and better overall performance characteristics. The administrator would then do the following:

1. From a machine with the Oracle NoSQL Database software installed that has network access to one of the machines running an admin service for the deployed KVStore, start the administrative CLI; connecting it to that admin service. The machine on which the CLI is run can be any of the machines making up the store – even the machine to be replaced – or a separate client machine. For example, if the administrative CLI is started on the `sn1` Storage Node, and one wishes to connect that CLI to the admin service running on that same `sn1` host, the following would be typed from a command prompt on the host named, `host-sn1`:

   ```
   > java -Xmx64m -Xms64m \
   -jar /opt/ondb/kv/lib/kvstore.jar runadmin \
          -host host-sn1 -port 13230
   ```

2. From the administrative CLI just started, execute the `show pools` command to determine the Storage Node `pool` the new Storage Node will need to join after deployment; for example,

   ```
   kv-> show pools
   ```

   which, given the initial assumptions, should produce output that looks like the following:

   ```
   AllStorageNodes: sn1 sn2 sn3
   snpool: sn1 sn2 sn3
   ```

   where, from this output, one should note that the name of the pool the new Storage Node should join below is "snpool"; and the pool named "AllStorageNodes" is the pool that all Storage Nodes join by default when deployed.

3. From the administrative CLI just started, execute the `show topology` command to determine the zone to use when deploying the new Storage Node; for example,

   ```
   kv-> show topology
   ```

   which, should produce output that looks like the following:

   ```
   store=example-store numPartitions=300 sequence=308
     zn: id=1 name=Zone1 repFactor=3

     sn=[sn1] zn:[id=1 name=Zone1] host-sn1 capacity=1 RUNNING
       [rg1-rn1] RUNNING

     sn=[sn2] zn:[id=1 name=Zone1] host-sn2 capacity=1 RUNNING
       [rg1-rn2] RUNNING

     sn=[sn3] zn:[id=1 name=Zone1] host-sn3 capacity=1 RUNNING
       [rg1-rn3] RUNNING
     ........
   ```

   where, from this output, one should then note that the id of the zone to use when deploying the new Storage Node is "1".

4. Install and provision a new machine with a network configuration that is different than each of the machines currently known to the deployed KVStore. For example, provision the new machine with a hostname such as, host-sn4, and an IP address unique to the store's members.
5. Install the KVStore software on the new machine under `KVHOME=/opt/ondb/kv`.
6. Create the new Storage Node's KVROOT directory; for example:

```
> mkdir -p /opt/ondb/var/kvroot
```

7. Create the new Storage Node's data directory on a separate disk than KVROOT; for example:

```
> mkdir -p /disk1/ondb/data
```

> **Note:**
>
> The path used for the data directory of the replacement SN must be identical to the path used by the SN to be replaced.

8. From the command prompt of the new host-sn4 machine, use the `makebootconfig` utility (described in Chapter 2 of the Oracle NoSQL Database Administrator's Guide, section, "Installation Configuration") to create an initial configuration for the new Storage Node that is consistent with the Assumptions specified above; for example:

```
> java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar makebootconfig \
       -root /opt/ondb/var/kvroot \
       -port 13230  \
       -host host-sn4
       -harange 13232,13235 \
       -num_cpus 0  \
       -memory_mb 0 \
       -capacity 1  \
       -admindir /opt/ondb/var/admin \
       -admindirsize 3_gb \
       -storagedir /disk1/ondb/data \
       -rnlogdir /disk1/ondb/rnlog
```

which produces the file named config.xml, under `KVROOT=/opt/ondb/var/kvroot`.

9. Using the configuration just created, start the KVStore software (the SNA and its managed services) on the new host-sn4 machine; for example,

> **Note:**
>
> Before starting the SNA, set the environment variable `MALLOC_ARENA_MAX` to `1`. Setting `MALLOC_ARENA_MAX` to `1` ensures that the memory usage is restricted to the specified heap size.

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config.xml &
```

10. Using the information associated with the sn2 Storage Node (the SN to replace) that was gathered from the `show topology` and `show pools` commands above, use the administrative CLI to deploy the new Storage Node and join the desired pool; that is,

    ```
    kv-> plan deploy-sn -znname Zone1 -host host-sn4 -port 13230 -wait
    kv-> pool join -name snpool -sn sn4
    ```

    For an SN to join a pool, the SN must have been successfully deployed and the id of the deployed SN must be specified in the `pool join` command; for example, "sn4" above. But upon examination of the `plan deploy-sn`, command you can see that the id to assign to the SN being deployed is not specified. This is because it is the KVStore itself – not the administrator – that determines the id to assign to a newly deployed SN. Thus, given that it was assumed that only 3 Storage Nodes were initially deployed in the example used to demonstrate this procedure, when deploying the next Storage Node, the system will increment by 1 the integer component of the id assigned to the most recently deployed SN – "sn3" or 3 in this case – and use the result to construct the id to assign to the next SN that is deployed. Hence, "sn4" was assumed to be the id to specify to the `pool join` command above. But if you want to ascertain the assigned id, then before joining the pool, execute the `show topology` command which will display the id that was constructed and assigned to the newly deployed SN.

11. Since the old SN must not be running when the migrate operation is performed (see the next step), if the SN to be replaced is still running at this point, programmatically shut it down, and then power off and disconnect the associated machine. This step can be performed at any point prior to performing the next step. Thus, to shut down the SN to be replaced, type the following from the command prompt of the machine hosting that SN:

    ```
    > java -Xmx64m -Xms64m \
           -jar /opt/ondb/kv/lib/kvstore.jar stop \
           -root /opt/ondb/var/kvroot
    ```

    On completion, the associated machine can then be powered down and disconnected if desired.

12. After the new Storage Node has been deployed, joined the desired pool, and the SN to be replaced is no longer running, use the administrative CLI to migrate that old SN to the new SN. This means, in this case, that the SNA, and RN associated

with sn4 will take over the duties previously performed in the store by the corresponding services associated with sn2; and the data previously stored by sn2 will be moved – via the network – to the storage directory for sn4. To perform this step then, execute the following command from the CLI:

```
kv-> plan migrate-sn -from sn2 -to sn4 [-wait]
```

The `-wait` argument is optional in the command above. If `-wait` is used, then the command will not return until the full migration has completed; which, depending on the amount of data being migrated, can take a long time. If `-wait` is not specified, then the `show plan -id <migration-plan-id>` command is used to track the progress of the migration; allowing other administrative tasks to be performed during the migration.

13. After the migration process completes, remove the old SN from the store's topology. You can do this by executing the `plan remove-sn` command from the administrative CLI. For example,

```
kv-> plan remove-sn -sn sn2 -wait
```

At this point, the store should have a structure similar to its original structure; except that the data that was originally stored by sn2 on the host named host-sn2 via that node's rg1-rn2 service, is now stored on host-sn4 by the sn4 Storage Node (via the migrated service named rg1-rn2 that sn4 now manages).

Examples

In this section, two examples are presented that should allow you to gain some practical experience with the SN replacement procedures presented above. Each example uses the same initial configuration, and is intended to simulate a 3-node KVStore cluster using a single machine with a single disk. Although no machines will actually fail or be physically replaced, you should still get a feel for how the cluster and the data stored by a given SN is automatically recovered when that Storage Node is replaced using one of the procedures described above.

Assume that a KVStore is deployed in a manner similar to the section Assumptions Specifically, assume that a KVStore is initially deployed using 3 Storage Nodes - named sn1, sn2, and sn3 – on a single host with IP address represented by the string, <host-ip> where the host's actual IP address (or hostname) is substituted for <host-ip> when running either example. Additionally, since your development system will typically not contain a disk named /disk1 (as specified in the Assumptions section), rather than provisioning such a disk, assume instead that the data written to the store will be stored under `/tmp/sn1/disk1`, `/tmp/sn2/disk1`, and `/tmp/sn3/disk1` respectively. Finally, since each Storage Node runs on the same host, assume each Storage Node is configured with different ports for the services and admins run by those nodes; otherwise, all other assumptions are as stated above in the Assumptions section.

Setup

As indicated above, the initial configuration and setup is the same for each example presented below. Thus, if not done so already, first create the `KVROOT` directory; that is,

```
> mkdir -p /opt/ondb/var/kvroot
```

Then, to simulate the data disk, create the following directories:

```
> mkdir -p /tmp/sn1/disk1/ondb/data
> mkdir -p /tmp/sn2/disk1/ondb/data
> mkdir -p /tmp/sn3/disk1/ondb/data
```

Next, open 3 windows; Win_A, Win_B, and Win_C, which will represent the 3 machines running each Storage Node. In each window, execute the `makebootconfig` command (remembering to substitute the actual IP address or hostname for the string <host-ip>) to create a different, but similar, boot config for each SN that will be configured.

On Win_A

```
java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar makebootconfig \
     -root /opt/ondb/var/kvroot \
     -host <host-ip> \
     -config config1.xml \
     -port 13230 \
     -harange 13232,13235 \
     -memory_mb 100 \
     -capacity 1 \
     -admindir /opt/ondb/var/admin \
     -admindirsize 2000-Mb \
     -storagedir /tmp/sn1/disk1/ondb/data \
     -rnlogdir /tmp/sn1/disk1/ondb/rnlog
```

On Win_B

```
java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar makebootconfig \
     -root /opt/ondb/var/kvroot \
     -host <host-ip> \
     -config config2.xml \
     -port 13240 \
     -harange 13242,13245 \
     -memory_mb 100 \
     -capacity 1 \
     -admindir /opt/ondb/var/admin \
     -admindirsize 2000-Mb \
     -storagedir /tmp/sn1/disk2/ondb/data \
     -rnlogdir /tmp/sn1/disk2/ondb/rnlog
```

On Win_C

```
java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar makebootconfig \
     -root /opt/ondb/var/kvroot \
     -host <host-ip> \
     -config config3.xml \
     -port 13250 \
     -harange 13252,13255 \
```

```
    -memory_mb 100 \
    -capacity 1 \
    -admindir /opt/ondb/var/admin \
    -admindirsize 2000-Mb \
    -storagedir /tmp/sn1/disk3/ondb/data \
    -rnlogdir /tmp/sn1/disk3/ondb/rnlog
```

This will produce 3 configuration files:

```
/opt/ondb/var/kvroot
    /config1.xml
    /config2.xml
    /config3.xml
```

Next, using the different configurations just generated, from each window, start a corresponding instance of the KVStore Storage Node agent (SNA); which, based on the specific configurations generated, will start and manage an admin service and an RN service.

> **Note:**
>
> Before starting the SNA, set the environment variable `MALLOC_ARENA_MAX` to `1`. Setting `MALLOC_ARENA_MAX` to `1` ensures that the memory usage is restricted to the specified heap size.

Win_A

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config1.xml &
```

Win_B

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config2.xml &
```

Win_C

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config3.xml &
```

Finally, from any window (Win_A, Win_B, Win_C, or a new window), start the KVStore administrative CLI and use it to configure and deploy the store. For example, to start an

administrative CLI connected to the admin service that was started above using the configuration employed in Win_A, you would execute the following command:

```
> java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar runadmin \
       -host <host-ip> -port 13230
```

To configure and deploy the store, type the following commands from the administrative CLI prompt (remembering to substitute the actual IP address or hostname for the string <host-ip>):

```
configure -name example-store
plan deploy-zone -name Zone1 -rf 3 -wait
plan deploy-sn -znname Zone1 -host <host-ip> -port 13230 -wait
plan deploy-admin -sn 1 -port 13231 -wait
pool create -name snpool
pool join -name snpool -sn sn1
plan deploy-sn -znname Zone1 -host <host-ip> -port 13240 -wait
plan deploy-admin -sn 2 -port 13241 -wait
pool join -name snpool -sn sn2
plan deploy-sn -znname Zone1 -host <host-ip> -port 13250 -wait
plan deploy-admin -sn 3 -port 13251 -wait
pool join -name snpool -sn sn3
change-policy -params "loggingConfigProps=oracle.kv.level=INFO;"
change-policy -params cacheSize=10000000
topology create -name store-layout -pool snpool -partitions 300
plan deploy-topology -name store-layout -plan-name RepNode-Deploy -wait
```

> **Note:**
>
> The CLI command prompt (`kv->`) was excluded from the list of commands above to facilitate cutting and pasting the commands into a CLI load script.

When the commands above complete (use `show plans` to verify each plan's completion), the store is up and running and ready for data to be written to it. Before proceeding though, verify that directories like those shown below have been created and populated:

```
     - Win_A -                  - Win_B -                - Win_C -

  /opt/ondb/var/             /opt/ondb/var/             /opt/ondb/var/
      admin                      admin                      admin
/opt/ondb/var/kvroot      /opt/ondb/var/kvroot      /opt/ondb/var/
kvroot
  log files                 log files                 log files
  /example-store            /example-store            /example-store
    /log                      /log                      /log
    /sn1                      /sn2                      /sn3
      config.xml                config.xml                config.xml
      /admin1                   /admin2                   /admin3
        /env                      /env                      /env
```

```
/tmp/sn1/disk1/ondb/data  /tmp/sn2/disk1/ondb/data /tmp/sn3/disk1/ondb/data
  /rg1-rn1                  /rg1-rn2                  /rg1-rn3
    /env                      /env                      /env
      00000000.jdb              00000000.jdb              00000000.jdb
```

Because rf=3 for the deployed store, and capacity=1 for each SN in that store, when a key/value pair is initially written to the store, the pair is stored by each of the replication nodes – rn1, rn2, and rn3 – in their corresponding data file named "00000000.jdb"; where each replication node is a member of the replication group – or shard – named rg1; that is, the key/value pair is stored in:

```
/tmp/sn1/disk1/ondb/data/rg1-rn1/env/00000000.jdb
/tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb
/tmp/sn3/disk1/ondb/data/rg1-rn3/env/00000000.jdb
```

At this point in the setup, each file should contain no key/value pairs. Data can be written to the store in a way most convenient. But a utility that is quite handy for doing this is the KVStore client shell utility; which is a process that connects to the desired store and then presents a command line interface that takes interactive commands for putting and getting key/value pairs. To start the KVStore client shell, type the following from a command window (remembering to substitute the actual IP address or hostname for the string <host-ip>):

```
> java -Xmx64m -Xms64m \
       -jar /opt/ondb/kv/lib/kvstore.jar runadmin\
       -host <host-ip> -port 13230 -store example-store

kv-> get -all
  0 Record returned.

kv-> put -key /FIRST_KEY -value "HELLO WORLD"
  Put OK, inserted.

kv-> get -all
  /FIRST_KEY
  HELLO WORLD
```

Although simplistic and not very programmatic, a quick way to verify that the key/value pair was stored by each RN service is to simply grep for the string "HELLO WORLD" in each of the data files; which should work with binary files on most linux systems. Using the "grep" command in this way is practical for examples that consist of only a small amount of data.

```
> grep "HELLO WORLD" /tmp/sn1/disk1/ondb/data/rg1-rn1/env/00000000.jdb
  Binary file /tmp/sn1/disk1/ondb/data/rg1-rn1/env/00000000.jdb matches
> grep "HELLO WORLD" /tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb
  Binary file /tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb matches
> grep "HELLO WORLD" /tmp/sn3/disk1/ondb/data/rg1-rn3/env/00000000.jdb
  Binary file /tmp/sn3/disk1/ondb/data/rg1-rn3/env/00000000.jdb matches
```

Based on the output above, the key/value pair that was written to the store was stored by each RN service belonging to the shard rg1; that is, each RN service that is a member of the replication group with id equal to 1 (rg1-rn1, rg1-rn2, and rg1-rn3). With which shard a particular key is associated depends on the key's value (specifically, the hash of the key's

value) as well as the number of shards maintained by the store (1 in this case). It is also worth noting that although this example shows log files with the name 00000000.jdb, those files are only the first of possibly many such log files containing data written by the corresponding RN service. Over time, as the current log file reaches its maximum capacity, a new file will be created to receive all new data being written. That new file has a name derived from the previous file by incrementing the prefix of the previous file. For example, you might see files with names such as, "..., 00000997.jdb, 00000998.jdb, 00000999.jdb, 00001000.jdb, 00001001.jdb, ...".

Now that data has been written to the store, a failed storage node can be simulated, and an example of the first SN replacement procedure can be performed.

Example 1: Replace a Failed SN with an Identical SN

To simulate a failed Storage Node, pick one of the Storage Nodes started above, programmatically stop it's associated processes, and delete all files and directories associated with that process. For example, suppose sn2 is the "failed" Storage Node. But before stopping the sn2 Storage Node, you might first (optionally) identify the processes that are running as part of the deployed store; that is:

```
> jps -m
408 kvstore.jar start -root /opt/ondb/var/kvroot -config config1.xml
833 ManagedService -root /opt/ondb/var/kvroot -class Admin -service
BootstrapAdmin.13230 -config config1.xml
1300 ManagedService -root /opt/ondb/var/kvroot/example-store/sn1 -store
example-store -class RepNode -service rg1-rn1
....
563 kvstore.jar start -root /opt/ondb/var/kvroot -config config2.xml
1121 ManagedService -root /opt/ondb/var/kvroot/example-store/sn2
-store example-store -class Admin -service admin2
1362 ManagedService -root /opt/ondb/var/kvroot/example-store/sn2
-store example-store -class RepNode -service rg1-rn2
....
718 kvstore.jar start -root /opt/ondb/var/kvroot -config config3.xml
1232 ManagedService -root /opt/ondb/var/kvroot/example-store/sn3 -store
example-store -class Admin -service admin3
1431 ManagedService -root /opt/ondb/var/kvroot/example-store/sn3 -store
example-store -class RepNode -service rg1-rn3
....
```

The output above was manually re-ordered for readability. In reality, each process listed may appear in a random order. But it should be noted that each SN from the example deployment corresponds to 3 processes:

- The SNA process, which is characterized by the string "kvstore.jar start", and identified by the corresponding configuration file; for example, `config1.xml` for `sn1`, `config2.xml` for `sn2`, and `config3.xml` for `sn3`.
- An admin service is characterized by the string `-class Admin` , and either a string of the form `-service BootstrapAdmin.<port>` or a string of the form `-service admin<id>` (see the explanation below).
- An RN service characterized by the string `-class RepNode` along with a string of the form `-service rg1-rn<id>`; where "<id>" is 1, 2, etc. and maps to the SN hosting the given RN service, and where for a given SN, if the `capacity` of that SN

is N>1, then for that SN, there will be N processes listed that reference a different `RepNode` service.

> **Note:**
>
> With respect to the line in the process list above that references the string `-service BootstrapAdmin.<port>`, some explanation may be useful. When an SNA starts up and the `-admin` argument is specified in the configuration, the SNA will initially start what is referred to as a bootstrap admin. Because this example specified the `-admin` argument in the configuration of all 3 Storage Nodes, each SNA in the example starts a corresponding bootstrap admin. The fact that the process list above contains only one entry referencing a `BootstrapAdmin` is explained below.

Recall that Oracle NoSQL Database requires the deployment of at least 1 admin service. If more than 1 such admin is deployed, the admin that is deployed first takes on a special role within the KVStore. In this example, any of the 3 bootstrap admins that were started by the corresponding Storage Node Agent can be that first deployed admin service. After configuring the store and deploying the zone, the deployer must choose one of the Storage Nodes that was started and use the plan `deploy-sn` command to deploy that Storage Node to the desired zone within the store. After deploying that first Storage Node, the admin service corresponding to that Storage Node must then be deployed, using the `plan deploy-admin` command.

Until that first admin service is deployed, no other storage nodes or admins can be deployed. When that first admin service is deployed to the machine running the first SN (sn1 in this case), the bootstrap admin running on that machine continues running, and takes on the role of the very first admin service in the store. This is why the `BootstrapAdmin.<port>` process continues to appear in the process list; whereas, as explained below, the processes associated with the other Storage Nodes are identified by admin2 and admin3 rather than `BootstrapAdmin.<port>`. It is only after this first admin is deployed that the other Storage Nodes (and admins) can be deployed.

Upon deployment of any of the other Storage Nodes, the BootstrapAdmin process associated with each such Storage Node is shut down and removed from the RMI registry. This is because there is no longer a need for the bootstrap admin on these additional Storage Nodes. The existence of a bootstrap admin is an indication that the associated Storage Node Agent can host the first admin if desired. But once the first Storage Node is deployed and its corresponding bootstrap admin takes on the role of the first admin, the other Storage Nodes can no longer host that first admin; and so, upon deployment of each additional Storage Node, the corresponding `BootstrapAdmin` process is stopped. Additionally, if that first process referencing the `BootstrapAdmin` is stopped and restarted at some point after the store has been deployed, then the new process will be identified in the process list with the string `-class Admin`, just like the other admin processes.

Finally, recall that although a store can be deployed with only 1 admin service, it is strongly recommended that multiple admin services be run for greater availability; where the number of admins deployed should be large enough that quorum loss is unlikely in the event of failure of an SN. Thus, as this example demonstrates, after each additional Storage Node is deployed (and the corresponding bootstrap admin is stopped), a new admin service should then be deployed that will coordinate with the first admin service to replicate the administrative information that is persisted. Hence, the admin service associated with sn1 in

the process list above is identified as a BootstrapAdmin (the first admin service), and the other admin services are identified as simply admin2 and admin3.

Thus, to simulate a "failed" Storage Node, sn2 should be stopped; which is accomplished by typing the following at the command prompt:

```
> java -Xmx64m -Xms64m \
       -jar /opt/ondb/kv/lib/kvstore.jar stop \
       -root /opt/ondb/var/kvroot \
       -config config2.xml
```

Optionally, use the `jps` command to examine the processes that remain; that is,

```
> jps -m

408 kvstore.jar start -root /opt/ondb/var/kvroot
-config config1.xml
833 ManagedService -root /opt/ondb/var/kvroot
-class Admin -service BootstrapAdmin.13230 -config config1.xml
1300 ManagedService -root /opt/ondb/var/kvroot/
example-store/sn1 -store example-store -class RepNode -service rg1-rn1
....
718 kvstore.jar start -root /opt/ondb/var/
kvroot -config config3.xml
1232 ManagedService -root /opt/ondb/var/kvroot/example-store/
sn3 -store example-store -class Admin -service admin3
1431 ManagedService -root /opt/ondb/var/kvroot/example-store/
sn3 -store example-store -class RepNode -service rg1-rn3
....
```

where the processes previously associated with sn2 are no longer running. Next, since the sn2 processes have stopped, the associated files can be deleted as follows:

```
> rm -rf /tmp/sn2/disk1/ondb/data/rg1-rn2
> rm -rf /opt/ondb/var/kvroot/example-store/sn2

> rm -f /opt/ondb/var/kvroot/config2.xml
> rm -f /opt/ondb/var/kvroot/config2.xml.log
> rm -f /opt/ondb/var/kvroot/snaboot_0.log.1*

> rm -r /opt/ondb/var/kvroot/example-store/log/admin2*
> rm -r /opt/ondb/var/kvroot/example-store/log/rg1-rn2*
> rm -r /opt/ondb/var/kvroot/example-store/log/sn2*
> rm -r /opt/ondb/var/kvroot/example-store/log/config.rg1-rn2
> rm -r /opt/ondb/var/kvroot/example-store/log/example-store_0.*.1*
```

where the files above that contain a suffix component of "1" (for example, snaboot_0.log.1 and example-store_0.log.1, example-store_0.perf.1,example-store_0.stat.1, etc.) are associated with the sn2 Storage Node.

Executing the above commands should then simulate a catastrophic failure of the "machine" to which sn2 was deployed; where the configuration and data associated with sn2 is now completely unavailable, and is only recoverable via the deployment of

a "new" – and in this example, identical – sn2 Storage Node. To verify this, execute the `show topology` command from the administrative CLI previously started; that is,

```
kv-> show topology
```

which should produce output that looks like the following:

```
store=example-store numPartitions=300 sequence=308
  zn: id=1 name=Zone1 repFactor=3

  sn=[sn1] zn:[id=1 name=Zone1] <host-ip> capacity=1 RUNNING
    [rg1-rn1] RUNNING

  sn=[sn2] zn:[id=1 name=Zone1] <host-ip> capacity=1 UNREACHABLE
    [rg1-rn2] UNREACHABLE

  sn=[sn3] zn:[id=1 name=Zone1] <host-ip> capacity=1 RUNNING
    [rg1-rn3] RUNNING
  ........
```

where the actual IP address or hostname appears instead of the string <host-ip>, and observe that sn2 is now UNREACHABLE.

At this point, the first 2 steps of the SN replacement procedure have been executed. That is, because the sn2 processes have been stopped and their associated files deleted, from the point of view of the store's other nodes, the corresponding "machine" is inaccessible and so has been effectively "shut down" (step 1). Additionally, because a single machine is being used in this simulation, we are already logged in to the sn1 (and sn3) host (step 2). Thus, step 3 of the procedure can now be performed. That is, to retrieve the sn2 configuration from one of the store's remaining healthy nodes, execute the following command using the port for one of those remaining nodes (and remembering to substitute the actual IP address or hostname for the string <host-ip>):

```
> java -Xmx64m -Xms64m \
       -jar /opt/ondb/kv/lib/kvstore.jar generateconfig \
       -host <host-ip> -port 13230 \
       -sn sn2 -target /tmp/sn2-config
```

Verify that the command above produced the expected zip file:

```
> ls -al /tmp/sn2-config.zip
-rw-rw-r-- 1 <group> <owner> 2651 2013-07-08 12:53 /tmp/sn2-config.zip
```

where the contents of `/tmp/sn2-config.zip` should look something like:

```
> unzip -t /tmp/sn2-config.zip

Archive: /tmp/sn2-config.zip
testing: kvroot/config.xml  OK
testing: kvroot/example-store/sn2/config.xml  OK
testing: kvroot/example-store/security.policy  OK
```

```
testing: kvroot/security.policy  OK
No errors detected in compressed data of /tmp/sn2-config.zip
```

Next, because this example is being run on a single machine, steps 4, 5, 6, and 7 of the SN replacement procedure have already been performed. Thus, the next step to perform is to install the contents of the ZIP file just generated; that is,

```
> unzip /tmp/sn2-config.zip -d /opt/ondb/var
```

which will overwrite `kvroot/security.policy` and `kvroot/example-store/security.policy` with identical versions of that file.

When the store was originally deployed, the names of the top-level configuration files were not identical; that is, `config1.xml` for sn1, config2.xml for the originally deployed sn2, and config3.xml for sn3. This was necessary because, for convenience, all three SNs were deployed using the same KVROOT; which would have resulted in conflict among sn1, sn2, and sn3, had identical names been used for those files. With this in mind, it should then be observed that the `generateconfig` command executed above produces a top-level configuration file for the new sn2 that has the default name (config.xml), rather than `config2.xml`. Because both names – `config2.xml` and `config.xml` – are unique relative to the names of the configuration files for the store's other nodes, either name can be used in the next step of the procedure (see below). But to be consistent with the way sn2 was originally deployed, the original file name will also be used when deploying the replacement. Thus, before proceeding with the next step of the procedure, the name of the `kvroot/config.xml` file is changed to `kvroot/config2.xml`; that is,

```
> mv /opt/ondb/var/kvroot/config.xml /opt/ondb/var/kvroot/config2.xml
```

Finally, the last step of the first SN replacement procedure can be performed. That is, a "new" but identical sn2 is started using the old sn2 configuration:

> **Note:**
>
> Before starting the SNA, set the environment variable `MALLOC_ARENA_MAX` to `1`. Setting `MALLOC_ARENA_MAX` to `1` ensures that the memory usage is restricted to the specified heap size.

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config2.xml &
```

Verification

To verify that sn2 has been successfully replaced, first execute the `show topology` command from the administrative CLI; that is,

```
kv-> show topology
```

which should produce output that looks like the following:

```
store=example-store numPartitions=300 sequence=308
  zn: id=1 name=Zone1 repFactor=3

  sn=[sn1] zn:[id=1 name=Zone1] <host-ip> capacity=1 RUNNING
    [rg1-rn1] RUNNING

  sn=[sn2] zn:[id=1 name=Zone1] <host-ip> capacity=1 RUNNING
    [rg1-rn2] RUNNING

  sn=[sn3] zn:[id=1 name=Zone1] <host-ip> capacity=1 RUNNING
    [rg1-rn3] RUNNING
  ........
```

where the actual IP address or hostname appears instead of the string <host-ip>, and observe that sn2 is again RUNNING.

In addition to executing the `show topology` command, you can also verify that the previously removed sn2 directory structure has been recreated and repopulated; that is, directories and files like the following should again exist:

```
/opt/ondb/var/kvroot
  ....
  config2.xml*
  ....
  /example-store
    /log
      ....
      admin2*
      rg1-rn2*
      sn2*
      config.rg1-rn2
      ....
    /sn2
      config.xml
      /admin2
        /env

/tmp/sn2/disk1/ondb/data
  /rg1-rn2
    /env
      00000000.jdb
```

And finally, verify that the data stored previously by the original sn2 has been recovered; that is,

```
> grep "HELLO WORLD" /tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb
  Binary file /tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb matches
```

Example 2: New SN Takes Over Duties of Existing SN

In this example, the second replacement procedure described above will be employed to replace/upgrade an existing, healthy storage node (sn2 in this case) with a new Storage Node that will take over the duties of the old Storage Node. As indicated previously, the assumptions and setup for this example are identical to the first example's assumptions and setup. Thus, after setting up this example as previously specified, start an administrative CLI connected to the admin service associated with the sn1 Storage Node; that is, substituting the actual IP address or hostname for the string <host-ip>, execute the following command:

```
> java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar runadmin \
       -host <host-ip> -port 13230
```

Then, from the administrative CLI just started, execute the `show pools` and `show topology` commands; that is,

```
kv-> show pools
kv-> show topology
```

which should, respectively, produce output that looks something like:

```
AllStorageNodes: sn1 sn2 sn3
snpool: sn1 sn2 sn3
```

and

```
store=example-store numPartitions=300 sequence=308
  zn: id=1 name=Zone1 repFactor=3

  sn=[sn1] zn: [id=1 name=Zone1] host-sn1 capacity=1 RUNNING
    [rg1-rn1] RUNNING

  sn=[sn2] zn:[id=1 name=Zone1] host-sn2 capacity=1 RUNNING
    [rg1-rn2] RUNNING

  sn=[sn3] zn:[id=1 name=Zone1] host-sn3 capacity=1 RUNNING
    [rg1-rn3] RUNNING
  ........
```

> **Note:**
>
> At this point, the pool to join is named "snpool", and the id of the zone to deploy to is "1".

Next, recall that in a production environment, where the old and new SNs run on separate physical machines, the old SN would typically remain up – servicing requests – until the last step of the procedure. In this example though, the old and new SNs run

on a single machine, where the appearance of separate machines and file systems is simulated. Because of this, the next step to perform in this example is to programmatically shut down the sn2 Storage Node by executing the following command:

```
> java -Xmx64m -Xms64m \
       -jar /opt/ondb/kv/lib/kvstore.jar stop \
       -root /opt/ondb/var/kvroot \
       -config config2.xml
```

After stopping the sn2 Storage Node, you might (optionally) execute the `show topology` command and observe that the sn2 Storage Node is no longer RUNNING; rather, it is UNREACHABLE, but will continue to be referenced in the topology until the node is explicitly removed from the topology (see below). For example, from the administrative CLI, execute the following command:

```
kv-> show topology
```

which should produce output that looks like the following:

```
store=example-store numPartitions=300 sequence=308
  zn: id=1 name=Zone1 repFactor=3

  sn=[sn1] zn:[id=1 name=Zone1] host-sn1 capacity=1 RUNNING
    [rg1-rn1] RUNNING

  sn=[sn2] zn:[id=1 name=Zone1] host-sn2 capacity=1 UNREACHABLE
    [rg1-rn2] UNREACHABLE

  sn=[sn3] zn:[id=1 name=Zone1] host-sn3 capacity=1 RUNNING
    [rg1-rn3] RUNNING
  ........
```

At this point, preparation of the new, replacement sn4 storage node can begin; where steps 4, 5, and 6 of the procedure have already been completed, since a single machine hosts both the old and new SN in this example.

With respect to the next step (7), recall that when employing this procedure, step 7 requires that the path of the replacement SN's data directory must be identical to the path used by the SN to be replaced. But in this example, the same disk and file system is used for the location of the data stored by each SN. Therefore, the storage directory that would be created for the new sn4 Storage Node in step 7 already exists and has been populated by the old sn2 Storage Node. Thus, to perform step 7 in this example's simulated environment, as well as to support verification (see below), after shutting down sn2 above, the storage directory used by that node should be renamed; which makes room for the storage directory that needs to be provisioned in step 7 for sn4. That is, type the following at the command line:

```
> mv /tmp/sn2 /tmp/sn2_old
```

> **Note:**
>
> The renaming step above is performed only for this example, and would never be performed in a production environment.

Next, provision the storage directory that sn4 will use; where the path specified must be identical to the original path of the storage directory used by sn2. That is,

```
> mkdir -p /tmp/sn2/disk1/ondb/data
```

The next step to perform when preparing the replacement SN is to generate a boot configuration for the new Storage Node by executing the `makebootconfig` command (remember to substitute the actual IP address or hostname for the string <host-ip>):

```
java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar makebootconfig \
     -root /opt/ondb/var/kvroot \
     -host <host-ip> \
     -config config4.xml \
     -port 13260 \
     -harange 13262,13265 \
     -memory_mb 100 \
     -capacity 1 \
     -admindir /opt/ondb/var/admin \
     -admindirsize 2000 MB \
     -storagedir /tmp/sn2/disk1/ondb/data \
     -rnlogdir /tmp/sn2/disk1/ondb/rnlog
```

which will produce a configuration file for the new Storage Node; `/opt/ondb/var/kvroot/config4.xml`.

After creating the configuration above, use that new configuration to start a new instance of the KVStore Storage Node Agent (SNA), along with its managed services; that is,

> **Note:**
>
> Before starting the SNA, set the environment variable `MALLOC_ARENA_MAX` to `1`. Setting `MALLOC_ARENA_MAX` to `1` ensures that the memory usage is restricted to the specified heap size.

```
> nohup java -Xmx64m -Xms64m \
             -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config4.xml &
```

After executing the command above, use the administrative CLI to deploy a new Storage Node by executing the following command (with the actual IP address or hostname substituted for the string <host-ip>):

```
kv-> plan deploy-sn -znname Zone1 -host <host-ip> -port 13260 -wait
```

As explained previously, because "sn3" was the id assigned (by the store) to the most recently deployed storage node, the next Storage Node that is deployed – that is, the storage node deployed by the command above – will be given "sn4" as its assigned id. After deploying the sn4 Storage Node above, you might then (optionally) execute the `show pools` command from the administrative CLI and observe that the new Storage Node has joined the default pool named "AllStorageNodes"; for example:

```
kv-> show pools
```

which should produce output that looks like the following:

```
AllStorageNodes: sn1 sn2 sn3 sn4
snpool: sn1 sn2 sn3
```

where upon deployment, although sn4 has joined the pool named "AllStorageNodes", it has not yet joined the pool named "snpool".

Next, after successfully deploying the sn4 Storage Node, use the CLI to join the pool named "snpool"; that is:

```
kv-> pool join -name snpool -sn sn4
```

After deploying the new Storage Node and joining the pool named "snpool", using the administrative CLI, you might (optionally) execute the `show topology` command followed by the `show pools` command; and then observe that the new Storage Node has been deployed to the store and has joined the pool named "snpool"; for example,

```
kv-> show topology
kv-> show pools
```

which, given the initial assumptions, should produce output that looks like the following:

```
store=example-store numPartitions=300 sequence=308
  zn: id=1 name=Zone1 repFactor=3

  sn=[sn1] zn:[id=1 name=Zone1] host-sn1 capacity=1 RUNNING
    [rg1-rn1] RUNNING

  sn=[sn2] zn:[id=1 name=Zone1] host-sn2 capacity=1 UNREACHABLE
    [rg1-rn2] UNREACHABLE

  sn=[sn3] zn:[id=1 name=Zone1] host-sn3 capacity=1 RUNNING
    [rg1-rn3] RUNNING
```

```
  sn=[sn4] zn:[id=1 name=Zone1] host-sn4 capacity=1 RUNNING
  ........
```

and

```
AllStorageNodes: sn1 sn2 sn3 sn4
snpool: sn1 sn2 sn3 sn4
```

The output above shows that the sn4 Storage Node has been successfully deployed (is RUNNING) and is now a member of the pool named "snpool". But it does not yet include an RN service corresponding to sn4. Such a service will not appear in the store's topology until sn2 is migrated to sn4 (see below).

At this point, after the sn4 Storage Node is deployed and has joined the pool named "snpool", and the old sn2 Storage Node has been stopped, sn4 is ready to take over the duties of sn2. This is accomplished by migrating the sn2 services and data to sn4 by executing the following command from the administrative CLI (remembering to substitute the actual IP address or hostname for the string<host-ip>):

```
kv-> plan migrate-sn -from sn2 -to sn4 -wait
```

After migrating sn2 to sn4 you might (optionally) execute the `show topology` command again and observe that the rg1-rn2 service has moved from sn2 to sn4 and is now RUNNING; that is,

```
kv-> show topology

store=example-store numPartitions=300 sequence=308
  zn: id=1 name=Zone1 repFactor=3

  sn=[sn1] zn:[id=1 name=Zone1] host-sn1 capacity=1 RUNNING
    [rg1-rn1] RUNNING

  sn=[sn2] zn:[id=1 name=Zone1] host-sn2 capacity=1 UNREACHABLE

  sn=[sn3] zn:[id=1 name=Zone1] host-sn3 capacity=1 RUNNING
    [rg1-rn3] RUNNING

  sn=[sn4] zn:[id=1 name=Zone1] host-sn4 capacity=1 RUNNING
    [rg1-rn2] RUNNING
  ........
```

Finally, after the migration process is complete, remove the old sn2 Storage Node from the store's topology; which can be accomplished by executing the `plan remove-sn` command from the administrative CLI in the following way:

```
kv-> plan remove-sn -sn sn2 -wait
```

Verification

To verify that sn2 has been successfully replaced/upgraded by sn4, first execute the `show topology` command from the previously started administrative CLI; that is,

```
kv-> show topology
```

The output is like the following:

```
store=example-store numPartitions=300 sequence=308
  zn: id=1 name=Zone1 repFactor=3

  sn=[sn1] zn:[id=1 name=Zone1] <host-ip> capacity=1 RUNNING
    [rg1-rn1] RUNNING

  sn=[sn3] zn:[id=1 name=Zone1] <host-ip> capacity=1 RUNNING
    [rg1-rn3] RUNNING

  sn=[sn4] zn:[id=1 name=Zone1] <host-ip> capacity=1 RUNNING
    [rg1-rn2] RUNNING
  ........
```

Here the actual IP address or hostname appears instead of the string <host-ip>, and only sn4 appears in the output rather than sn2.

In addition to executing the `show topology` command, you can also verify that the expected sn4 directory structure is created and populated; that is, directories and files like the following should exist:

```
/opt/ondb/var/kvroot
  ....
  config4.xml
  ....
  /example-store
    /log
      ....
      sn4*
      ....
    /sn4
      config.xml
      /admin2
        /env

/tmp/sn2/disk1/ondb/data
  /rg1-rn2
    /env
      00000000.jdb
```

You can also verify that the data stored previously by sn2 has been migrated to sn4; that is:

```
> grep "HELLO WORLD" /tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb
  Binary file /tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb matches
```

Note:

Although sn2 was stopped and removed from the topology, the data files created and populated by sn2 in this example were not deleted. They were moved under the `/tmp/sn2_old` directory. Thus, the old sn2 storage directory and data files can still be accessed. That is:

```
/tmp/sn2_old/disk1/ondb/data
  /rg1-rn2
    /env
      00000000.jdb
```

And the original key/value pair should still exist in the old sn2 data file; that is,

```
> grep "HELLO WORLD" \
  /tmp/sn2_old/disk1/ondb/data/rg1-rn2/env/00000000.jdb
  Binary file
  /tmp/sn2_old/disk1/ondb/data/rg1-rn2/env/00000000.jdb
  matches
```

Finally, the last verification step that can be performed is intended to show that the new sn4 Storage Node has taken over the duties of the old sn2 Storage Node. This step consists of writing a new key/value pair to the store and then verifying that the new pair has been written to the data files of sn1, sn3, and sn4, as was originally done with sn1, sn3, and sn2 prior to replacing sn2. To perform this step, you can use the KVStore client shell utility in the same way as described in Setup , when the first key/value pair was initially inserted. That is, you can execute the following (remembering to substitute the actual IP address or hostname for the <host-ip> string):

```
> java -Xmx64m -Xms64m \
-jar /opt/ondb/kv/lib/kvstore.jar runadmin\
       -host <host-ip> -port 13230 -store example-store

kv-> get -all
  /FIRST_KEY
  HELLO WORLD

kv-> put -key /SECOND_KEY -value "HELLO WORLD 2"
  Put OK, inserted.

kv-> get -all
  /SECOND_KEY
  HELLO WORLD 2
  /FIRST_KEY
  HELLO WORLD
```

After performing the insertion, use the "grep" command to verify that the new key/value pair was written by sn1, sn3, and sn4; and of course, the old sn2 data file still only contains the first key/value pair. That is,

```
> grep "HELLO WORLD 2" /tmp/sn1/dsk1/ondb/data/rg1-rn1/env/00000000.jdb
  Binary file /tmp/sn1/disk1/ondb/data/rg1-rn1/env/00000000.jdb matches
> grep "HELLO WORLD 2" /tmp/sn2/dsk1/ondb/data/rg1-rn2/env/00000000.jdb
  Binary file /tmp/sn2/disk1/ondb/data/rg1-rn2/env/00000000.jdb matches
> grep "HELLO WORLD 2" /tmp/sn3/dsk1/ondb/data/rg1-rn3/env/00000000.jdb
  Binary file /tmp/sn3/disk1/ondb/data/rg1-rn3/env/00000000.jdb matches
> grep "HELLO WORLD 2"
       /tmp/sn2_old/dsk1/ondb/data/rg1-rn2/env/00000000.jdb
```

Standardized Monitoring Interfaces

In addition to the native monitoring provided by the Admin CLI, Oracle NoSQL Database allows Java Management Extensions (JMX) agents to be optionally available for monitoring. These agents provide interfaces on each storage node that allow management clients to poll them for information about the status, performance metrics, and operational parameters of the storage node and its managed services, including replication nodes, and admin instances. You can also use JMX to monitor Arbiter Nodes.

Both of these management agents can also be configured to push notifications about status changes of any services, and for violations of preset performance limits.

You can enable the JMX interface in either the Community Edition or the Enterprise Edition.

The JMX service exposes MBeans for the three types of components. These MBeans are the java interfaces StorageNodeMBean, RepNodeMBean, and AdminMBean in the package oracle.kv.impl.mgmt.jmx. For more information about the status reported for each component, see the javadoc for these interfaces.

> **Note:**
>
> For information on using JMX securely, see Guidelines for using JMX securely in the *Security Guide*.

Java Management Extensions (JMX)

JMX agents in Oracle NoSQL Database are read-only interfaces. These interfaces let you poll Storage Nodes for information about the storage node and about any replication nodes or Admins that the Storage Node hosts. The information available from polling includes the service status (RUNNING, STOPPED, and so on), operational parameters, and performance metrics.

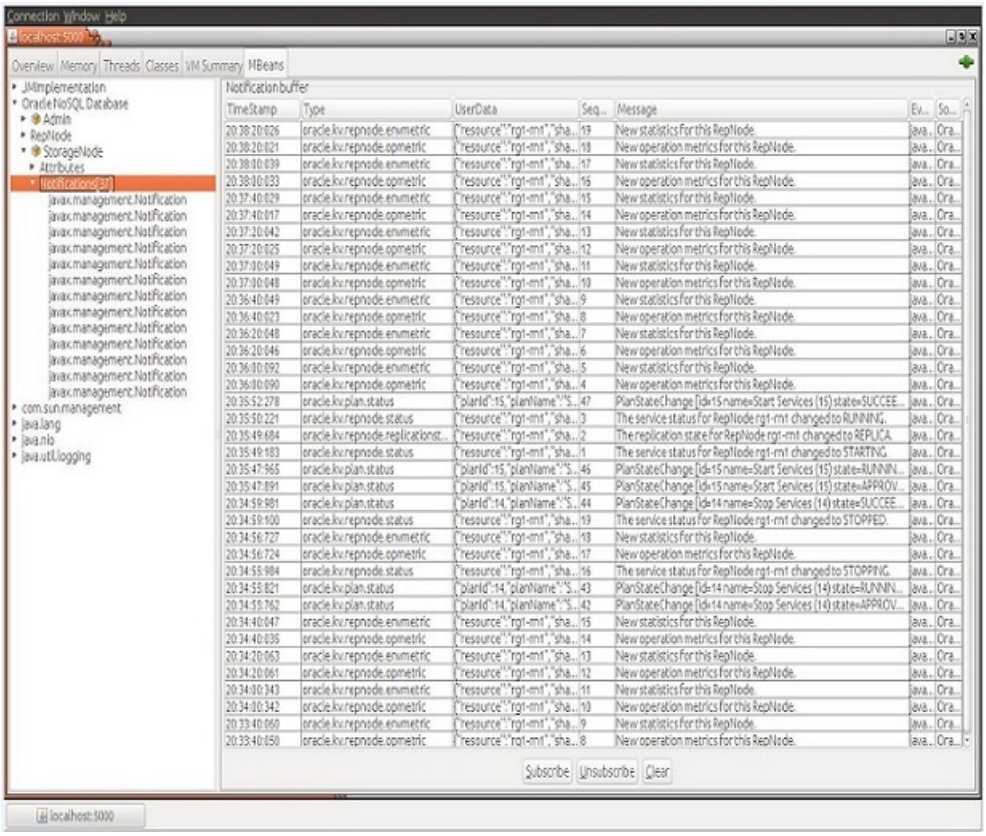
JMX agents also deliver event traps and notifications for particular events. For example, JMX sends notifications for every service status state change, and any performance limits that the store exceeds.

Displaying the Oracle NoSQL Database MBeans

You can view the Oracle NoSQL Database JMX MBeans in a monitoring tool such as JConsole. In the JConsole, use the Remote Process option and connect using hostname and registry port for each Storage Node that you would like to view.

> **Note:**
>
> You should not choose Local Process option to connect to Oracle NoSQL database.

For example, in the following case, in the JConsole New Connection window, you should choose the Remote Process option and specify localhost:5000 to connect to Oracle NoSQL Database.



Enabling JMX Monitoring

You can enable monitoring on a per-storage node basis in two different ways:

- In the boot configuration file.
- Change the `mgmtClass` parameter.

In the Bootfile

You can specify that you want to enable JMX in the boot configuration file for the Storage Node.

> **Note:**
>
> When you specify -mgmt jmx, a storage node's JMX agent uses the RMI registry at the same port number as it uses for all other RMI services that the Storage Node manages.

By Changing Storage Node Parameters

You can enable JMX after you deploy a store by changing the storage node parameter `mgmtClass`.

The `mgmtClass` parameter value may be one of the following class names:

- To enable JMX:

  ```
  oracle.kv.impl.mgmt.jmx.JmxAgent
  ```

- To disable JMX:

  ```
  oracle.kv.impl.mgmt.NoOpAgent
  ```

Using ELK to Monitor Oracle NoSQL Database

“ELK” is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana.

Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then send it to a “stash” like Elasticsearch. Kibana lets users visualize data with charts and graphs in Elasticsearch. The ELK stack can be used to monitor Oracle NoSQL Database.

> **Note:**
>
> For a Storage Node Agent (SNA) to be discovered and monitored, it must be configured for JMX. JMX is not enabled by default. You can tell whether JMX is enabled on a deployed SNA issuing the `show parameters` command and checking the reported value of the mgmtClass parameter. If the value is not `oracle.kv.impl.mgmt.jmx.JmxAgent`, then you need to issue the `change-parameters` plan command to enable JMX.
>
> For example:
>
> ```
> plan change-parameters -service sn1 -wait \
> -params mgmtClass=oracle.kv.impl.mgmt.jmx.JmxAgent
> ```
>
> For more information, see Standardized Monitoring Interfaces .

You can integrate Oracle NoSQL metrics and log information **with an existent ELK configuration**. It is recommended to install ELK components in dedicated servers. Do not use Oracle NoSQL nodes. For more information, see Elastic Stack and Product documentation. If you do not have an existent ELK setup, follow the instructions provided but always refer to elastic documentation for details. Configuration files provided must be edited if you decide to use a secure ELK deployment, according to your setup and your security requirements.

Filebeat and metricbeat help you monitor your servers and the services they host by collecting metrics, especially from the NoSQL services. Filebeat is the component that sends the files generated by the **Collector Service** in each Storage Node (SN) to ELK for analysis and monitoring. Filebeat component must be deployed in all NoSQL nodes of your cluster.

You'll learn how to:

- install and configure Filebeat on each NoSQL cluster you want to monitor
- specify the metrics you want to collect
- send the metrics to Logstash and then to Elasticsearch
- visualize the metrics data in Kibana
- change the configuration files if you decide to use a secure ELK deployment

Enabling the Collector Service

Follow the steps below to enable collector service in Oracle NoSQL Database:

1. Set the `collectorEnabled` parameter across the store to `true`.

   ```
   plan change-parameter -global -wait -params collectorEnabled=true
   ```

2. Set an appropriate value for `collectorInterval`. Low interval value collects more details and requires more storage. High interval value comparatively collects lesser details and requires lesser storage.

   ```
   plan change-parameter -global -wait -params collectorInterval="30 s"
   ```

3. Provide an appropriate storage size for `collectorStoragePerComponent`. The data collected by each component (each SN and RN) is stored in a buffer. This buffer size can be changed by setting this parameter.

   ```
   plan change-parameter -global -wait -params
   collectorStoragePerComponent="50 MB"
   ```

Setting Up Elasticsearch

Follow the steps below to setup Elasticsearch:

1. Download and decompress Elasticsearch 8.7.0.
2. Modify the `$ELASTICSEARCH/config/elasticsearch.yml` file as per your configuration.

   For example: Set values for `path.data` and `path.logs` to store data and logs in the specified location.
3. Startup Elasticsearch.

   ```
   $ cd $ELASTICSEARCH
   $ sudo sysctl -q -w vm.max_map_count=262144;
   $ nohup bin/elasticsearch &
   ```

For more information, see Elasticsearch Reference guide.

Setting Up Kibana

Follow the steps below to setup Kibana:

1. Download and decompress Kibana 8.7.0.
2. Modify the `$KIBANA/config/kibana.yml` file as per your configuration.

   For example: If Elasticsearch is not deployed on the same machine as Kibana, add line `elasticsearch.url:”<your_es_hostname>:9200”`. This sets Kibana to connect to the Elasticsearch address specified instead of `127.0.0.1:9200`.
3. Startup Kibana.

   ```
   $ cd $KIBANA
   $ nohup bin/kibana &
   ```

For more information, see Kibana Reference guide.

Setting Up Logstash

Follow the steps below to setup Logstash:

1. Download and decompress Logstash 8.7.0.

2. Place the `logstash.config` file in the same directory where Logstash is decompressed. Modify the `logstash.config` file as per your configuration.

   For example: If Elasticsearch is not deployed on the same machine as Logstash, change the Elasticsearch hosts from `localhost:9200` to `<your_es_hostname>:9200`.

3. Place the templates (`kvevents.template, kvpingstats.template, kvrnenvstats.template, kvrnjvmstats.template, kvrnopstats.template`) in the same directory where Logstash is decompressed. Modify the templates as per your configuration.

4. Switch to the `$LOGSTASH` directory . Verify that the directory contains the Logstash setup files, configuration file, and all the templates. Then, startup Logstash.

```
$ cd $LOGSTASH
$ logstash-5.6.4/bin/logstash -f logstash.config &
```

For more information, see Logstash Reference guide.

Setting Up Filebeat on Each Storage Node

Follow the steps below to setup Filebeat on each storage node:

1. Download and decompress Filebeat 8.7.0.
2. Replace the existing `filebeat.yml` with `filebeat.yml`. Edit the file and replace all occurrences of `/path/of/kvroot` with the actual KVROOT path of this SN. Also, replace `LOGSTASH_HOST` with the actual IP of Logstash.
3. Startup Filebeat.

```
$ cd $FILEBEAT
$ ./filebeat &
```

4. Repeat the above steps in all the storage nodes of the cluster.

For more information, see Filebeat Reference guide.

Configure security for the Elastic Stack

Security needs vary depending on whether you're developing a test environment or securing all communications in a production environment. The following scenarios provide different options for configuring the Elastic Stack.

Minimal security (Elasticsearch Development)

You can use this to set up Elasticsearch on your test environment. This configuration prevents unauthorized access to your cluster by setting up passwords for the built-in users. You also configure password authentication for Kibana. This minimal security scenario is not sufficient for production mode clusters. If your cluster has multiple nodes, you must enable minimal security and then configure Transport Layer Security (TLS) between nodes.

Basic security (Elasticsearch Production)

This scenario builds on the minimal security requirements by adding transport Layer Security (TLS) for communication between nodes. This additional layer requires that

nodes verify security certificates, which prevents unauthorized nodes from joining your Elasticsearch cluster.

Basic security plus secured HTTPS traffic (Elastic Stack)

This scenario builds on the one for basic security and secures all HTTP traffic with TLS. In addition to configuring TLS on the transport interface of your Elasticsearch cluster, you configure TLS on the HTTP interface for both Elasticsearch and Kibana. If you need mutual (bidirectional) TLS on the HTTP layer, then you'll need to configure mutual authenticated encryption. You then configure Kibana and Beats to communicate with Elasticsearch using TLS so that all communications are encrypted. This level of security is strong, and ensures that any communications in and out of your cluster are secure. For more information, see Configuring Stack Security.

In this case, you must modify the logstash.config as per your configuration. You need provide the user/password and the TLS certificate to do the connection. For example:

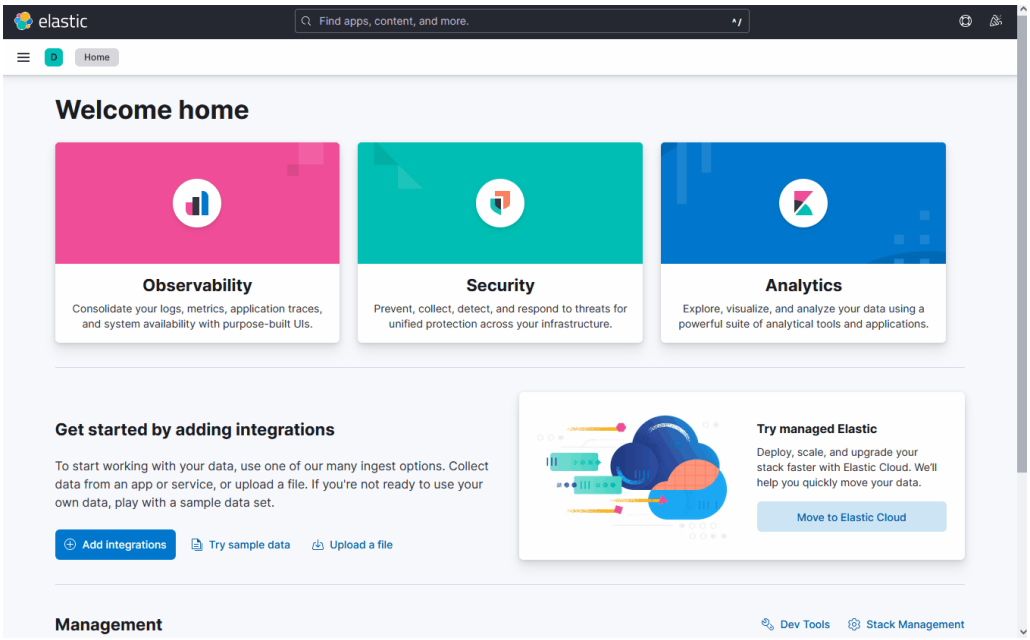
```
 elasticsearch {
            manage_template => true
            template => "kvrnjvmstats.template"
            template_name => "kvrnjvmstats"
            template_overwrite => true
            index => "kvrnjvmstats-%{+YYYY.MM.dd}"
            hosts => "elk-node1:9200"
            user => "logstash_internal"
            password => "thepwd"
            cacert => '/etc/logstash/config/certs/ca.crt'
}
```

Using Kibana for Analyzing Oracle NoSQL Database

You will learn how to visualize and monitor NoSQL metrics using Kibana. Kibana requires an index pattern to access the Elasticsearch data that you want to explore. An index pattern selects the data to use and allows you to define the properties of the fields.

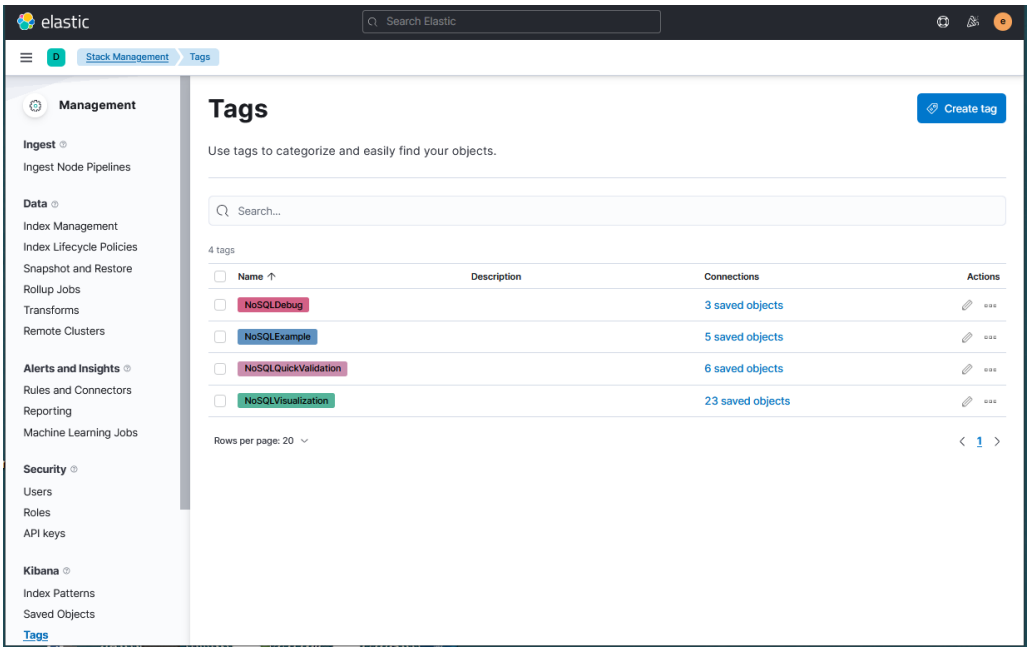
Using the import and export actions, you can move objects between different Kibana instances. This action is useful when you have multiple environments for development and production. Import and export also work well when you have a large number of objects to update and want to batch process them. The **Saved Objects** UI helps you to keep track of and manage your saved objects. These objects store data for later use, including dashboards, visualizations, maps, index patterns, Canvas workpads, and more.

In order to improve the task of creating dashboards, visualizations, index patterns, you can use this json file that you can import in your Kibana configuration. This is very handy to import this json file so that you can have access to the dashboards, visualizations, index patterns that has been provided.



The save objects are classified into 4 types using tags.

1. NoSQLVisualization
2. NoSQLExample
3. NoSQLDebug
4. NoSQLQuickValidation

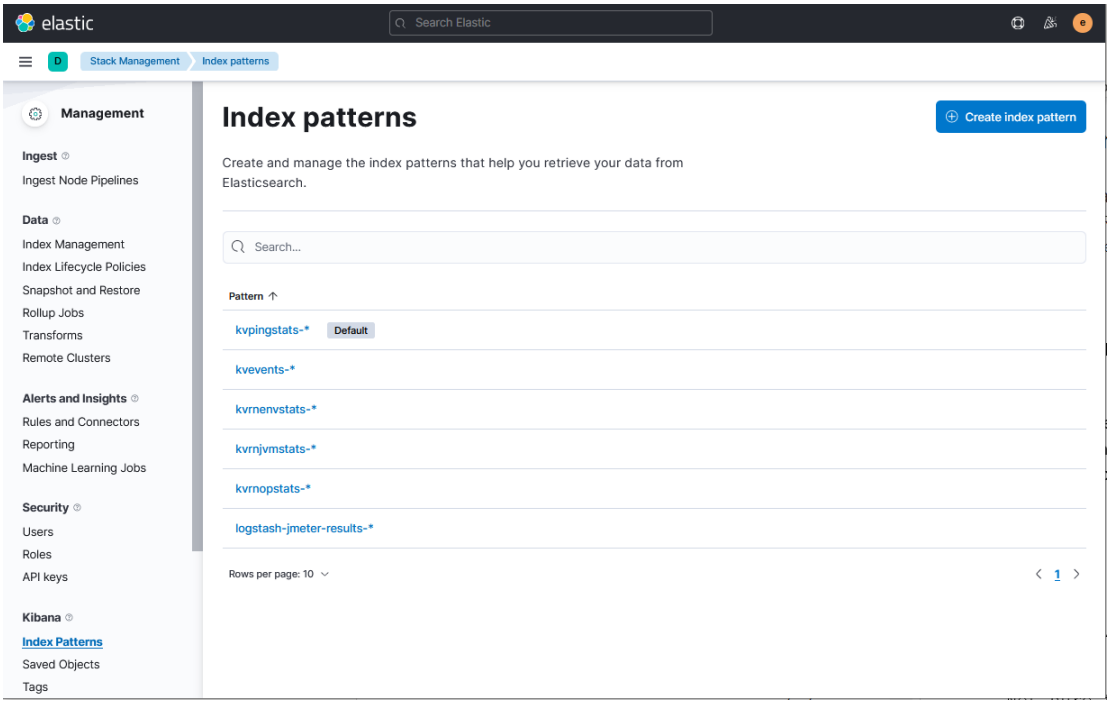


Creating Index Patterns

By using the command import, you automatically configure index patterns.

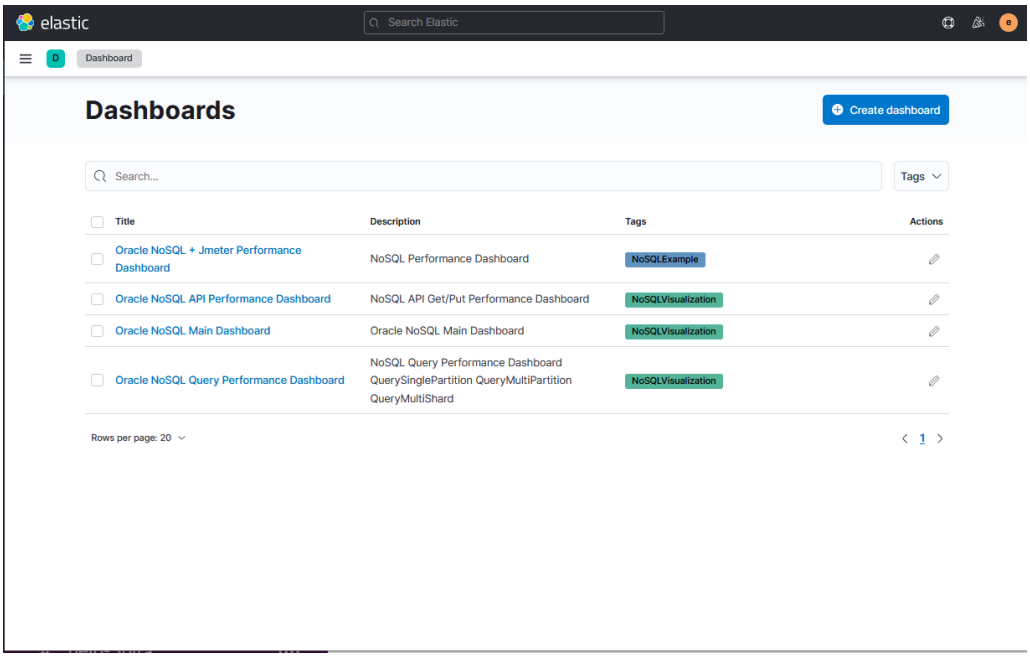
1. `kvrnjvmstats-*`
2. `kvrnenvstats-*`
3. `kvpingstats-*`
4. `kvrnopstats-*`
5. `kvevents-*`

> **Note:**
>
> This index may not exist if the store is brand new as no events have occurred.

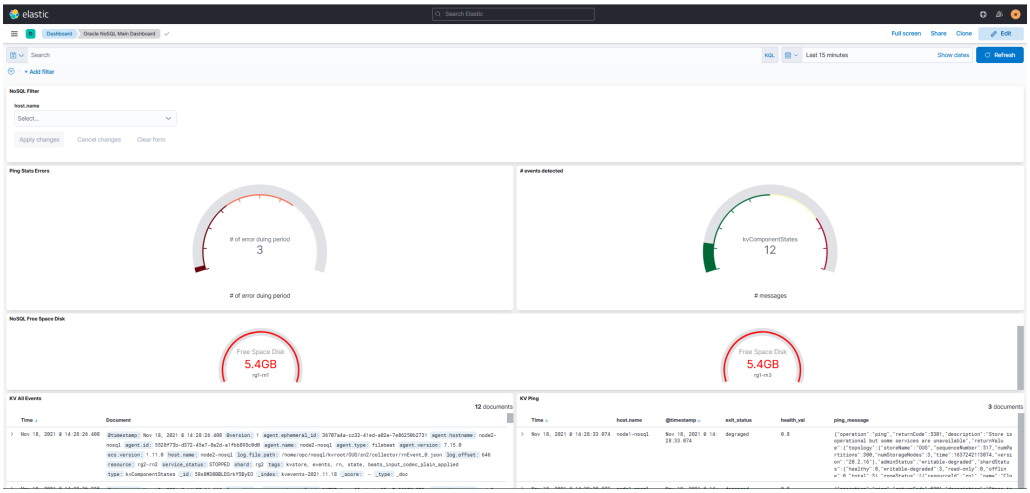


Analyzing the Data

By using the command import, you create standard visualizations that can be explored using dashboards.



The **Oracle NoSQL Main Dashboard** shows you the important information about the health of the cluster.

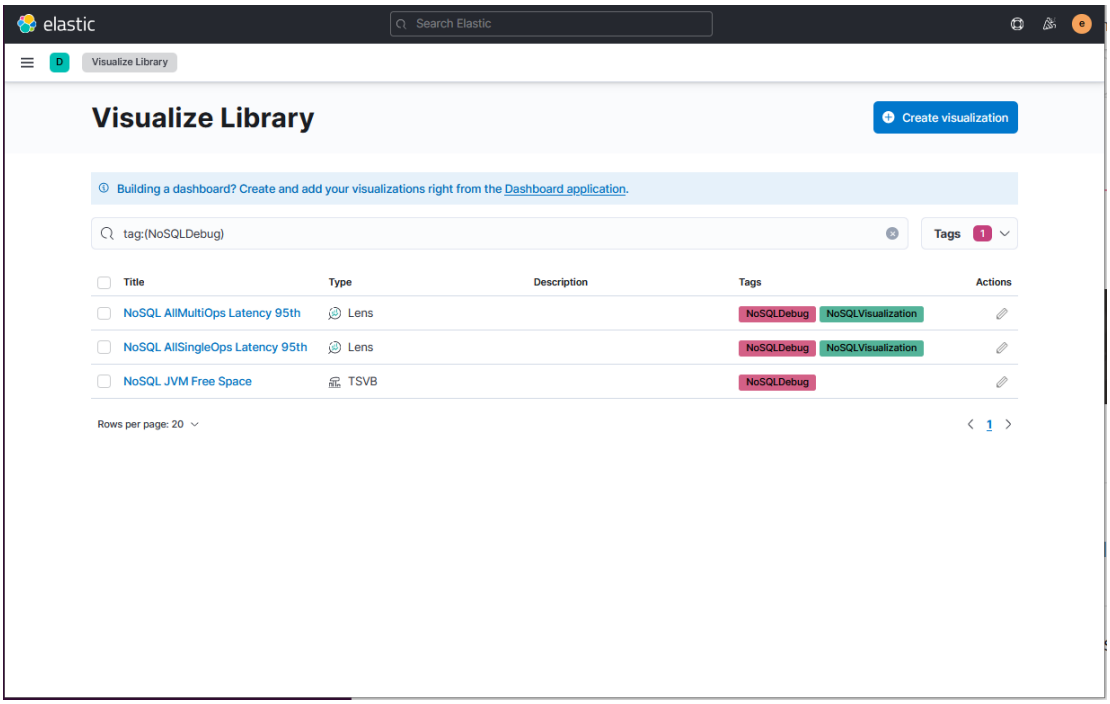


The **Oracle NoSQL API Performance Dashboard** and **Oracle NoSQL Query Performance Dashboard** shows you summarized information about the performance of the NoSQL cluster. You provide information about throughput and average latency for the following important API calls.

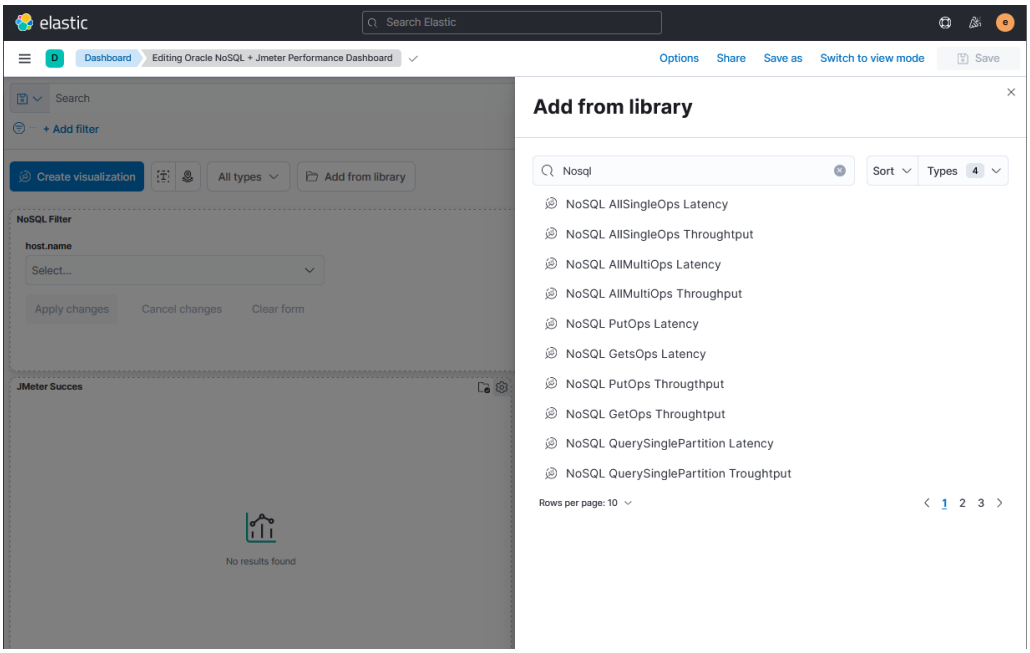
1. NoSQL AllSingleOps/AllMultiOps shows throughput and average latency.
2. NoSQL QuerySinglePartition/NoSQL QueryMultiShard/NoSQL QueryMultiPartition shows detailed information on SQL query performance.
   - QuerySinglePartition, operations using primary key indexes
   - QueryMultiShard, operations using secondary indexes
   - QueryMultiPartition, operations doing full scans

3. PutOps/ GetOps shows detailed information on CRUD API calls.

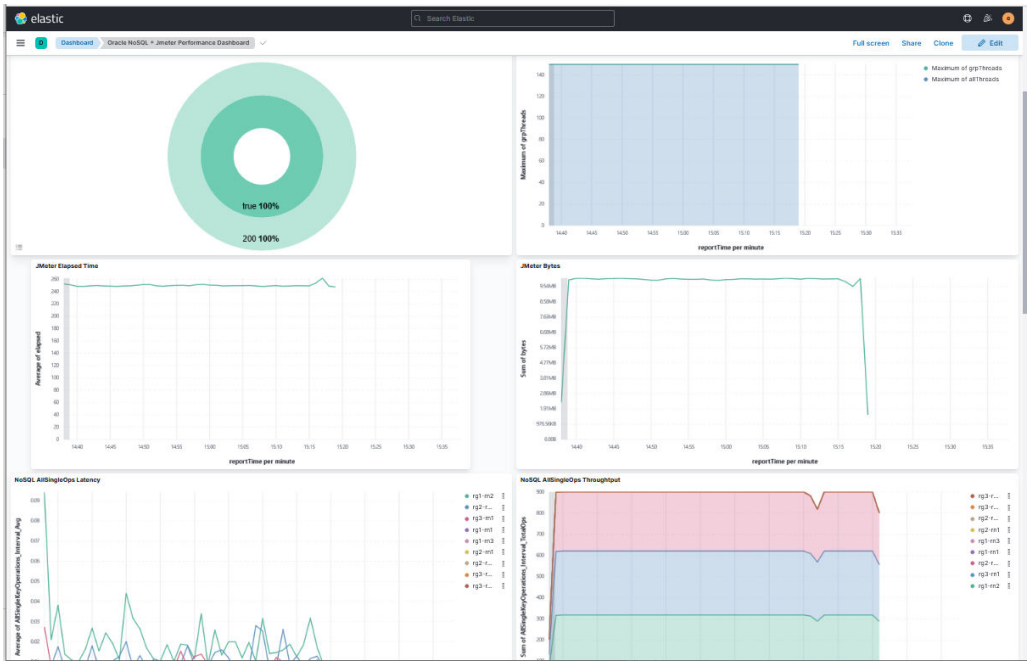
Oracle NoSQL exposes hundreds of metrics that could be valuable when debugging. In order to visualize this information, visualizations are provided in the NoSQLDebug category.You can use these visualization and also create more customized visualizations when necessary. These metrics contain statistics of each type of API operation. And each operation statistics is calculated by interval and cumulative statistics.



By providing standard Kibana Visualization, you can add this information to your Standard Application Performance Kibana dashboards. It allows you to do a correlation between the application activity, the NoSQL performance and other OS statistics collected (e.g CPU metrics collected using Metricbeat).



The **Oracle NoSQL + Jmeter Performance Dashboard** is a good example of the applications dashboards allowing these kind of correlations. The dashboard is used to show how to do the correlation between Application Performance statistics (provided by JMeter) and NoSQL standard performance metrics.



7
Reference

The articles in this section contains reference information on various command line tools and utilities.

Terminologies used in Oracle NoSQL Database

Some of the terminologies used in Oracle NoSQL Database

Release: Oracle NoSQL database is released three times in a year. The release number of the Oracle NoSQL database follows this pattern `release.major.minor` where **release** is the last 2 characters of the year(For example: 23), **major release number** denotes the quarter which is 1,2, or 3 and **minor release number** is the final patch number of the release. For example : `23.1.16` .

Storage Node: A Storage Node is a physical (or virtual) machine with its own local storage, which houses the Replication Node

Replication Node: Every Storage Node hosts one or more replication nodes(RN) as determined by its capacity. A Storage Node's capacity serves as a rough measure of the hardware resources associated with it (memory, CPUs, and disks). , each of which hosts one or more replication node.

Storage Node Agent: . The Storage Node Agent manages all the Replication Nodes running on the Storage Node (host)

Shard: A shard is a horizontal partition of data in a database. Your data store's replication nodes are organized into shards. A single shard contains multiple replication nodes.

Majority in a Shard: The number of replication nodes which are available and online in a shard. This will help in ensuring read and write availability in the shard and to elect master when one of the nodes fail.

Master Node and Replica Nodes: There are two types of Replication Nodes, namely, **master** and **replica.** Each shard must contain one master node. The master node performs all database write activities. Each shard can also contain one or more read-only replicas. The master node copies all new write activity data to the replicas. The replicas are then used to service read-only operations.

Quorum: Quorum is the minimum number of primary nodes required in a shard, or in the set of admin nodes, to permit electing a master to support write operations.

Zone: A zone is a physical location that supports high-capacity network connectivity between the Storage Nodes deployed within it.

Master Affinity Zones: Master Affinity is a way for you to indicate which primary zones can host master replication nodes. Master Affinity zones service high demand "write" requests across shards.

Replication factor: The total number of masters and replicas in a shard are equal to the replication factor (RF). You can also think of Replication Factor as the number of copies of your data.

Zone Replication factor: The number of copies, or replicas, maintained in a zone.

Primary Replication factor: The total number of replicas in all primary zones.

Secondary Replication factor: The total number of replicas in all secondary zones.

Store Replication factor: The total number of replicas in all zones across the entire store.

Arbiter Node: An Arbiter Node is a lightweight process that is capable of supporting write availability when the primary replication factor is two and a single replication node becomes unavailable or when two replication nodes are unable to communicate to determine which one of them is the master.

Admin CLI Reference

This appendix describes the following commands:

- aggregate
- await-consistent
- change-policy
- configure
- connect
- delete
- execute
- exit
- get
- help
- hidden
- history
- load
- logtail
- namespace
- page
- ping
- plan
- pool
- put
- repair-admin-quorum
- show
- snapshot
- table
- table-size

- timer
- topology
- verbose
- verify

The Command Line Interface (CLI) is run interactively or used to run single commands. The general usage to start the CLI is:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar runadmin \
-host <hostname> -port <port> [single command and arguments]
-security KVROOT/security/client.security
```

If you want to run a script file, you can use the "load" command on the command line:

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar runadmin -host <hostname> -port <port> \
-security \
KVROOT/securtiy/client.security \
load -file <path-to-script>
```

If none of the optional arguments are passed, it starts interactively. If additional arguments are passed they are interpreted as a single command to run, then return. The interactive prompt for the CLI is:

```
"kv-> "
```

Upon successful completion of the command, the CLI's process exit code is zero. If there is an error, the exit code will be non-zero.

The CLI comprises a number of commands, some of which have subcommands. Complex commands are grouped by general function, such as "show" for displaying information or "ddl" for manipulating schema. All commands accept the following flags:

- -help

  Displays online help for the command or subcommand.
- ?

  Synonymous with -help. Displays online help for the command or subcommand.
- -verbose

  Enables verbose output for the command.

CLI commands have the following general format:

1. All commands are structured like this:

   ```
   "kv-> command [sub-command] [arguments]
   ```

2. All arguments are specified using flags which start with "-"
3. Commands and subcommands are case-insensitive and match on partial strings(prefixes) if possible. The arguments, however, are case-sensitive.

Inside a CLI script file, you can use # to designate a comment. Also, you can terminate a line with a backslash \ to continue a command onto the next line.

aggregate

Performs simple data aggregation operations on numeric fields like count, sum, average, keys, start and end. The aggregate command iterates matching keys or rows in the store so, depending on the size of the specified key or row, it may take a very long time to complete.

aggregate table is an aggregate subcommand.

aggregate table

```
aggregate table -name <name>
    [-count] [-sum <field[,field,..]>]
    [-avg <field[,field,..]>]
    [-index <name>]
    [-field <name> -value <value>]*
    [-field <name> [-start <value>] [-end <value>]]
    [-json <string>]
```

Performs simple data aggregation operations on numeric fields of the table.

where:

- `-name`

  Specifies the table for the operation.

- `-count`

  Returns the count of matching records.

- `-sum`

  Returns the sum of the values of matching fields.

- `-avg`

  Returns the average of the values of matching fields.

- `-index`

  Specifies the name of the index to use. When an index is used, the fields named must belong to the specified index and the aggregation is performed over rows with matching index entries.

- `-field` and `-value` pairs are used to specify the field values of the primary key to use to match for the aggregation, or you can use an empty key to match the entire table.

- The `-field` flat, along with its `-start` and `-end` flags, can be used for restricting the range used to match rows.

- `-json`

  Specifies the fields and values to use for the aggregation as a JSON input string.

See the example below:

```
# Create a table 'user_test' with an index on user_test(age):
kv-> execute 'CREATE TABLE user_test (id INTEGER,
firstName STRING, lastName STRING, age INTEGER, PRIMARY KEY (id))'
Statement completed successfully

kv-> execute 'CREATE INDEX idx1 on user_test (age)'
Statement completed successfully

# Insert 3 rows:
kv-> put table -name user_test -json
'{"id":1,"firstName":"joe","lastName":"wang","age":21}'
Operation successful, row inserted.
kv-> put table -name user_test -json
'{"id":2,"firstName":"jack","lastName":"zhao","age":32}'
Operation successful, row inserted.
kv-> put table -name user_test -json
'{"id":3,"firstName":"john","lastName":"gu","age":43}'
Operation successful, row inserted.

# Get count(*), sum(age) and avg(age) of rows in table:
kv-> aggregate table -name user_test -count -sum age -avg age
Row count: 3
Sum:
        age(3 values): 96
Average:
        age(3 values): 32.00

# Get count(*), sum(age) and avg(age) of rows where
age >= 30, idx1 is utilized to filter the rows:
kv-> aggregate table -name user_test -count -sum age
-avg age -index idx1 -field age -start 30
Row count: 2
Sum:
        age(2 values): 75
Average:
        age(2 values): 37.50
```

await-consistent

```
await-consistent -timeout <timeout-secs> [-zn <id> | -znname <name> ]...
[-replica-delay-threshold <time-millis>]
```

Waits for up to the specified number of seconds for the replicas in one or more zones, or in the entire store, to catch up with the masters in their associated shards. Prints information about whether consistency was achieved or, if not, details about which nodes failed to become consistent.

where:

- `-timeout`

Specifies the number of seconds for the replicas to catch up with the masters in their associated shards.

- `-zn <id>`

  Specifies the zone name to restrict the zones whose replicas need to satisfy the requested consistency requirements. If this option is not specified, all replicas must meet the consistency requirements.

- `-znname <name>`

  Specifies the zone name to restrict the zones whose replicas need to satisfy the requested consistency requirements. If this option is not specified, all replicas must meet the consistency requirements.

- `-replica-delay-threshold <time-millis>`

  Specifies the maximum number of milliseconds that a replica may be behind the master and be considered caught up. The default if 1000 milliseconds (1 second).

When performing a switchover, you can use this command to wait for secondary nodes to catch up with their masters, and to obtain information about progress towards reaching consistency.

change-policy

```
change-policy [-dry-run] -params [name=value]*
```

Modifies store-wide policy parameters to services you have not yet deployed. Specify the parameters to change after the `-params` flag, separating each parameter with a space character.

To specify parameter values that include embedded spaces, use quotation marks (`"`) around the value, like this:

```
name="value with spaces"
```

If you use `-dry-run`, the command returns the parameters you specify without changing them.

For more information on setting policy parameters, see Setting Store Wide Policy Parameters.

configure

```
configure -name <storename> -json
```

Configures a new store. This call must be made before any other administration can be performed.

Use the `-name` option to specify the name of the KVStore that you want to configure. The name is used to form a path to records kept in the store. For this reason, you should avoid using characters in the store name that might interfere with its use within

a file path. The command line interface does not allow an invalid store name. Valid characters are alphanumeric, '-', '_', and '.'.

```
kv-> configure -name mystore -json{
"operation" : "configure",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "storeName" : "mystore"
    }
}
```

connect

Encapsulates commands that connect to the specified host and registry port to perform administrative functions or connect to the specified store to perform data access functions.

The current store, if any, will be closed before connecting to another store. If there is a failure opening the specified KVStore, the following warning is displayed: "Warning: You are no longer connected to KVStore".

The subcommands are as follows:

- connect admin
- connect store

connect admin

```
connect admin -host <hostname> -port <registry port>
[-username <user>] [-security <security-file-path>]
```

Connects to the specified host and registry port to perform administrative functions. An Admin service must be active on the target host. If the instance is secured, you may need to provide login credentials.

where:

- `-host <hostname>`

  Identifies the host name of a node in your store.

- `-port <registry port>`

  The TCP/IP port on which Oracle NoSQL Database should be contacted. This port should be free (unused) on each node. It is sometimes referred to as the registry port.

- `-username <user>`

  Specifies a username to log on as in a secure deployment.

- `-security <security-file-path>`

  In a secured deployment, specifies a path to the security file. If not specified in a secure store, updating the sn-target-list will fail.

connect store

```
connect store [-host <hostname>] [-port <port>]
-name <storename> [-timeout <timeout ms>]
[-consistency <NONE_REQUIRED(default)
                                | ABSOLUTE|
NONE_REQUIRED_NO_MASTER>]
[-durability <COMMIT_SYNC(default)
                              | COMMIT_NO_SYNC |
COMMIT_WRITE_NO_SYNC>]
[-username <user>] [-security <security-file-path>]
```

Connects to a KVStore to perform data access functions. If the instance is secured, you may need to provide login credentials.

Use the timeout, consistency and durability flags to override the default connect configuration.

where:

- `-host <hostname>`

  Identifies the host name of a node in your store.

- `-port <port>`

  The TCP/IP port on which Oracle NoSQL Database should be contacted. This port should be free (unused) on each node.

- `-name <storename>`

  Identifies the name of the store.

- `-timeout <timeout ms>`

  Specifies the store request timeout in milliseconds.

- `-consistency`

  Specifies the store request consistency. The default value is NONE_REQUIRED.

- `-durability`

  Specifies the store request durability. The default value is COMMIT_SYNC.

- `-username <user>`

  Specifies a username to log on as in a secure deployment.

- `-security <security-file-path>`

  In a secured deployment, specifies a path to the security file.

delete

Encapsulates commands that delete key/value pairs from store or rows from table. The subcommands are as follows:

- delete kv
- delete table

delete kv

```
delete kv [-key <key>] [-start prefixString] [-end prefixString] [-all]
```

Deletes one or more keys. If `-all` is specified, delete all keys starting at the specified key. If no key is specified, delete all keys in the store. The `-start` and `-end` flags can be used for restricting the range used for deleting.

For example, to delete all keys in the store starting at root:

```
kv -> delete kv -all
301 Keys deleted starting at root
```

delete table

```
kv-> delete table -name <table_name>
 [-field <name> -value <value>]*
 [-field <name> [-start <value>] [-end <value>]]
 [-ancestor <name>]* [-child <child_name>]*
 [-json <string>] [-delete-all]
```

Deletes one or multiple rows from the named table.

- `-name`
  Identifies a table name, which can be any of the following:
  - *`table_name`* – The target table is a top level table created in the default namespace, `sysdefault`. The default namespace (`sysdefault:`) prefix is not required to identify such tables.
  - *`table_name.child_name`* – The target table is the child of a parent table. Identify the child table by preceding it with the parent *`table_name`*, followed by a period (.) separator before *`child_name`*.
  - *`namespace_name:table_name`* – The target table was not created in the default (`sysdefault`) namespace. Identify *`table_name`* by preceding it with its *`namespace_name`*, followed by a colon (:).
  - *`namespace_name:table_name.child_name`* – The target table is the child of a parent table that was created in a namespace. Identify *`child_name`* by preceding it with both *`namespace_name:`* and the parent *`table_name`*, , followed by a period (.) separator.
- `-field` and `-value`

  Pairs specify the field values of the primary key or, use an empty key to delete all rows from the table.
- `-field` , `-start`, and `-end`

  Use these flags to restrict the sub-range for deletion associated with the parent key.
- `-ancestor` and `-child`

  Use to delete rows from a specific ancestor or descendant tables, in addition to the target table.

- `-json`

  Indicates that the key field values are in JSON format.

- `-delete-all`

  Indicates to delete all rows in a table.

execute

```
execute <statement> [-json] [-wait]
```

Oracle NoSQL Database provides a way to run Data Definition Language (DDL) statements used to form table and index statements. Using the `execute` command runs each statement you specify synchronously. You must enclose each DDL statement in single or double quotes. You must connect to a database store before using the execute command.

> **Note:**
>
> All DDL commands from the Admin CLI, including `execute`, are deprecated. Use the SQL for Oracle NoSQL Database Shell to execute this command. For more information, see Appendix A Introduction to the SQL for Oracle NoSQL Database Shell.

For example:

```
kv-> plan execute -id 19 -json -wait
{
    "operation" : "plan deploy-zone -name zn6 -rf 1 -type PRIMARY -no-
arbiters -no-master-affinity",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "id" : 19,
        "name" : "Deploy Zone",
        "isDone" : true,
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 09:35:31 UTC",
        "interrupted" : null,
        "end" : "2017-09-28 09:35:31 UTC",
        "error" : null,
        "executionDetails" : {
            "taskCounts" : {
            "total" : 1,
            "successful" : 1,
            "failed" : 0,
            "interrupted" : 0,
            "incomplete" : 0,
            "notStarted" : 0
            },
        "finished" : [ {
            "taskNum" : 1,
```

```
            "name" : "Plan 19 [Deploy Zone] task [DeployDatacenter
zone=zn6]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 09:35:31 UTC",
            "end" : "2017-09-28 09:35:31 UTC"
            } ],
        "running" : [ ],
        "pending" : [ ]
        },
        "planId" : 19,
        "zoneName" : "zn6",
        "zoneId" : "zn4",
        "type" : "PRIMARY",
        "rf" : 1,
        "allowArbiters" : false,
        "masterAffinity" : false
        }
    }
```

exit

```
exit | quit
```

Exits the interactive command shell.

get

Encapsulates commands that get key/value pairs from store or get rows from table. The subcommands are as follows:

- get kv
- get table

get kv

```
get kv [-key <keyString>] [-file <output>] [-all] [-keyonly]
[-valueonly] [-start <prefixString>] [-end <prefixString>]
```

Perform a simple get operation using the specified key. The obtained value is printed out if it contains displayable characters, otherwise the bytes array is encoded using Base64 for display purposes. "[Base64]" is appended to indicate this transformation. The arguments for the get command are:

- `-key <keyString>`

  Indicates the full or the prefix key path to use. If `<keyString>` is a full key path, it returns a single value information. The format of this `get` command is: `get -key <keyString>`. If `<keyString>` is a prefix key path, it returns multiple key/value pairs. The format of this `get` command is: `get -key <keyString> -all`. Key can be composed of both major and minor key paths, or a major key path only. The <keyString> format is: "major-key-path/-/minor-key-path". Additionally, in the case of the prefix key path, a key can be composed of the prefix part of a major key path.

For example, with some sample keys in the KVStore:

```
/group/TC/-/user/bob
/group/TC/-/user/john
/group/TC/-/dep/IT
/group/SZ/-/user/steve
/group/SZ/-/user/diana
```

A get command with a key containing only the prefix part of the major key path results in:

```
kv -> get kv -key /group -all -keyonly
/group/TC/-/user/bob
/group/TC/-/user/john
/group/TC/-/dep/IT
/group/SZ/-/user/steve
/group/SZ/-/user/diana
```

A get command with a key containing a major key path results in:

```
kv -> get kv -key /group/TC -all -keyonly
/group/TC/-/user/bob
/group/TC/-/user/john
/group/TC/-/dep/IT
```

Get commands with a key containing major and minor key paths results in:

```
kv -> get kv -key /group/TC/-/user -all -keyonly
/group/TC/-/user/bob
/group/TC/-/user/john
kv -> get kv -key /group/TC/-/user/bob
{
    "name"  : "bob.smith",
    "age"   : 20,
    "email" : "bob.smith@example.com",
    "phone" : "408 555 5555"
}
```

- `-file <output>`

  Specifies an output file, which is truncated, replacing all existing content with new content.

  In the following example, records from the key `/Smith/Bob` are written to the file `"data.out"`.

  ```
  kv -> get kv -key /Smith/Bob -all -file ./data.out
  ```

  In the following example, contents of the file `"data.out"` are replaced with records from the key `/Wong/Bill`.

  ```
  kv -> get kv -key /Wong/Bill -all -file ./data.out
  ```

- `-all`

  Specified for iteration starting at the specified key. If the key argument is not specified, the entire store will be iterated.

- `-keyonly`

  Specified with `-all` to return only keys.

- `-valueonly`

  Specified with `-all` to return only values.

- `-start <prefixString>` and `-end <prefixString>`

  Restricts the range used for iteration. This is particularly helpful when getting a range of records based on a key component, such as a well-formatted string. Both the `-start` and `-end` arguments are inclusive.

  > **Note:**
  >
  > `-start` and `-end` only work on the key component specified by `-key <keyString>`. The value of `<keyString>` should be composed of simple strings and cannot have multiple key components specified.

  For example, a log where its key structure is:

  ```
  /log/<year>/<month>/-/<day>/<time>
  ```

  puts all log entries for the same day in the same partition, but splits the days across shards. The `time` format is: "hour.minute".

  In this way, you can do a `get` of all log entries in February and March, 2013 by specifying:

  ```
  kv-> get kv -all -keyonly -key /log/2013 -start 02 -end 03
  /log/2013/02/-/01/1.45
  /log/2013/02/-/05/3.15
  /log/2013/02/-/15/10.15
  /log/2013/02/-/20/6.30
  /log/2013/02/-/28/8.10
  /log/2013/03/-/01/11.13
  /log/2013/03/-/15/2.28
  /log/2013/03/-/22/4.52
  /log/2013/03/-/31/11.55
  ```

  You can be more specific to the `get` command by specifying a more complete key path. For example, to display all log entries from April 1st to April 4th:

  ```
  kv-> get kv -all -keyonly -key /log/2013/04 -start 01 -end 04
  /log/2013/04/-/01/1.03
  /log/2013/04/-/01/4.05
  /log/2013/04/-/02/7.22
  /log/2013/04/-/02/9.40
  /log/2013/04/-/03/4.15
  ```

```
/log/2013/04/-/03/6.30
/log/2013/04/-/03/10.25
/log/2013/04/-/04/4.10
/log/2013/04/-/04/8.35
```

See the subcommand get table

get table

```
kv-> get table -name <table_name> [-index <name>]
    [-field <name> -value <value>]+
    [-field <name> [-start <value>] [-end <value>]]
    [-ancestor <name>]+ [-child <name>]+
    [-json <string>] [-file <output>] [-keyonly]
    [-pretty] [-report-size]
```

Identifies a table name, which can be any of the following:

- `-name`
  Identifies any of the following tables:
  - *`table_name`* – The target table is a top level table created in the default namespace, `sysdefault`. The default namespace (`sysdefault:`) prefix is not required to identify such tables.
  - *`table_name.child_name`* – The target table is the child of a parent table. Identify the child table by preceding it with the parent *`table_name`*, followed by a period (.) separator before *`child_name`*.
  - *`namespace_name:table_name`* – The target table was not created in the default (`sysdefault`) namespace. Identify *`table_name`* by preceding it with its *`namespace_name`*, followed by a colon (:).
  - *`namespace_name:table_name.child_name`* – The target table is the child of a parent table that was created in a namespace. Identify *`child_name`* by preceding it with both *`namespace_name`*: and the parent *`table_name`*, , followed by a period (.) separator.

  `-field` and `-value` pairs are used to specify the field values of the primary key or index key if using an index, specified by `-index`, or with an empty key to iterate the entire table.
- `-field` flag, along with its `-start` and `-end` flags, can be used to define a value range for the last field specified.
- `-ancestor` and `-child` flags are used to return results from specific ancestor and/or descendant tables as well as the target table.
- `-json` indicates that the key field values are in JSON format.
- `-file` is used to specify an output file, which is truncated.
- `-keyonly` is used to restrict information to keys only.
- `-pretty` is used for a nicely formatted JSON string with indentation and carriage returns.

- `-report-size` is used to show key and data size information for primary keys, data values, and index keys for matching records. When `-report-size` is specified no data is displayed.

help

```
help [command [sub-command]] [-include-deprecated]
```

Prints help messages. With no arguments the top-level shell commands are listed. With additional commands and sub-commands, additional detail is provided.

```
kv-> help load
Usage: load -file <path to file>
    Load the named file and interpret its contents as a script of
    commands to be executed.  If any command in the script fails
    execution will end.
```

Use `-include-deprecated` to show deprecated commands.

For example:

```
kv-> help show -include-deprecated
Encapsulates commands that display the state of the store and its
components.
Usage: show admins |
            datacenters |
            events |
            faults |
            indexes |
            parameters |
            perf |
            plans |
            pools |
            schemas |
            snapshots |
            tables |
            topology |
            upgrade-order |
            users |
            versions |
            zones
```

hidden

```
hidden [on|off]
```

Toggles visibility and setting of parameters that are normally hidden. Use these parameters only if advised to do so by Oracle Support.

history

```
history [-last <n>] [-from <n>] [-to <n>]
```

Displays command history. By default all history is displayed. Optional flags are used to choose ranges for display.

load

```
load -file <path to file>
```

Loads the named file and interpret its contents as a script of commands to be executed. If any of the commands in the script fail, execution will stop at that point.

For example, users of the Table API can use the load command to define a table and insert data using a single script. Suppose you have a table defined like this:

```
create table IF NOT EXISTS Users (
  id integer,
  firstname string,
  lastname string,
  age integer,
  income integer,
  primary key (id)
);
```

Then sample data for that table can be defined using JSON like this:

```
{
"id":1,
"firstname":"David",
"lastname":"Morrison",
"age":25,
"income":100000
}
{
"id":2,
"firstname":"John",
"lastname":"Anderson",
"age":35,
"income":100000
}
{
"id":3,
"firstname":"John",
"lastname":"Morgan",
"age":38,
"income":200000
}
{
"id":4,
```

```
"firstname":"Peter",
"lastname":"Smith",
"age":38,
"income":80000
}
{
"id":5,
"firstname":"Dana",
"lastname":"Scully",
"age":47,
"income":400000
}
```

Assume that the sample data is contained in a file called `Users.json`. Then you can define the table and load the sample data using a script that looks like this (file name `loadTable.txt`) :

```
### Begin Script ###
execute "create table IF NOT EXISTS Users ( \
  id integer, \
  firstname string, \
  lastname string, \
  age integer, \
  income integer, \
  primary key (id) \
)"

put table -name Users -file users.json
```

Then, the script can be run by using the `load` command:

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar runadmin -host node01 -port 5000 \
-security \
KVROOT/securtiy/client.security \
-store mystore
kv-> load -file ./loadTable.txt
Statement completed successfully
Loaded 5 rows to Users

kv->
```

If you are using the Key/Value API, first you create schema in the store:

```
{
     "type": "record",
     "name": "ContactInfo",
     "namespace": "example",
     "fields": [
         {"name": "phone", "type": "string", "default": ""},
         {"name": "email", "type": "string", "default": ""},
         {"name": "city", "type": "string", "default": ""}
```

```
     ]
}
```

Then you can collect the following commands in the script file `load-contacts-5.txt`:

```
### Begin Script ###
put -key /contact/Bob/Walker -value "{\"phone\":\"857-431-9361\", \
\"email\":\"Nunc@Quisque.com\",\"city\":\"Turriff\"}" \
-json example.ContactInfo
put -key /contact/Craig/Cohen -value "{\"phone\":\"657-486-0535\", \
\"email\":\"sagittis@metalcorp.net\",\"city\":\"Hamoir\"}" \
-json example.ContactInfo
put -key /contact/Lacey/Benjamin -value "{\"phone\":\"556-975-3364\", \
\"email\":\"Duis@laceyassociates.ca\",\"city\":\"Wasseiges\"}" \
-json example.ContactInfo
put -key /contact/Preston/Church -value "{\"phone\":\"436-396-9213\", \
\"email\":\"preston@mauris.ca\",\"city\":\"Helmsdale\"}" \
-json example.ContactInfo
put -key /contact/Evan/Houston -value "{\"phone\":\"028-781-1457\", \
\"email\":\"evan@texfoundation.org\",\"city\":\"Geest-G\"}" \
-json example.ContactInfo
exit
### End Script ###
```

The script can be run by using the `load` command:

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar runadmin -host node01 -port 5000 \
-security \
KVROOT/securtiy/client.security \
-store mystore
kv-> load -file ./load-contacts-5.txt
Operation successful, record inserted.
Operation successful, record inserted.
Operation successful, record inserted.
Operation successful, record inserted.
Operation successful, record inserted.
```

For more information on using the load command, see Create a script to configure the data store.

logtail

Monitors the store-wide log file until interrupted by an "enter" key press.

namespace

```
namespace [namespace_name]
```

Sets *namespace_name* as the default namespace for table operations and queries. For example:

```
kv-> namespace ns1
Namespace is ns1
```

Entering the command without *namespace_name* returns to the default namespace:

```
kv-> namespace
Namespace is sysdefault
```

page

```
page [on|<n>|off]
```

Turns query output paging on or off. If specified, `n` is used as the page height.

If `n` is 0, or "on" is specified, the default page height is used. Setting `n` to "off" turns paging off.

ping

```
ping [-json] [-shard <shardId>]
```

The `ping` and `verify` commands return information about the runtime entities of a data store. The command accesses components and Admin services available from the topology, returning information about the state of various components.

- `-json`

  Displays output in JSON format.

- `-shard <shardId>`

  Displays a subset of status information about the specific shard ID you supply.

Here is a basic example of calling ping from the Admin CLI:

```
kv-> ping
Pinging components of store mystore based upon topology sequence #308
300 partitions and 3 storage nodes
Time: 2019-01-03 20:19:27 UTC   Version: 19.1.0
Shard Status: healthy:1 writable-degraded:0 read-only:0 offline:0 total:1
Admin Status: healthy
Zone [name=1 id=zn1 type=PRIMARY allowArbiters=false masterAffinity=false]
RN Status: online:3 read-only:0 offline:0
maxDelayMillis:0 maxCatchupTimeSecs:0
Storage Node [sn1] on localhost:13230
Zone: [name=1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 19.1.0 2019-01-03 08:17:52 UTC  Build id:
12641466031c Edition: Enterprise
Admin [admin1]        Status: RUNNING,MASTER
Rep Node [rg1-rn1]    Status: RUNNING,MASTER sequenceNumber:633 haPort:13233
```

```
available storage size:109 GB
Storage Node [sn2] on localhost:13240
Zone: [name=1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING   Ver: 19.1.0 2019-01-03 08:17:52 UTC  Build id:
12641466031c Edition: Enterprise
Admin [admin2]          Status: RUNNING,REPLICA
    Rep Node [rg1-rn2]     Status: RUNNING,REPLICA sequenceNumber:633
haPort:13243 available storage size:109 GB delayMillis:0
catchupTimeSecs:0
Storage Node [sn3] on localhost:13250    Zone: [name=1 id=zn1
type=PRIMARY allowArbiters=false masterAffinity=false]    Status:
RUNNING   Ver: 19.1.0 2019-01-03 08:17:52 UTC  Build id: 12641466031c
Edition: Enterprise
    Admin [admin3]      Status: RUNNING,REPLICA
    Rep Node [rg1-rn3]     Status: RUNNING,REPLICA sequenceNumber:633
haPort:13253 available storage size:109 GB delayMillis:0
catchupTimeSecs:0
```

About Shard and Admin Status

After running a `ping` command, you should understand what is most useful (or troubling) about the system health. The most important content is the *Shard Status* entry. The following `ping` output details indicate one shard (`total:1`) that is healthy (`healthy:1`). All of the status types you'd prefer not to see (`writable-degraded`, `read-only`, and `offline` are zero (`0`), indicating nothing has one of those states. Everything is good.

```
Shard Status: healthy:1
writable-degraded:0
read-only:0
offline:0
total:1
```

What exactly does a healthy shard indicate? A healthy shard is one with all of its RNs running. Thus, if all shards in the topology are healthy, then all RNs are running, and no failures exist. Why are RNs so important? Because they are the components that perform read and write data operations.

Checking the Admin nodes status is also useful. In this simple example, only one Admin shard exists, so there is a single result: `Admin Status: healthy`. Other possible states are: `writable-degraded`, `read-only`, or `offline`.

For both RN shards and admins, these are what each result indicates:

Result	Meaning
`healthy`	All nodes are running, and the system is fully operational.
`writable-degraded`	A majority of the nodes are running. All operations are supported, but a minority of the nodes are offline or don't support writes. If you are using RF=3, this state is one step closer to being unable to support all operations. For example, with one node offline, losing another node means quorum will be lost, and the shard becomes read-only. Most people use RF=3, so this is typically what writable-degraded means.

Result	Meaning
`read-only`	Only a minority of the nodes are running. Read operations are supported, but write operations are not.
`offline`	No nodes are running, so no operations are supported.

About Zone Status

The next information from **ping** is about zones:

```
Zone [name=1 id=zn1
type=PRIMARY
allowArbiters=false
masterAffinity=false]
RN Status: online:3 read-only:0 offline:0
maxDelayMillis:0
maxCatchupTimeSecs:0
```

For stores with multiple zones, this information provides the status of nodes in different locations. For example, if a store was deployed using three zones, with the machines for each zone in a separate building, this information gives a quick summary status for machines in each building. In this simple example, there is only one zone, so that status information is similar to that for the entire store. The `maxDelayMillis` and `maxCatchupTimeSecs` entries provide information about data replication to replicas located in the zone. In our example, both values are zero (0). However, having large numbers for these entries could suggest that there are hardware problems with the machines in the zone, or problems with the network that connects that zone to other zones. Such information would be used only for more detailed debugging.

About Storage Nodes

Next, there is information about the nodes associated with a particular storage node:

```
Storage Node [sn1] on localhost:13230 Zone:
[name=1 id=zn1 type=PRIMARY allowArbiters=false masterAffinity=false]
Status: RUNNING Ver: 19.1.0 2019-01-03 08:17:52 UTC Build id: 12641466031c
Edition: Enterprise
Admin [admin1] Status: RUNNING,MASTER
Rep Node [rg1-rn1] Status: RUNNING,MASTER
sequenceNumber:633 haPort:13233 available storage size:109 GB
```

The `Status:` entry for the SN can have several possible values:

Status	Description
`STARTING`	The storage node is starting up.
`WAITING_FOR_DEPLOY`	The storage node is running but is waiting to be deployed in a new store.
`RUNNING`	The storage node is running -- this is the usual state.
`STOPPING`	The storage node is in the process of stopping, but is not yet in a `STOPPED` status.
`STOPPED`	The storage node is stopped.

Status	Description
UNREACHABLE	The storage node is not reachable, either because the SN service is down, the host machine is offline, or the machine is not reachable over the network.

About RNs and Admins on the Storage Node

The next entries provide status information about RNs and any Admin processes that are running on the storage node. Not all storage nodes have admin nodes. The number of RNs running on the storage node depends on the SN capacity.

```
Admin [admin1] Status: RUNNING,MASTER Rep Node [rg1-rn1]
Status: RUNNING,MASTER sequenceNumber:633 haPort:13233 available
storage size:109 GB
```

The `Status:` entry for both admin nodes and RNs, can have the following values:

Status	Description
STARTING	The node is starting up.
RUNNING,MASTER	The node is up and is the master. The master is in contact with a majority of nodes in the shard, and can perform writes requiring acknowledgment. This is the first of two normal states.
RUNNING,REPLICA	The node is up and is a replica. This is the second of two normal states.
RUNNING,MASTER (non-authoritative)	The node is up and is the master, but is not in contact with a majority of nodes in the shard. A non-authoritative master can perform only writes that do not require acknowledgment.
STOPPING	The node is stopping.
UNREACHABLE	The node could not be contacted over the network. The node is either stopped, failed, or there is a problem with the network connection to the machine.
Additional status values that can be appended to the status line to provide more information:	
readonly requests enabled	The node is running in read-only mode because the `plan enable-requests` command was run to set the node into read-only user operations mode.
requests disabled	The node is running with all user operation requests disabled, because the `plan enable-requests` command was run to disable all requests on the node. The `plan enable-requests` command disables requests on a per-shard basis, so it will prevent writes or all operations on all data in the shard.

While not shown in the initial example, the `ping` and `verify` commands can display one of the following states for RNs and shards. The table describes their effects and outcomes:

Displayed State	Effects	Outcome
`Unknown`	Masters go down.	Represents the read-only state of the RNs and shards still running. Currently, we do not support read-only status for any RN.
`Non-Authoritative Master`	Replica nodes go down.	After Replica nodes are down, remaining RNs and shards are in read-only mode. Currently, we do not support read-only status for any RN.
`Out of disk space`	Masters and replica nodes go down. Replicas are left in the `RUNNING, UNKNOWN` state, and the masters are in the `Non-Authoritative` state.	When masters and replica nodes go down, any remaining RNs and shards are in read-only mode. Currently, we do not support read-only status for any RN.
`Write requests disabled`	RNs and shard health are in read-only enabled request state.	RNs and shards are unable to accept any user requests, and are marked offline.

Both the `ping` and `verify` commands detect these states. Following is the output of a ping command on a shard (`rg2`), in a normal state, showing how results are returned:

```
kv-> ping -shard rg2
Pinging components of store mystore based upon topology sequence #2376
shard rg2 500 partitions and 3 storage nodes Time: 2018-09-28 07:06:46 UTC
Version: 18.3.2
Shard Status: healthy: Admin Status: healthy Zone [name=shardzone id=zn1
type=PRIMARY
allowArbiters=false masterAffinity=false]
RN Status: online:3 offline:0 maxDelayMillis:0 maxCatchupTimeSecs:0
Storage Node [sn10] on nodeA:5000    Zone: [name=shardzone id=zn1
type=PRIMARY
allowArbiters=false masterAffinity=false] Status: RUNNING   Ver: 18.3.2
2018-09-17 09:33:45 UTC
Build id: a72484b8b33c Edition: Enterprise
        Rep Node [rg2-rn1]
        Status: RUNNING,MASTER sequenceNumber:71,166 haPort:5010
        available storage size:8 GB Storage Node [sn11] on nodeB:5000
        Zone: [name=shardzone id=zn1 type=PRIMARY
        allowArbiters=false masterAffinity=false]  Status: RUNNING   Ver:
18.3.2 2018-09-17 09:33:45 UTC
        Build id: a72484b8b33c Edition: Enterprise
        Rep Node [rg2-rn2]
        Status: RUNNING,REPLICA sequenceNumber:71,166 haPort:5011
        available storage size:4 GB delayMillis:0 catchupTimeSecs:0
Storage Node [sn12] on nodeC:5000    Zone: [name=shardzone id=zn1
type=PRIMARY
allowArbiters=false masterAffinity=false] Status: RUNNING   Ver: 18.3.2
2018-09-17 09:33:45 UTC
Build id: a72484b8b33c Edition: Enterprise
```

```
        Rep Node [rg2-rn3]
        Status: RUNNING,REPLICA sequenceNumber:71,166 haPort:5012
        available storage size:6 GB delayMillis:0 catchupTimeSecs:0
```

Following are examples of return information when different states occur.

- Shard status becomes writable-degraded and is read-only:

```
kv-> ping
Pinging components of store concurrent plan store based upon topology
sequence #1082
 1000 partitions and 9 storage nodes
Time: 2018-11-06 05:12:36 UTC Version: 18.3.8
        Shard Status: healthy:2 writable-degraded:12 read-only:4
offline:0 total:18
Admin Status: healthy
Zone [name=dc1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online:30 read-only:24 offline:0 maxDelayMillis:0
maxCatchupTimeSecs:0
Storage Node [sn1] on slcao397:5000
Zone: [name=dc1 id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false] Status: RUNNING
Ver: 18.3.8 2018-10-26 11:36:43 UTC Build id: 6259xxxxxxxx Edition:
Enterprise
```

- RNs can have the `RUNNING,UNKNOWN` state for more than one reason, including reaching a disk limit, or when the RN is down:

```
Storage Node [sn4] on slcao400:5000 Zone: [name=dc1 id=zn1
type=PRIMARY
allowArbiters=false masterAffinity=false] Status: RUNNING Ver: 18.3.8
2018-10-26 11:36:43 UTC
Build id: 6259xxxxxxxx Edition: Enterprise
        Rep Node [rg7-rn1] Status: RUNNING,UNKNOWN
sequenceNumber:173,717,825 haPort:5020
        available storage size:-3 GB delayMillis:? catchupTimeSecs:?
        Rep Node [rg8-rn1] Status: RUNNING,UNKNOWN
sequenceNumber:173,555,937 haPort:5021
        available storage size:-3 GB delayMillis:? catchupTimeSecs:?
        Rep Node [rg9-rn1] Status: RUNNING,MASTER
sequenceNumber:173,697,007 haPort:5022 available storage size:-3 GB
        Rep Node [rg10-rn1] Status: RUNNING,UNKNOWN
sequenceNumber:173,293,747 haPort:5023
        available storage size:-3 GB delayMillis:? catchupTimeSecs:?
        Rep Node [rg11-rn1] Status: RUNNING,UNKNOWN
sequenceNumber:170,561,758 haPort:5024 available storage size:-3 GB
        delayMillis:? catchupTimeSecs:?
        Rep Node [rg12-rn1] Status: RUNNING,MASTER
sequenceNumber:170,410,483 haPort:5025 available storage size:-3 GB
```

- A running out of disk space error results in the master becoming `non-authoritative`:

```
 Storage Node [sn6] on slcao402:5000 Zone: [name=dc1 id=zn1 type=PRIMARY
allowArbiters=false
masterAffinity=false] Status: RUNNING Ver: 18.3.8 2018-10-26 11:36:43 UTC
Build id: 6259xxxxxxxx
Edition: Enterprise
Rep Node [rg7-rn3] Status: RUNNING,MASTER (non-authoritative)
sequenceNumber:173,754,579 haPort:5020 available storage size:45 GB
Rep Node [rg8-rn3] Status: RUNNING,REPLICA sequenceNumber:173,555,937
haPort:5021 available storage size:46 GB
delayMillis:0 catchupTimeSecs:0
Rep Node [rg9-rn3] Status: RUNNING,REPLICA
sequenceNumber:173,697,007 haPort:5022 available storage size:45 GB
delayMillis:0 catchupTimeSecs:0
Rep Node [rg10-rn3] Status: RUNNING,MASTER (non-authoritative)
sequenceNumber:173,293,747 haPort:5023 available storage size:45 GB
Rep Node [rg11-rn3] Status: RUNNING,REPLICA sequenceNumber:170,561,758
haPort:5024 available storage size:45 GB delayMillis:0 catchupTimeSecs:0
Rep Node [rg12-rn3] Status: RUNNING,REPLICA sequenceNumber:170,410,483
haPort:5025
available storage size:46 GB delayMillis:0 catchupTimeSecs:0
```

Finally, here is a basic example of calling `ping -json`:

```
kv-> ping -json
{
  "operation" : "ping",
  "returnCode" : 5000,
  "description" : "No errors found",
  "returnValue" : {
    "topology" : {
      "storeName" : "OurStore",
      "sequenceNumber" : 104,
      "numPartitions" : 100,
      "numStorageNodes" : 1,
      "time" : 1546801860520,
      "version" : "18.3.4"
    },
    "adminStatus" : "healthy",
    "shardStatus" : {
      "healthy" : 1,
      "writable-degraded" : 0,
      "read-only" : 0,
      "offline" : 0,
      "total" : 1
    },
    "zoneStatus" : [ {
      "resourceId" : "zn1",
      "name" : "OurZone",
      "type" : "PRIMARY",
      "allowArbiters" : false,
      "masterAffinity" : false,
      "rnSummaryStatus" : {
```

```
        "online" : 1,
        "offline" : 0,
        "read-only" : 0,
        "hasReplicas" : false
      }
    } ],
    "snStatus" : [ {
      "resourceId" : "sn1",
      "hostname" : "OurHost",
      "registryPort" : 5000,
      "zone" : {
        "resourceId" : "zn1",
        "name" : "OurZone",
        "type" : "PRIMARY",
        "allowArbiters" : false,
        "masterAffinity" : false
      },
      "serviceStatus" : "RUNNING",
      "version" : "18.4.0 2018-12-06 09:21:03 UTC  Build id:
fbfbd1541004 Edition: Enterprise",
      "adminStatus" : {
        "resourceId" : "admin1",
        "status" : "RUNNING",
        "state" : "MASTER",
        "authoritativeMaster" : true
      },
      "rnStatus" : [ {
        "resourceId" : "rg1-rn1",
        "status" : "RUNNING",
        "requestsEnabled" : "ALL",
        "state" : "MASTER",
        "authoritativeMaster" : true,
        "sequenceNumber" : 381,
        "haPort" : 5013,
        "availableStorageSize" : "97 GB"
      } ],
      "anStatus" : [ ]
    } ],
    "exitCode" : 0
  }
}
```


You can also access the `ping` utility through Admin utility tools, available in `kvtool.jar`. For more information see ping.

plan

Encapsulates operations, or jobs that modify store state. All subcommands with the exception of interrupt and wait change persistent state. Plans are asynchronous jobs so they return immediately unless `-wait` is used. Plan status can be checked using `show plans`. The optional arguments for all plans include:

- `-wait`

  Wait for the plan to complete before returning.

- `-plan-name`

  The name for a plan. These are not unique.

- `-noexecute`

  Do not execute the plan. If specified, the plan can be run later using `plan execute`.

- `-force`

  Used to force plan execution and plan retry.

- `-json | -json-v1`

  Displays the plan output as json or json-v1. The `-json` flag can be used to output in the new json format. The `-json-v1` flag can be used to output in the json-v1 format. If you have an existing script that relies on an older version of JSON output, you may want to consider using `-json-v1` flag so that your existing scripts continue to function.

The subcommands are as described below.

- plan add-index
- plan add-table
- plan cancel
- plan change-parameters
- plan change-storagedir
- plan change-user
- plan create-user
- plan deploy-admin
- plan deploy-datacenter
- plan deploy-sn
- plan deploy-topology
- plan deploy-zone
- plan deregister-es
- plan drop-user
- plan enable-requests
- plan evolve-table
- plan execute
- plan failover
- plan grant
- plan interrupt
- plan migrate-sn
- plan network-restore
- plan register-es
- plan remove-admin
- plan remove-datacenter

- plan remove-index
- plan remove-sn
- plan remove-table
- plan remove-zone
- plan repair-topology
- plan revoke
- plan start-service
- plan stop-service
- plan verify-data
- plan wait

plan add-index

```
plan add-index -name <name> -table <name> [-field <name>]*
    [-desc <description>]
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Adds an index to a table in the store.

where:

- `-name`

  Specifies the name of the index to add to a table.

- `-table`

  Specifies the table name where the index will be added. The table name is a dot-separated name with the format `tableName[.childTableName]*`.

- `-field`

  Specifies the field values of the primary key.

plan add-table

```
plan add-table -name <name>
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Adds a new table to the store. The table name is a dot-separated name with the format `tableName[.childTableName]*`.

Before adding a table, first use the `table create` command to create the named table. The following example defines a table (creates a table by name, adds fields and other table metadata).

```
## Enter into table creation mode
table create -name user -desc "A sample user table"
user->
user-> help
Usage: add-array-field |
```

```
add-field |
add-map-field |
add-record-field |
cancel |
exit |
primary-key |
remove-field |
set-description |
shard-key |
show
## Now add the fields
user-> help add-field
Usage: add-field -type <type> [-name <field-name> ] [-not-required]
[-nullable] [-default <value>] [-max <value>] [-min <value>]
[-max-exclusive] [-min-exclusive] [-desc <description>]
[-size <size>] [-enum-values <value[,value[,...]]
<type>: INTEGER, LONG, DOUBLE, FLOAT, STRING, BOOLEAN, DATE, BINARY, FIX
ED_BINARY, ENUM
## Adds a field. Ranges are inclusive with the exception of String,
## which will be set to exclusive.
user-> add-field -type Integer -name id
user-> add-field -type String -name firstName
user-> add-field -type String -name lastName
user-> help primary-key
Usage: primary-key -field <field-name> [-field <field-name>]*
## Sets primary key.
user-> primary-key -field id
## Exit table creation mode
user-> exit
## Table User built.
```

Use `table list -create` to see the list of tables that can be added. The following example lists and displays tables that are ready for deployment.

```
kv-> table list
## Tables to be added:
## User -- A sample user table
kv-> table list -name user
## Add table User:
{
  "type" : "table",
  "name" : "User",
  "id" : "User",
  "description" : "A sample user table",
  "shardKey" : [ "id" ],
  "primaryKey" : [ "id" ],
  "fields" : [ {
    "name" : "id",
    "type" : "INTEGER"
  }, {
    "name" : "firstName",
    "type" : "STRING"
  }, {
    "name" : "lastName",
```

```
    "type" : "STRING"
  } ]
}
```

The following example adds the table to the store.

```
## Add the table to the store.
kv-> help plan add-table
kv-> plan add-table -name user -wait
Executed plan 5, waiting for completion...
Plan 5 ended successfully
kv-> show tables -name user
{
  "type" : "table",
  "name" : "User",
  "id" : "r",
  "description" : "A sample user table",
  "shardKey" : [ "id" ],
  "primaryKey" : [ "id" ],
  "fields" : [ {
    "name" : "id",
    "type" : "INTEGER"
  }, {
    "name" : "firstName",
    "type" : "STRING"
  }, {
    "name" : "lastName",
    "type" : "STRING"
  } ]
  }
```

For more information and examples on table design, see Table Management in the *SQL Reference Guide*.

plan cancel

```
plan cancel -id <plan id> | -last - json
```

Cancels a plan that is not running. A running plan must be interrupted before it can be canceled.

Use `show plans` to list all plans that have been created along with their corresponding plan IDs and status.

Use the `-last` option to reference the most recently created plan.

```
kv-> plan cancel -id 23 -json
{
"operation" : "plan cancel|interrupt",
"returnCode" : 5000,
"description" : "Plan 23 was canceled",
```

```
"returnValue" : null
}
```

plan change-parameters

```
plan change-parameters -security | -service <id> |
    -all-rns [-zn <id> | -znname <name>] | -all-ans [-zn <id> |
    -znname <name>] | -all-admins [-zn <id> | -znname <name>]
    [-dry-run] [-plan-name <name>]
    [-json] [-wait] [-noexecute] [-force] -params [name=value]*
```

Changes parameters for either the specified service, or for all service instances of the same type that are deployed to the specified zone or all zones.

The `-security` flag allows changing store-wide global security parameters, and should never be used with other flags.

The `-service` flag allows a single instance to be affected; and should never be used with either the `-zn` or `-znname` flag.

The `-all-*` flags can be used to change all instances of the service type. The parameters to change follow the `-params` flag and are separated by spaces. The parameter values with embedded spaces must be quoted; for example, name="value with spaces".

One of the -all-* flags can be combined with the -zn or -znname flag to change all instances of the service type deployed to the specified zone; leaving unchanged, any instances of the specified type deployed to other zones. If one of the -all-* flags is used without also specifying the zone, then the desired parameter change will be applied to all instances of the specified type within the store, regardless of zone.

If `-dry-run` is specified, the new parameters are returned without changing them. Use the command `show parameters` to see what parameters can be modified. For more information, see show parameters.

For more information on changing parameters in the store, see Setting Store Parameters.

> **Note:**
>
> The `plan change-parameters` updates the store metadata database even if the component is not available. The component's configuration will be made consistent when the KVStore system detects an inconsistency.

```
kv-> plan change-parameters -service rg1-rn2 -json -wait -params
loggingConfigProps="oracle.kv.level=DEBUG"
{
    "operation" : "Change RepNode Params",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "id" : 20,
        "owner" : "root(id:u1)",
        "name" : "Change RepNode Params",
```

```
        "isDone" : true,
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 05:31:05 UTC",
        "interrupted" : null,
        "end" : "2017-09-28 05:31:10 UTC",
        "error" : null,
        "executionDetails" : {
            "taskCounts" : {
                "total" : 4,
                "successful" : 4,
                "failed" : 0,
                "interrupted" : 0,
                "incomplete" : 0,
                "notStarted" : 0
            },
        "finished" : [ {
            "taskNum" : 1,
            "name" : "Plan 20 [Change RepNode Params] task
[WriteNewParams rg1-rn2]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 05:31:05 UTC",
            "end" : "2017-09-28 05:31:06 UTC"
        }, {
            "taskNum" : 2,
            "name" : "Plan 20 [Change RepNode Params] task [StopNode
rg1-rn2]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 05:31:06 UTC",
            "end" : "2017-09-28 05:31:07 UTC"
        }, {
            "taskNum" : 3,
            "name" : "Plan 20 [Change RepNode Params] task
[StartNode]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 05:31:07 UTC",
            "end" : "2017-09-28 05:31:07 UTC"
        }, {
            "taskNum" : 4,
            "name" : "Plan 20 [Change RepNode Params] task
[WaitForNodeState rg1-rn2 to reach RUNNING]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 05:31:07 UTC",
            "end" : "2017-09-28 05:31:10 UTC"
        } ],
        "running" : [ ],
        "pending" : [ ]
        }
    }
}
```

plan change-storagedir

```
plan change-storagedir -sn <id> -storagedir <path> -add | -remove
    [-storagedirsize <size>] [-plan-name <name>] [-json] [-wait] [-
```

```
noexecute]
    [-force]
```

Adds or removes a storage directory on a Storage Node, for storing a Replication Node.

where:

- `-sn`

  Specifies the Storage Node where the storage directory is added or removed.

- `-storagedir`

  Specifies the path to the storage directory on a Storage Node for storing a Replication Node.

- `-add | -remove`

  Specifies to add (`-add`) the storage dir.

  Specifies to remove (`-remove`) the storage dir.

- `-storagedirsize`

  Specifies the size of the directory specified in `-storagedir`. This parameter is optional; however, it is an error to specify this parameter for some, but not all, storage directories.

  Use of this parameter is recommended for heterogeneous installation environments where some hardware has more storage capacity than other hardware. If this parameter is specified for all storage directories, then the store's topology will place more data on the shards that offer more storage space. If this parameter is not used, then data is spread evenly across all shards.

  The value specified for this parameter must be a long, optionally followed by a unit string. Accepted unit strings are: KB, MB, GB, and TB, corresponding to 1024, 1024^2, 1024^3, 1024^4 respectively. Acceptable strings are case insensitive. Valid delimiters between the long value and the unit string are " ", "-", or "_".

```
-storagedirsize 200 MB
-storagedirsize 4_tb
-storagedirsize 5000-Mb

kv-> plan change-storagedir -sn sn2 -storagedir /tmp/kvroot -add -json -
wait
{
"operation" : "Change Storage Node Params",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "id" : 21,
    "owner" : "root(id:u1)",
    "name" : "Change Storage Node Params",
    "isDone" : true,
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:33:14 UTC",
    "interrupted" : null,
    "end" : "2017-09-28 05:33:14 UTC",
    "error" : null,
    "executionDetails" : {
        "taskCounts" : {
```

```
            "total" : 1,
            "successful" : 1,
            "failed" : 0,
            "interrupted" : 0,
            "incomplete" : 0,
            "notStarted" : 0
        },
        "finished" : [ {
            "taskNum" : 1,
            "name" : "Plan 21 [Change Storage Node Params] task
    [WriteNewSNParams sn2]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 05:33:14 UTC",
            "end" : "2017-09-28 05:33:14 UTC"
        } ],
    "running" : [ ],
    "pending" : [ ]
            }
        }
    }
```

plan change-user

```
plan change-user -name <user name>
    [-disable | -enable] [-set-password [-password <new password>]
    [-retain-current-password]] [-clear-retained-password]
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Change a user with the specified name in the store. The -retain-current-password argument option causes the current password to be remembered during the -set-password operation as a valid alternate password for configured retention time or until cleared using -clear-retained-password. If a retained password has already been set for the user, setting password again will cause an error to be reported.

This command is deprecated. For more information see User Modification in the *Security Guide*.

plan create-user

```
plan create-user -name <user name>
    [-admin] [-disable] [-password <new password>]
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Create a user with the specified name in the store. The -admin argument indicates that the created user has full administrative privileges.

This command is deprecated. For more information, see User Creation in the *Security Guide*.

plan deploy-admin

```
plan deploy-admin -sn <id> [-plan-name <name>]
    [-wait] [-noexecute] [-force]
```

Deploys an Admin to the specified Storage Node. The admin type (PRIMARY/SECONDARY) is the same type as the zone the Storage Node is in.

For more information on deploying an admin, see Create an Administration Process on a Specific Storage Node.

```
kv-> plan deploy-admin -sn sn1 -json -wait
"operation" : "plan deploy-admin -sn 1",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "id" : 22,
    "owner" : "root(id:u1)",
    "name" : "Deploy Admin Service",
    "isDone" : true,
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:34:26 UTC",
    "interrupted" : null,
    "end" : "2017-09-28 05:34:27 UTC",
    "error" : null,
    "executionDetails" : {
        "taskCounts" : {
            "total" : 4,
            "successful" : 4,
            "failed" : 0,
            "interrupted" : 0,
            "incomplete" : 0,
            "notStarted" : 0
        },
    "finished" : [ {
            "taskNum" : 1,
            "name" : "Plan 22 [Deploy Admin Service] task [DeployAdmin
admin1 on sn1]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 05:34:26 UTC",
            "end" : "2017-09-28 05:34:27 UTC"
        }, {
            "taskNum" : 2,
            "name" : "Plan 22 [Deploy Admin Service] task [WaitForAdminState
admin1 to reach RUNNING]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 05:34:27 UTC",
            "end" : "2017-09-28 05:34:27 UTC"
        }, {
            "taskNum" : 3,
            "name" : "Plan 22 [Deploy Admin Service] task
[UpdateAdminHelperHost admin1]",
            "state" : "SUCCEEDED",
```

```
            "start" : "2017-09-28 05:34:27 UTC",
            "end" : "2017-09-28 05:34:27 UTC"
        }, {
            "taskNum" : 4,
            "name" : "Plan 22 [Deploy Admin Service] task
[NewAdminParameters refresh admin1 parameter state
            without restarting]",
            "state" : "SUCCEEDED",
            "start" : "2017-09-28 05:34:27 UTC",
            "end" : "2017-09-28 05:34:27 UTC"
        } ],
    "running" : [ ],
    "pending" : [ ]
    },
    "planId" : 22,
    "resourceId" : "admin1",
    "snId" : "sn1"
    }
}
```

plan deploy-datacenter

Deprecated. See plan deploy-zone instead.

plan deploy-sn

```
plan deploy-sn -zn <id> | -znname <name> -host <host> -port <port>
    [-plan-name <name>] [-json] [-wait] [-noexecute] [-force]
```

Deploys the Storage Node at the specified host and port into the specified zone.

where:

- `-sn`

  Specifies the Storage Node to deploy.

- `-zn <id> | -znname <name>`

  Specifies the Zone where the Storage Node is going to be deployed.

- `-host`

  Specifies the host name where the Storage Node is going to be deployed.

- `-port`

  Specifies the port number of the host.

For more information on deploying your Storage Nodes, see Create the Remainder of your Storage Nodes.

```
kv-> plan deploy-sn -zn 1 -json -host localhost -port 10000 -wait
{
"operation" : "plan deploy-sn -zn 1 -host localhost -port 10000",
"returnCode" : 5000,
"description" : "Operation ends successfully",
```

```
"returnValue" : {
    "id" : 25,
    "owner" : "root(id:u1)",
    "name" : "Deploy Storage Node",
    "isDone" : true,
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:40:50 UTC",
    "interrupted" : null,
    "end" : "2017-09-28 05:40:51 UTC",
    "error" : null,
    "executionDetails" : {
        "taskCounts" : {
            "total" : 1,
            "successful" : 1,
            "failed" : 0,
            "interrupted" : 0,
            "incomplete" : 0,
            "notStarted" : 0
        },
    "finished" : [ {
        "taskNum" : 1,
        "name" : "Plan 25 [Deploy Storage Node] task [DeploySN
sn4(localhost:10000)]",
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 05:40:50 UTC",
        "end" : "2017-09-28 05:40:51 UTC"
        } ],
    "running" : [ ],
    "pending" : [ ]
    },
    "planId" : 25,
    "resourceId" : "sn4",
    "zoneId" : "zn1",
    "host" : "localhost",
    "port" : 10000
    }
}
```

plan deploy-topology

```
plan deploy-topology -name <topology name> [-plan-name <name>]
    [-json] [-wait] [-noexecute] [-force]
```

Deploys the specified topology to the store. The KVStore size determines how long the command takes to deploy replication and arbiter nodes to become fully functional shard members. The `plan deploy-topology` command does not wait for this command to finish.

After running the `plan deploy-topology` command, use the `verify configuration` command to check the running state of the components in the topology. See Deploy the Topology Candidate.

```
kv-> plan deploy-topology -name MyStoreLayout -json -wait
{
```

```
"operation" : "plan deploy-topology -name MyStoreLayout",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "id" : 26,
    "owner" : "root(id:u1)",
    "name" : "Deploy Topo",
    "isDone" : true,
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:56:25 UTC",
    "interrupted" : null,
    "end" : "2017-09-28 05:56:26 UTC",
    "error" : null,
    "executionDetails" : {
    "taskCounts" : {
    "total" : 6,
    "successful" : 6,
    "failed" : 0,
    "interrupted" : 0,
    "incomplete" : 0,
    "notStarted" : 0
    },
"finished" : [ {
    "taskNum" : 1,
    "name" : "Plan 26 [Deploy Topo] task [UpdateDatacenterV2
zone=zn1]",
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:56:25 UTC",
    "end" : "2017-09-28 05:56:25 UTC"
    }, {
    "taskNum" : 2,
    "name" : "Plan 26 [Deploy Topo] task [UpdateDatacenterV2
zone=zn2]",
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:56:25 UTC",
    "end" : "2017-09-28 05:56:25 UTC"
    }, {
    "taskNum" : 3,
    "name" : "Plan 26 [Deploy Topo] task [UpdateDatacenterV2
zone=zn3]",
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:56:25 UTC",
    "end" : "2017-09-28 05:56:25 UTC"
    }, {
    "taskNum" : 4,
    "name" : "Plan 26 [Deploy Topo] task [BroadcastTopo]",
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:56:25 UTC",
    "end" : "2017-09-28 05:56:26 UTC"
    }, {
    "taskNum" : 5,
    "name" : "Plan 26 [Deploy Topo] task [BroadcastMetadata]",
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:56:26 UTC",
    "end" : "2017-09-28 05:56:26 UTC"
```

```
    }, {
    "taskNum" : 6,
    "name" : "Plan 26 [Deploy Topo] task [BroadcastTopo]",
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 05:56:26 UTC",
    "end" : "2017-09-28 05:56:26 UTC"
    } ],
    "running" : [ ],
    "pending" : [ ]
    },
    "planId" : 26,
    "topoName" : "MyStoreLayout"
    }
}
```

plan deploy-zone

```
plan deploy-zone -name <zone name>
    -rf <replication factor>
    [-type [primary | secondary]]
    [-arbiters | -no-arbiters]
    [-json ]
    [-master-affinity | -no-master-affinity]
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Deploys the specified zone to the store and creates a primary zone if you do not specify a `-type`.

where:

- `-name`

  Specifies the name of the zone to deploy.

- `-rf`

  Specifies the replication factor of the zone.

- `-type`

  Specifies the type of the zone to deploy. It can be a primary or a secondary zone. If -type is not specified, a primary zone is deployed.

- `-json`

  Formats the command output in JSON.

- `-arbiters | -no-arbiters`

  If you specify `-arbiters`, you can allocate Arbiter Nodes on the Storage Node in the zone. You can specify this flag only on a primary zone.

  Specifying `-no-arbiters` precludes allocating Arbiter Nodes on the Storage Node in the zone.
  The default value is `-no-arbiters`.

- `-master–affinity | -no-master-affinity`

  Specifying `-master-affinity` indicates that this zone can host a master.

Specifying `-no-master-affinity` indicates that this zone cannot host a master. The default value is `-no-master-affinity`.

For more information on creating a zone, see Create a Zone.

```
kv-> plan deploy-zone -name zn6 -rf 1 -json -wait
{
    "operation" : "plan deploy-zone -name zn6 -rf 1 -type PRIMARY -no-
arbiters -no-master-affinity",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "id" : 27,
        "owner" : "root(id:u1)",
        "name" : "Deploy Zone",
        "isDone" : true,
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 05:57:29 UTC",
        "interrupted" : null,
        "end" : "2017-09-28 05:57:29 UTC",
        "error" : null,
        "executionDetails" : {
            "taskCounts" : {
            "total" : 1,
            "successful" : 1,
            "failed" : 0,
            "interrupted" : 0,
            "incomplete" : 0,
            "notStarted" : 0
        },
    "finished" : [ {
        "taskNum" : 1,
        "name" : "Plan 27 [Deploy Zone] task [DeployDatacenter
zone=zn6]",
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 05:57:29 UTC",
        "end" : "2017-09-28 05:57:29 UTC"
        } ],
        "running" : [ ],
        "pending" : [ ]
        },
    "planId" : 27,
    "zoneName" : "zn6",
    "zoneId" : "zn4",
    "type" : "PRIMARY",
    "rf" : 1,
    "allowArbiters" : false,
    "masterAffinity" : false
    }
}
```

plan deregister-es

```
plan deregister-es
```

Deregisters the Elasticsearch cluster from the Oracle NoSQL Database store, using the `deregister-es` plan command. This is allowed only if all full text indexes are first removed using the `plan remove-index` command, see plan remove-index.

For example:

```
kv-> plan deregister-es
Cannot deregister ES because these text indexes exist:
mytestIndex
JokeIndex
```

For more information, see Integration with Elastic Search for Full Text Search in the *Integrations Guide*.

plan drop-user

```
plan drop-user -name <user name>
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Drop a user with the specified name in the store. A logged-in user may not drop itself.

This command is deprecated. For more information, see User Removal in the *Security Guide*.

plan enable-requests

This command will change the type of user requests supported by a set of shards or the entire store.

```
plan enable-requests
    -request-type {all|readonly|none}
    {-shards <shardId[,shardId]*> | -store}
    [-plan-name <name>] [-wait]
    [-noexecute] [-force]
    [-json|-json-v1]
```

Limit the type of requests enabled for specific shards or the whole store.

The `-request-type` flag configures the read and write requests. The following request types can be configured by this command.

- **all** means the store or shards can process both read and write requests;
- **readonly** makes the store or shards only respond to read requests;
- **none** means no read or write requests will be processed by the store or shards.

The `-shards` flag specifies the list of shards that should be configured, if you want the configuration to be done on one or more shards. You can get details about the `shardid` by

executing the `show topology` command. The `rgXX` portion in the show topology output denotes the `shardid`. See show topology.

The `-store` flag specifies that the configuration to be done on the entire store.

You should specify either the `-shard` flag or the `-store` flag.

Example 7-1 plan enable-requests

For example, If you want to put the shard rg1 in readonly mode, you would specify `rg1` as the `shardid` and `readonly` as the `request-type`.

```
kv-> plan enable-requests
    -request-type readonly -shards rg1
Started plan 25. Use show plan -id 25 to check status.
    To wait for completion, use plan wait -id 25
```

Example 7-2 plan enable-requests

For example, If you want to put the whole store in readonly mode and to get the output in json format, you would specify the `store` attribute, `request-type` attribute as `readonly` and `json` attribute.

```
kv-> plan enable-requests
    -request-type readonly -store -json
{
  "operation" : "plan enable-requests",
  "returnCode" : 5000,
  "description" : "Operation ends successfully",
  "returnValue" : {
    "planId" : 26
  }
}
```

Example 7-3 plan enable-requests

For example, If you want to put the whole store in readonly mode and to get the output in json v1 format, you would specify the `store` attribute, `request-type` attribute as `readonly` and `json-v1` attribute.

```
kv-> plan enable-requests
    -request-type readonly -store -json-v1
{
  "operation" : "plan enable-requests",
  "return_code" : 5000,
  "description" : "Operation ends successfully",
  "return_value" : {
    "plan_id" : 27
  }
}
```

plan evolve-table

```
plan evolve-table -name <name>
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Evolves a table in the store. The table name is a dot-separate with the format `tableName[.childTableName]*`.

Use the `table evolve` command to evolve the named table. The following example evolves a table.

```
## Enter into table evolution mode
kv-> table evolve -name User
kv-> show
{
  "type" : "table",
  "name" : "User",
  "id" : "r",
  "description" : "A sample user table",
  "shardKey" : [ "id" ],
  "primaryKey" : [ "id" ],
  "fields" : [ {
    "name" : "id",
    "type" : "INTEGER"
  }, {
    "name" : "firstName",
    "type" : "STRING"
  }, {
    "name" : "lastName",
    "type" : "STRING"
  } ]
}
## Add a field
kv-> add-field -type String -name address
## Exit table creation mode
kv-> exit
## Table User built.
kv-> plan evolve-table -name User -wait
## Executed plan 6, waiting for completion...
## Plan 6 ended successfully
kv-> show tables -name User
{
  "type" : "table",
  "name" : "User",
  "id" : "r",
  "description" : "A sample user table",
  "shardKey" : [ "id" ],
  "primaryKey" : [ "id" ],
  "fields" : [ {
    "name" : "id",
    "type" : "INTEGER"
  }, {
    "name" : "firstName",
```

```
    "type" : "STRING"
  }, {
    "name" : "lastName",
    "type" : "STRING"
  }, {
    "name" : "address",
    "type" : "STRING"
  } ]
}
```

Use `table list -evolve` to see the list of tables that can be evolved. For more information, see plan add-table .

plan execute

```
plan execute -id <id> | -last
    [-plan-name <name>] [-json] [-wait] [-noexecute] [-force]
```

Executes an existing plan that has not yet been executed. The plan must have been previously created using the `-noexecute` flag .

Use the `-last` option to reference the most recently created plan.

```
kv-> plan execute -id 19 -json -wait
{
    "operation" : "plan deploy-zone -name zn6 -rf 1 -type PRIMARY -no-
arbiters -no-master-affinity",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "id" : 19,
        "name" : "Deploy Zone",
        "isDone" : true,
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 09:35:31 UTC",
        "interrupted" : null,
        "end" : "2017-09-28 09:35:31 UTC",
        "error" : null,
        "executionDetails" : {
            "taskCounts" : {
            "total" : 1,
            "successful" : 1,
            "failed" : 0,
            "interrupted" : 0,
            "incomplete" : 0,
            "notStarted" : 0
        },
    "finished" : [ {
        "taskNum" : 1,
        "name" : "Plan 19 [Deploy Zone] task [DeployDatacenter
zone=zn6]",
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 09:35:31 UTC",
```

```
        "end" : "2017-09-28 09:35:31 UTC"
} ],
"running" : [ ],
"pending" : [ ]
},
"planId" : 19,
"zoneName" : "zn6",
"zoneId" : "zn4",
"type" : "PRIMARY",
"rf" : 1,
"allowArbiters" : false,
"masterAffinity" : false
}
}
```

plan failover

```
plan failover  { [-zn <zone-id> | -znname <zone-name>]
    -type [primary | offline-secondary] }...
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

where:

- `-zn <zone-id> | -znname <zone-name>`

  Specifies a zone either by zone ID or by name.

- `-type [primary | offline-secondary]`

  Specifies the new type for the associated zone.

Changes zone types to failover to either Primary or Secondary zones, whenever a primary zone failure results in a loss of quorum. Arbiters will not be created or removed from the topology. This command can introduce violations if a zone that contains Arbiters is specified as secondary-offline. Use the force flag if arbiter violations are introduced.

Zones whose new type is primary are taking over from failed primary zones to reestablish quorum. For these zones, a quorum of storage nodes in each shard in the zone must be available and responding to requests.

Zones whose new type is offline-secondary represent primary zones that are currently offline, resulting in the current loss of quorum. For these zones, all of the storage nodes in the zones must currently be unavailable. No zone type changes can be performed if these requirements are not met when the command starts.

> **Note:**
>
> Arbiter nodes are not currently supported during failover and switchover operations.

To correct any violations after the topology components are repaired, the `plan failover` command executes a `rebalance` command. To successfully deploy the new topology after a `rebalance`, the Storage Nodes hosting topology components must be running. If a Storage

Node in a zone that failed over to a Secondary zone that contained an Arbiter, when the SN restarts, the Arbiter rejoins the shard.

You cannot execute this command when other plans are in progress for the data store. Before executing this plan, cancel or interrupt any other plans.

plan grant

```
plan grant [-role <role name>]* -user <user_name>
```

Allows granting roles to users.

where:

- `-role <role name>`

  Specifies the roles that will be granted. The role names should be the system-defined roles (except `public`) listed in the *Security Guide*.

- `-user <user_name>`

  Specifies the user who the role will be granted from.

This command is deprecated. For more information see Grant Roles or Privileges in the *Security Guide*.

plan interrupt

```
plan interrupt -id <plan id> | -last [-json]
```

Interrupts a running plan. An interrupted plan can only be re-executed or canceled. Use -last to reference the most recently created plan.

```
kv-> plan interrupt -id 20 -json
{
"operation" : "plan cancel|interrupt",
"returnCode" : 5000,
"description" : "Plan 20 was interrupted",
"returnValue" : null
}
```

plan migrate-sn

```
plan migrate-sn -from <id> -to <id>
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Migrates the services from one Storage Node to another. The old node must not be running.

where:

- `-from`

  Specifies the Storage Node (old) that you are migrating from.

- `-to`

  Specifies the Storage Node (new) that you are migrating to.

For example, assuming that you are migrating from Storage Node 25 to 26, you would use:

```
kv-> plan migrate-sn -from sn25 -to sn26
```

Before executing the `plan migrate-sn` command, you can stop any running old Storage Node by using `-java -Xmx64m -Xms64m -jar KVHOME/lib/kvstore.jar stop -root KVROOT`.

plan network-restore

```
plan network-restore -from <id> -to <id> -retain-logs
    [-plan-name <name>] [-wait] [-noexecute] [-force] [-json|-json-v1]
```

The `plan network-restore` command restores a replication node (RN) with updates that the RN missed after losing networking connectivity. Use this only if the RN cannot be restored through the automatic procedures described here.

When a replication node becomes disconnected for any reason, it misses updates that occur while it was not connected. Oracle NoSQL Database uses two ways to update the recovered RN after it comes back online.

One way occurs within the RN's replication group. When the recovered RN returns, the replication group's master node streams all missed updates from the time the time the RN became disconnected, to the time it resumed operations.

Another way to restore a reconnected RN is over a network connection. Performing a network restore copies a complete set of data log files (`*.jdb`) from a peer, supplying the recovered RN with a comprehensive data set. The content contains many intermediate changes that are not reflected in the current store contents. This is because the data log files (`*.jdb`), which the recipient RN ingests, contain all changes, including any intermediate ones.

Do not confuse the data `*.jdb` log files, which contain data store activities, with the debug log files (`*.log`), which are used for debugging purposes.

If neither of the automatic Oracle NoSQL Database RN repopulation attempts succeed, it can be due to unforeseen circumstances, or a catastrophic situation that destroys data on multiple hosts. In this case, you can execute `plan network-restore` manually from the Admin CLI. However, doing so requires you to specify the RN that will supply the updated data.

You can attempt a network restore using the `plan network-restore` command from the admin CLI:

```
kv-> plan network-restore -help
Usage: plan network-restore -from <id> -to <id> [-retain-logs] \
[-plan-name <name>] [-wait] [-noexecute] [-force] [-json | json-v1]
Network restore a RepNode from another one in their replication group.
```

where:

- `-from` flag – Specifies the Replication Node ID from the same replication group (matching rgX). The `-from` node must be fully up to date, and able to supply the `*.dbd` log files to the destination RN. For example, if the `-to` recipient RN ID is `rg1-rn3`, and the ping output shows that `rg1-rn2` is the master, then that ID (`rg1-rn2`) is a good choice for the `-from` value.
- `-to` flag – Specifies the ID (rgX-rnY) of the recipient RN.
- `-retain-logs` flag – Retains obsolete log files on the lagging replica. The system renames the files, rather than deleting them. It is generally unnecessary to use this flag, unless you suspect that log files are corrupted on the recovering RN.

plan register-es

```
plan register-es -clustername <name> -host <host>
    -port <transport port> [-force]
```

Registers the Elasticsearch cluster with the Oracle NoSQL Database store, using the `register-es` plan command. It is only necessary to register one node of the cluster, as the other nodes in the cluster will be found automatically.

where:

- `-clustername`

  Specifies the name of the Elasticsearch cluster.
- `-host`

  Specifies the host name of a node in the cluster.
- `-port`

  Specifies the transport port of a node in the cluster.

For more information, see Integration with Elastic Search for Full Text Searchin the *Integrations Guide*.

plan remove-admin

```
plan remove-admin -admin <id> | -zn <id> | -znname <name>
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Removes the desired Admin instances; either the single specified instance, or all instances deployed to the specified zone.

If you use the `-admin` flag and there are 3 or fewer Admins running in the store, or if you use the `-zn` or `-znname` flag and the removal of all Admins from the specified zone would result in only one or two Admins in the store, then the desired Admins will be removed only if you specify the `-force` flag.

Also, if you use the `-admin` flag and there is only one Admin in the store, or if you use the `-zn` or `-znname` flag and the removal of all Admins from the specified zone would result in the removal of all Admins from the store, then the desired Admins will not be removed.

plan remove-datacenter

```
plan remove-datacenter
```

This command is deprecated. See plan remove-zone instead.

plan remove-index

```
plan remove-index -name <name> -table <name>
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Removes an index from a table.

where:

- `-name`

  Specifies the name of the index to remove.

- `-table`

  Specifies the table name to remove the index from. The table name is a dot-separated name with the format tableName[.childTableName]*.

plan remove-sn

```
plan remove-sn -sn <id>
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Removes the specified Storage Node from the topology. The Storage Node is automatically stopped before removal.

This command is useful when removing unused, old Storage Nodes from the store. To do this, see Replacing a Failed Storage Node.

If the Storage Node is being removed as part of removing a secondary zone then,

- any replication nodes must first be removed using the `topology change-replication-factor` and `plan deploy-topology` commands, and
- any Admin Nodes must first be removed using `plan remove-admin` command.

plan remove-table

```
plan remove-table -name <name> [-keep-data]
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Removes a table from the store. The named table must exist and must not have any child tables. Indexes on the table are automatically removed. By default data stored in this table is also removed. Table data may be optionally saved by specifying the -keep-data flag. Depending on the indexes and amount of data stored in the table this may be a long-running plan.

The following example removes a table.

```
## Remove a table.
kv-> plan remove-table -name User
## Started plan 7. Use show plan -id 7 to check status.
## To wait for completion, use plan wait -id 7.
kv-> show tables
## No table found.
```

plan remove-zone

```
plan remove-zone -zn <id> | -znname <name>
    [-plan-name <name>] [-wait] [-noexecute] [-force]
```

Removes the specified zone from the store.

Before running this command, all Storage Nodes that belong to the specified zone must first be removed using the `plan remove-sn` command.

plan repair-topology

```
plan repair-topology
    [-plan-name <name>] [-wait] [-json] [-noexecute] [-force]
```

Inspects the store's deployed, current topology for inconsistencies in location metadata that may have arisen from the interruption or cancellation of previous deploy-topology or migrate-sn plans. Where possible, inconsistencies are repaired. This operation can take a while, depending on the size and state of the store.

```
kv-> plan repair-topology -json -wait
{
"operation" : "Repair Topology",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "id" : 25,
    "name" : "Repair Topology",
    "isDone" : true,
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 09:43:06 UTC",
    "interrupted" : null,
    "end" : "2017-09-28 09:43:06 UTC",
    "error" : null,
    "executionDetails" : {
        "taskCounts" : {
        "total" : 1,
        "successful" : 1,
        "failed" : 0,
        "interrupted" : 0,
        "incomplete" : 0,
        "notStarted" : 0
        },
```

```
    "finished" : [ {
        "taskNum" : 1,
        "name" : "Plan 25 [Repair Topology] task [VerifyAndRepair]",
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 09:43:06 UTC",
        "end" : "2017-09-28 09:43:06 UTC"
    } ],
    "running" : [ ],
    "pending" : [ ]
    }
    }
}
```

plan revoke

```
plan revoke [-role <role name>]* -user <user_name>
```

Allows revoking roles to users.

where:

- `-role <role name>`

  Specifies the roles that will be revoked. The role names should be the system-defined roles (except `public`) listed in the *Security Guide*.

- `-user <user_name>`

  Specifies the user who the role will be revoked from.

This command is deprecated. For more information see Revoke Roles or Privileges in the *Security Guide*.

plan start-service

```
plan start-service {-service <id> | -all-rns [-zn <id> |
    -znname <name>] | -all-ans [-zn <id> | -znname <name>] |
    -zn <id> | -znname <name> } [-plan-name <name>]
    [-json] [-wait] [-noexecute] [-force]
```

Starts the specified service(s). The service may be a Replication Node, an Arbiter Node, or Admin service, as identified by any valid string.

For example, to identify a Replication Node, use `-service shardId-nodeId`, where `shardId-nodeId` must be given as a single argument with one embedded hyphen and no spaces. The `shardId` identifier is represented by rgX, where X refers to the shard number.

where:

- `-service`

  Specifies the name of the service to start.

- `-all-rns`

  If specified, starts the services of all Replication Nodes in a store.

- `-all-ans`

  If specified, starts all the Arbiter Nodes in the specified zone.

Note:

This plan cannot be used to start a Storage Node. Further, you cannot restart the Storage Node's services without first starting the Storage Node itself. To start the Storage Node, go to the Storage Node host and enter the following command:

```
nohup java -Xmx64m -Xms64m \
-jar <KVHOME>/lib/kvstore.jar start -root <KVROOT> &

kv-> plan start-service -service rg1-rn3 -json -wait
{
    "operation" : "Start Services",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "id" : 21,
        "name" : "Start Services",
        "isDone" : true,
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 09:50:54 UTC",
        "interrupted" : null,
        "end" : "2017-09-28 09:50:57 UTC",
        "error" : null,"executionDetails" : {
        "taskCounts" : {
            "total" : 2,
            "successful" : 2,
            "failed" : 0,
            "interrupted" : 0,
            "incomplete" : 0,
            "notStarted" : 0
        },
    "finished" : [ {
        "taskNum" : 1,
        "name" : "Plan 21 [Start Services] task [StartNode]",
        "state" : "SUCCEEDED",
        "start" : "2017-09-28 09:50:54 UTC",
        "end" : "2017-09-28 09:50:55 UTC"
        }, {
    "taskNum" : 2,
    "name" : "Plan 21 [Start Services] task [WaitForNodeState rg1-
rn3 to reach RUNNING]",
    "state" : "SUCCEEDED",
    "start" : "2017-09-28 09:50:55 UTC",
    "end" : "2017-09-28 09:50:57 UTC"
        } ],
    "running" : [ ],
    "pending" : [ ]
        }
    }
}
```

plan stop-service

```
plan stop-service {-service <id> |
    -all-rns [-zn <id> | -znname <name>] | -all-ans [-zn <id> |
    -znname <name>] | -zn <id> | -znname <name> }
    [-plan-name <name>] [-json] [-wait] [-noexecute] [-force]
```

Stops the specified service(s). The service may be a Replication Node, an Arbiter Node, or Admin service as identified by any valid string.

For example, to identify a Replication Node, use `-service` *`shardId-nodeId`*, where *`shardId-nodeId`* must be a single string with an embedded hyphen (-) and no spaces. The *`shardId`* identifier is represented as *`rgX`*, where *X* represents the shard number.

Other options to specify after `-service` include:

- `-all-rns`

  Stops the services of all Replication Nodes in a store.

- `-all-ans`

  Stops the services of all Arbiter Nodes in the specified zone.

Use this command to stop any affected services so that any attempts by the system to communicate with the services are no longer accepted. Stopping communication to one or more services reduces the amount of error output about a failure you are already aware of.

Whenever you execute the `plan stop-service` command, the system automatically initiates a health check. The health check determines if stopping an indicated service will result in losing quorum. There are no further checks performed, only whether quorum will be lost if you stop the service. To avoid losing quorum, the `plan stop-service` fails to execute if the health check fails, and outputs detailed health check information such as the following:

```
One of the groups is not healthy enough for the operation:
[rg1] Only 1 primary nodes are running such that a simple
majority cannot be formed which requires 2 primary nodes.
The shard is vulnerable and will not be able to elect a new master.
Nodes not running: [rg1-rn1]. Nodes to stop: {rg1=[rg1-rn2]}
...
```

If you cannot stop a service because it will result in lost quorum, you should determine what problem is occurring before trying to stop the service.

If, on the other hand, you understand that stopping a service will result in losing quorum, but such an event is necessary to make some important change, you can force the `plan stop-service` command to execute by appending the `-force` flag.

> **Note:**
>
> If you forcefully stop the Admin service and Admin quorum is lost, you cannot use the `start-service` plan to bring up the Admin services anymore. All plan operations will also fail thereafter.

The `plan stop-service` command is also useful during disk replacement process. Use the command to stop the affected service prior removing the failed disk. For more information, see Replacing a Failed Disk.

> **Note:**
>
> - This plan cannot be used to stop a Storage Node. To stop a Storage Node, first stop all services running on it. Then, find the ID of the Storage Node process by going to the Storage Node host and issuing this command:
>
>  ```
>  ps -af | grep -e "kvstore.jar.*start.*<KVROOT>"
>  ```
>
> Kill the process using:
>
>  ```
>  kill <storage node id>
>  ```
>
> - Also, because the `plan stop-service -all-rns` command always results in losing quorum, executing `plan stop-service` with this option skips running a health check. Further, you do not need to use the `-force` flag is when using the `-all-rns` option.

plan verify-data

```
plan verify-data
    [-verify-log <enable|disable> [-log-read-delay <milliseconds>]]
    [-verify-btree <enable|disable> [-btree-batch-delay <milliseconds>]
        [-index <enable|disable>] [-datarecord <enable|disable>]]
    [-valid-time <time>]
    [-show-corrupt-files <enable|disable>]
    -service <id> | -all-services [-zn <id> | -znname <name>] |
    -all-rns [-zn <id> | -znname <name>] |
    -all-admins [-zn <id> | -znname <name>]
    [-plan-name <name>] [-wait] [-noexecute] [-force] [-json|-json-v1]
```

Verifies and controls certain elements (such as log files and indexes), as presented in this section. Here is a description for each of the `verify-data` parameters and options:

Option	Description
`-verify-log`	Verifies the checksum of each data record in the JE log file of JE. The Berkeley DB Java Edition (JE) is the data storage engine of Oracle NoSQL Database. It is enabled by default.
`-log-read-delay`	Configures the delay time between file reads. The default value is 100 milliseconds.
`-verify-btree`	Verifies that the B-tree of the database in memory contains a valid reference to each data record on disk. You can combine `-verify-btree` with `-datarecord` and `-index`. It is enabled by default.
`-btree-batch-delay`	Configures the delay time, in milliseconds, between batches. The default delay value is 10 milliseconds.
`-datarecord`	Reads and verifies data records from disk, if the data record is not in cache. The `-datarecord` option takes longer than verifying records only in cache, and results in more read I/O. It is disabled by default.
`-index`	Verifies indexes. Using the `-index` option alone verifies only the reference from the index to the primary table, not the reference from the primary table to index. To verify both references from index to primary table, and primary table to index, specify the `-datarecord` and `-index` options. It is enabled by default.
`-valid-time`	Specifies the amount of time for which an existing verification will be considered valid and not be rerun. The format is 'number unit' where the unit can be minutes or seconds. The unit is case insensitive and can be separated from the number by a space, "-" or "_". The default is '10 minutes'.
`-show-corrupt-files`	Specifies whether to show corrupt files, including missing files and reserved files that are referenced. It is disabled by default.
`-service` *`id`*	Runs verification on the specified service (*id*)
`-all-services [-zn` *`id`* `| -znname` *`name`*`]`	Runs verification on all services, both RNs and Admins, in the specified zone, or in all zones if none is specified.
`| -all-rns [-zn` *`id`* `| -znname` *`name`*`]`	Runs verification on all RNs in the specified zone, or in all zones if none is specified.
`| -all-admins [-zn` *`id`* `| -znname` *`name`*`]`	Runs verification on all Admins in the specified zone, or in all zones if none is specified.
`[-plan-name` *`name`*`]`	Runs the named plan that you have saved to execute `plan verify-data` and its available options.
`[-wait]`	Runs a plan synchronously, so that the command line prompt returns after the command completes.
`[-noexecute]`	Lets you create a plan but delay its execution. Conversely, use the `plan execute` command to run the plan.
`[-force]`	Runs the plan as you enter it on the CLI, without validating the flags.

Option	Description
`[-json|-json-v1]`	Displays the plan output as json or json-v1.

Executing verify-data

The `plan verify-data` command is available to verify both primary table and secondary indexes. The command lets you verify either a checksum of data records, or the B-tree of the database.

> **Note:**
>
> Since Oracle NoSQL Database uses Oracle Berkeley DB Java Edition (JE) as its underlying storage engine, verifying data using `plan verify-data` depends on several low-level JE features that are neither described here, nor visible. Throughout this section, terms or concepts related to Oracle Berkeley DB Java Edition (JE) are indicated by the term *Berkeley*, indicating their origination. For more information about Oracle Berkeley DB Java Edition, start here: Oracle Berkeley DB Java Edition.

The `plan verify-data` has two parts for verifications:

- Log record integrity on disk
- B-tree integrity

To verify the integrity of log records on disk, `verify-data` accesses and verifies each record's checksum. Since this procedure includes disk reads, it consumes I/O resources and is relatively time consuming. To reduce the performance effects of verification, you can configure a longer delay time between reading each batch of log files. While increasing the delay time increases operation time overall, it consumes fewer I/O activities. If that choice is preferable for your requirements, use `-btree-batch-delay` to increase the delay between log file integrity checks during peak I/O operations.

When verifying B-tree integrity, the `plan verify-data` process verifies in-memory integrity. The basic verification checks only if the LSN (*Berkeley*) for each data record in primary tables is valid. You can configure the verification to include data records on disk, as well as secondary index integrity.

If you do not enable data record verification, the secondary index verification checks only the reference from secondary index to primary table, but not from primary table to index. Since basic verification checks only in-memory data structures, it is significantly faster and less resource intensive than verification involving disk reads.

plan wait

```
plan wait -id <id> | -last [-seconds <timeout in seconds>] [-json]
```

Waits indefinitely for the specified plan to complete, unless the optional timeout is specified.

Use the `-seconds` option to specify the time to wait for the plan to complete.

The `-last` option references the most recently created plan.

```
kv-> plan wait -id 26 -json
{
    "operation" : "plan wait",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "planId" : 26,
        "state" : "CANCELED"
    }
}
```

pool

Encapsulates commands that manipulates Storage Node pools, which are used for resource allocations. The subcommands are as follows:

- pool clone
- pool create
- pool join
- pool leave
- pool remove

pool clone

```
pool clone -name <name> -from <source pool name> [-json]
```

Clone an existing Storage Node pool to a new Storage Node pool to be used for resource distribution when creating or modifying a store.

For more information on using a cloned Storage Node pool when contracting a topology, see Contracting a Topology.

```
kv-> pool clone -name mypool from snpool -json{
"operation" : "pool clone",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "poolName" : "mypool"
    }
}
```

pool create

```
pool create -name <name> -json
```

Creates a new Storage Node pool to be used for resource distribution when creating or modifying a store.

For more information on creating a Storage Node pool, see Create a Storage Node Pool.

```
kv-> pool create -name newPool -json{
"operation" : "pool create",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "storeName" : "newPool"
    }
}
```

pool join

```
pool join -name <name> [-sn <snX>]* [-json]
```

Adds Storage Nodes to an existing Storage Node pool.

```
kv-> pool join -name newPool -sn sn1 -json{
"operation" : "pool join",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "storeName" : "newPool"
    }
}
```

pool leave

```
pool leave -name <name> [-sn <snX>]* [-json]
```

Remove Storage Nodes from an existing Storage Node pool.

```
kv-> pool leave -name newPool -sn sn1 -json{
"operation" : "pool leave",
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "storeName" : "newPool"
    }
}
```

pool remove

```
pool remove -name <name>
```

Removes a Storage Node pool.

```
kv-> pool remove -name newPool -json{
"operation" : "pool remove",
```

```
"returnCode" : 5000,
"description" : "Operation ends successfully",
"returnValue" : {
    "storeName" : "newPool"
    }
}
```

put

Encapsulates commands that put key/value pairs to the store or put rows to a table. The subcommands are as follows:

- put kv
- put table

put kv

```
put kv -key <keyString> -value <valueString> [-file]
    [-hex] [-if-absent] [-if-present]
```

Put the specified key/value pair into the store. The following arguments apply to the put command:

- `-key<keyString>`

  Specifies the name of the key to be put into the store. Key can be composed of both major and minor key paths, or a major key path only. The <keyString> format is: "major-key-path/-/minor-key-path".

  For example, a key containing major and minor key paths:

  ```
  kv-> put kv -key /Smith/Bob/-/email -value
  "{\"id\": 1,\"email\":\"bob.smith@example.com\"}"
  ```

  For example, a key containing only a major key path:

  ```
  kv-> put kv -key /Smith/Bob -value"{\"name\":
   \"bob.smith\", \"age\": 20, \"phone\":\"408 555 5555\", \"email\":
   \"bob.smith@example.com\"}"
  ```

- `-value <valueString>`

  If -file is not specified, the <valueString> is treated as a raw bytes array.

  For example:

  ```
  kv-> put kv -key /Smith/Bob/-/phonenumber -value "408 555 5555"
  ```

> **Note:**
>
> The mapping of the raw arrays to data structures (serialization and deserialization) is left entirely to the application.

- `-file`

  Indicates that the value is obtained from a file. The file to use is identified by the value parameter.

  For example:

  ```
  kv-> put kv -key /Smith/Bob -value ./smith-bob-info.txt
   -file
  ```

- `-hex`

  Indicates that the value is a BinHex encoded byte value with base64 encoding.

- `-if-absent`

  Indicates that a key/value pair is `put` only if no value for the given key is present.

- `-if-present`

  Indicates that a key/value pair is `put` only if a value for the given key is present.

put table

```
kv-> put table -name table_name [if-absent | -if-present ]
    [-json string] [-file file_name] [-exact] [-update]
```

Puts one or more rows into the named table.

- `-name`
  Specifies a table name, which can identify different types of tables:
  - *`table_name`* – The table is a top level table created in the default namespace, `sysdefault`. The default `sysdefault:` namespace prefix is not required.
  - *`table_name.child_name`* – The table is a child table. Always precede a *`child_name`* table with its parent *`table_name`*, followed by a period (.) separator.
  - *`namespace_name:table_name`* – The table was created in the namespace you specify. Always precede *`table_name`* with its *`namespace_name`*, followed by a colon (:).
  - *`namespace_name:table_name.child_name`* – The table is a child table of a parent table created in a namespace. Specify *`child_name`* by preceding it with both *`namespace_name:`* and its parent *`table_name`*, , followed by a period (.) separator.

- `-if-absent`

  Indicates to put a row only if the row does not exist.

- `-if-present`

  Indicates to put a row only if the row already exists.

- `-json`

Indicates that the value is a JSON string.

- `-file`

  Use to load a file of JSON strings.

- `-exact`

  Indicates that the input JSON string or file must contain values for all columns in the table and cannot contain extraneous fields.

- `-update`

  Can be used to partially update the existing record.

repair-admin-quorum

```
repair-admin-quorum {-zn <id> | -znname <name> | -admin <id>}...
```

Restores Admin quorum after it is lost by reducing membership of the admin group to the admins in the specified zones, or to the specific admins you can list. Use this command when attempting to recover from a failure that has resulted in losing admin quorum. This command can result in data loss.

After obtaining a working admin by using the `repair-admin-quorum` command, call the `plan failover` command to failover to the zones that remain available after a failure, and to update the topology to match the changes made to the admins.

The arguments specify which admins to use as the new set of primary admins, either by specifying all of the admins in one or more zones, or by identifying specific admins. The specified set of admins must not be empty, must contain only currently available admins, and must include all currently available primary admins. It may also include secondary admins, if desired, to increase the admin replication factor or because no primary admins are available.

> **Note:**
>
> You can repeat this command if a temporary network or component failure results in the initial command invocation to fail.

show

Encapsulates commands that display the state of the store and its components or schemas. The subcommands are as follows:

- show admins
- show datacenters
- show events
- show faults
- show indexes
- show mrtable-agent-statistics

- show parameters
- show perf
- show plans
- show pools
- show snapshots
- show regions
- show tables
- show topology
- show upgrade-order
- show users
- show versions
- show zones

show admins

```
show admins [-json]
```

Displays basic information about Admin services.

```
kv-> show admins -json
{
    "operation" : "show admins",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
    "admins" : [ {
        "id" : "admin1",
        "storageNode" : "sn1",
        "type" : "PRIMARY",
        "connected" : true,
        "adminStatus" : "RUNNING",
        "replicationState" : "MASTER",
        "authoritative" : true
    }, {
        "id" : "admin2",
        "storageNode" : "sn2",
        "type" : "PRIMARY",
        "connected" : false,
        "adminStatus" : "RUNNING",
        "replicationState" : "REPLICA",
        "authoritative" : true
        } ]
    }
}
```

show datacenters

```
show datacenters
```

Deprecated. See show zones instead.

show events

```
show events [-id <id>] | [-from <date>] [-to <date> ]
    [-type <stat | log | perf>] [-json]
```

Displays event details or list of store events. The status events indicate changes in service status.

Log events are noted if they require attention.

Performance events are not usually critical but may merit investigation. Events marked "SEVERE" should be investigated.

The following date/time formats are accepted. They are interpreted in the local time zone.

- **MM-dd-yy HH:mm:ss:SS**
- **MM-dd-yy HH:mm:ss**
- **MM-dd-yy HH:mm**
- **MM-dd-yy**
- **HH:mm:ss:SS**
- **HH:mm:ss**
- **HH:mm**

For more information on events, see Events.

```
kv-> show events -json
{
    "operation" : "show events",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "events" : [ {
            "event" : "j84a16s3S STAT 2017-09-28 09:48:12.819 UTC sn1
RUNNING sev1"
        }, {
            "event" : "j84a17j0S STAT 2017-09-28 09:48:13.788 UTC sn2
RUNNING sev1"
        }, {
            "event" : "j84a19xoS STAT 2017-09-28 09:48:16.908 UTC sn3
RUNNING sev1"
        }, {
            "event" : "j84a1cznS STAT 2017-09-28 09:48:20.867 UTC rg1-
rn1 RUNNING sev1"
```

```
        }, {
            "event" : "j84a1f75S STAT 2017-09-28 09:48:23.729 UTC rg1-rn2
RUNNING sev1"
        }, {
            "event" : "j84a1h7xS STAT 2017-09-28 09:48:26.349 UTC rg1-rn3
RUNNING sev1"
        }, {
            "event" : "j84a3i9rS STAT 2017-09-28 09:50:01.023 UTC rg1-rn3
STOPPED sev1 (reported by sn3)"
        }, {
            "event" : "j84a4oquS STAT 2017-09-28 09:50:56.070 UTC rg1-rn3
RUNNING sev1"
        }, {
            "event" : "j84a5hfeS STAT 2017-09-28 09:51:33.242 UTC rg1-rn3
STOPPED sev1 (reported by sn3)"
        }, {
            "event" : "j84aw53tS STAT 2017-09-28 10:12:16.985 UTC sn3
UNREACHABLE sev2 (reported by
            admin1)"
        }, {
            "event" : "j84b585yL LOG 2017-09-28 10:19:20.854 UTC SEVERE
[admin1] Plan 24 [Remove Admin
            Replica] task [DestroyAdmin admin3] of plan 24 ended in state
ERROR with java.rmi.ConnectException:
            Unable to connect to the storage node agent at host localhost,
port 22000, which may not be running;
            nested exception is: "
        }, {
            "event" : "j84b585zL LOG 2017-09-28 10:19:20.854 UTC SEVERE
[admin1] Plan [null] failed. Attempt 1
            [RUNNING] start=2017-09-28 10:19:20 UTC end=2017-09-28 10:19:20
UTC "
        } ]
    }
}
```

show faults

```
show faults [-last] [-command <command index>] [-json]
```

Displays faulting commands. By default all available faulting commands are displayed. Individual fault details can be displayed using the -last and -command flags.

```
kv-> show faults -json
{
    "operation" : "show faults",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "faultCommands" : [ {
            "faultCommand" : "503 plan remove-admin -admin 3 -json -wait:
class
            oracle.kv.util.shell.ShellException"
```

```
            }, {
            "faultCommand" : "526 topology create -name mytopo -pool
snpool -json -partitions 300 -json: class
            java.lang.NullPointerException"
            } ]
        }
}
```

show indexes

```
show indexes [-table <name>] [-name <name>] [-json]
```

Displays index metadata. By default the indexes metadata of all tables are listed.

If a specific table is named, its indexes metadata are displayed. If a specific index of the table is named, its metadata is displayed. For more information, see plan add-index.

Use `SHOW INDEX` statement to indicate the index type (TEXT, SECONDARY) when you enable text-searching capability to Oracle NoSQL Database, in-concert with the tables interface.

For example:

```
kv-> show index
Indexes on table Joke
JokeIndex (category, txt), type: TEXT
```

For more information, see Integration with Elastic Search for Full Text Search in the *Integrations Guide*.

```
kv-> show indexes -json
{
    "operation" : "show indexes",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "tables" : [ {
        "table" : {
            "tableName" : "t1",
        "indexes" : [ {
        "name" : "idx1",
        "fields" : [ "id1", "id2" ],
        "type" : "SECONDARY",
    "description" : null
    }, {
        "name" : "idx2",
        "fields" : [ "id2" ],
        "type" : "SECONDARY",
        "description" : null
    } ]
},
    "childTable" : [ {
```

```
        "tables" : [ ]
        } ]
    } ]
    }
}
```

show mrtable-agent-statistics

```
show mrtable-agent-statistics [-agent <agentID>][-table <tableName>][-json]
```

Shows the latest statistics as of the last one minute for multi-region table agents. With no arguments, this command shows combined statistics over all regions the MR Table spans.

Input Parameters

Optionally, you can enable the following flags with appropriate parameters with this command:

Table 7-1 Input Parameters

Flag	Parameter	Description
`-agent`	`agentID`	Limits the statistics to the agent ID specified. You can find the agent ID from the JSON config file created while configuring your agent. See Configure XRegion Service.
`-table`	`tableName`	Limits the statistics to the MR Table specified.
`-json`	-	Returns the complete statistics in JSON format. Even though the statistics are returned in JSON format by default, specifying this flag adds additional information in the output such as operation, return code, and the return code's description.

Output Statistics

The statistics reported by the `show mrtable-agent-statistics` can be categorized as those used to:

- **Monitor streams from other regions**

Table 7-2 Output Statistics 1

Statistic	Description
`completeWriteOps`	Number of complete write operations per region.
`lastMessageMs`	Timestamp when the agent sees the last message from a remote region, in milliseconds. If this statistic information is not available, -1 is printed as its output value.

Table 7-2 (Cont.) Output Statistics 1

Statistic	Description
`lastModificationMs`	Timestamp of the last operation performed in each remote region, in milliseconds. If this statistic information is not available, -1 is printed as its output value.
`laggingMs (avg, max, min)`	In a multi-region KVStore, each shard in a region pushes all the operations performed on all its tables to the agent's queue. The agent replicates the contents of its queue, in event order, to all other regions. The lagging statistic represents the time difference between an event being pushed into the queue and replicated to the other regions by the agent. If this value is high, it indicates that the queue is getting backed up. A smaller value indicates that the agent is able to keep up with the number of events coming from remote regions. The lagging statistics are reported as average, minimum, and maximum in milliseconds for each remote region. If this statistic information is not available, -1 is printed as its output value.
`latencyMs (avg, max, min)`	In MR tables, the latency statistic indicates the time taken in milliseconds for each operation to travel from its origin (remote) region to the target (local) region. The latency is computed as T2 - T1, where: – T1 is the timestamp when an operation is performed in the remote region, and – T2 is the timestamp when the agent persisted the replicated operation to the local region. For each remote region, the latency statistics are reported as the average, minimum, and maximum latency for all the operations from that region. If this statistic information is not available, -1 is printed as its output value.

- **Check the persistence of remote data**

Table 7-3 Output Statistics 2

Statistic	Description
`puts`	Number of write operations received.
`dels`	Number of delete operations received.
`streamBytes`	Total bytes replicated from a remote region.

Table 7-3 (Cont.) Output Statistics 2

Statistic	Description
`persistStreamBytes`	Reports the total number of bytes that are successfully committed in the local region. This is different from the total bytes replicated from a remote region because in case of any conflicts with operations from other regions, some of the operations may not persist if they fail the built-in conflict resolution rule.
`winPuts`	Number of write operations performed successfully. More specifically, this statistic excludes the writes that failed to win the conflict resolution rule, in case of a conflict with writes in other regions.
`winDels`	Number of delete operations performed successfully. More specifically, this statistic excludes the deletes that failed to win the conflict resolution rule, in case of a conflict with deletes in other regions.
`incompatibleRows`	Number of operations that did not persist because of incompatible table schemas. This can happen when there is a schema mismatch between the origin region and the region that is trying to replicate the row to its local data store.

- **Monitor the interaction between admin and the agent**

Table 7-4 Output Statistics 3

Statistic	Description
`requests`	All the DDL commands executed by the user on an MR table are converted into requests to the agent by the admin. This statistic reports the number of requests posted by the admin.
`responses`	Number of requests processed and responded by the agent.

- **Monitor multi-region tables**
  When you execute the `show mrtable-agent-statistics` command with the `-table` flag, it returns the table level statistics indicating:

  1. **Persistence of remote data in the local region:** This includes the statistics such as `puts`, `dels`, `winPuts`, `winDels`, `streamBytes`, `persistStreamBytes`, and `incompatibleRows` discussed above.
  2. **Progress of table initialization in each remote region:** This is indicated by the `state` attribute under the `Initialization statistics` in the output. The table below lists the different possible values for state and their meaning.

Table 7-5 Table Initialization States

State	Description
NOT_START	MR table initialization has not started, or there is no need to do initialization. For example, if the agent resumes the stream from an existing checkpoint successfully, there is no need to re-initialize the MR table.
IN_PROGRESS	MR table initialization is ongoing, that is, the MR table initialization has started and the data is being replicated from the remote regions.
COMPLETE	MR table initialization is complete and table transfer is done. The agent is streaming from the remote region.
ERROR	MR table initialization cannot complete because of an irrecoverable error. You can view the error severity in the agent log as WARNING or SEVERE. The agent log can be found in the directory specified in the JSON config file. See Configure XRegion Service.
SHUTDOWN	MR table initialization cannot complete because the service is shut down.

3. **Persistence of the table data per remote region:**

Table 7-6 Output Statistics 4

Statistic	Description
transferStartMs	Timestamp of the initiation of table initialization, in milliseconds. If this statistic information is not available, -1 is printed as its output value.
transferCompleteMs	Timestamp of the completion of table initialization, in milliseconds. If this statistic information is not available, -1 is printed as its output value.
elapsedMs	The time elapsed from the start of the table initialization until its completion. elapsedMs = transferCompleteMs - transferStartMs This statistic is reported in milliseconds. Before the transfer completion, it reports *-1* indicating the unavailability of this statistic.
transferBytes	Number of bytes transferred from the remote (origin or source) region to the local (target) region.
transferRows	Number of rows transferred from the remote region to the local region successfully.

Table 7-6 (Cont.) Output Statistics 4

Statistic	Description
`expireRows`	Number of rows expired before transferring from the remote region. Because of their TTL value, some rows might expire during the replication. Such rows expire by the time they reach the agent. This statistic counts such expired rows.
`persistBytes`	Reports the total number of bytes that are successfully committed in the local region. This excludes the rows that are not committed in the local region because they failed the built-in conflict resolution rule. In case of row updates, the entire row is counted for this statistic.
`persistRows`	Reports the total number of rows that are successfully committed in the local region. Similar to the above statistic, the rows that are not committed in the local region because of the built-in conflict resolution rule are excluded for this count.

Example

Below are a few examples of the statistics returned by the show mrtable-agent-statistics command with different input parameters.

> **Note:**
>
> If any of the statistics information is not available, -1 is reported as the value for that statistic parameter in the output.

```
# MR table agent statistics for a specific agent
kv-> show mrtable-agent-statistics -agent 0 -json
{
  "operation": "show mrtable-agent-statistics",
  "returnCode": 5000,
  "description": "Operation ends successfully",
  "returnValue": {
    "XRegionService-1_0": {
      "timestamp": 1592901180001,
      "statistics": {
        "agentId": "XRegionService-1_0",
        "beginMs": 1592901120001,
        "dels": 1024,
        "endMs": 1592901180001,
        "incompatibleRows": 100,
        "intervalMs": 60000,
        "localRegion": "slc1",
        "persistStreamBytes": 524288,
        "puts": 2048,
```

```
        "regionStat": {
          "lnd": {
            "completeWriteOps": 10,
            "laggingMs": {
              "avg": 512,
              "max": 998,
              "min": 31
            },
            "lastMessageMs": 1591594977587,
            "lastModificationMs": 1591594941686,
            "latencyMs": {
              "avg": 20,
              "max": 40,
              "min": 10
            }
          },
          "dub": {
            "completeWriteOps": 20,
            "laggingMs": {
              "avg": 535,
              "max": 1024,
              "min": 45
            },
            "lastMessageMs": 1591594978254,
            "lastModificationMs": 1591594956786,
            "latencyMs": {
              "avg": 30,
              "max": 45,
              "min": 15
            }
          }
        },
        "requests": 12,
        "responses": 12,
        "streamBytes": 1048576,
        "winDels": 1024,
        "winPuts": 2048
      }
    }
  }
}
```

```
# MR table agent statistics for a specific MR table
kv-> show mrtable-agent-statistics -table users -json
{
  "operation": "show mrtable-agent-statistics",
  "returnCode": 5000,
  "description": "Operation ends successfully",
  "returnValue": {
    "XRegionService-1_0": {
      "tableID": 12,
      "tableName": "users",
      "timestamp": 1592901300001,
      "statistics": {
```

```
          "agentId": "XRegionService-1_0",
          "beginMs": 1592901240001,
          "dels": 1000,
          "endMs": 1592901300001,
          "expiredPuts": 200,
          "incompatibleRows": 100,
          "initialization": {
            "lnd": {
              "elapsedMs": 476,
              "expireRows": 100,
              "persistBytes": 6492160,
              "persistRows": 6340,
              "state": "COMPLETE",
              "transferBytes": 8115200,
              "transferCompleteMs": 1592822625333,
              "transferRows": 7925,
              "transferStartMs": 1592822624857
            },
            "dub": {
              "transferStartMs": 0,
              "transferCompleteMs": 0,
              "elapsedMs": -1,
              "transferRows": 0,
              "persistRows": 0,
              "expireRows": 0,
              "transferBytes": 0,
              "persistBytes": 0,
              "state": "NOT_START"
            }
          },
          "intervalMs": 60000,
          "localRegion": "fra",
          "persistStreamBytes": 104960000,
          "puts": 100000,
          "streamBytes": 115200000,
          "tableId": 12,
          "tableName": "users",
          "winDels": 745,
          "winPuts": 90000
        }
      }
    }
  }
```


show parameters

```
show parameters -policy | -service <name>
```

Displays service parameters and state for the specified service. The service may be a Replication Node, Storage Node, or Admin service, as identified by any valid string, for

example rg1-rn1, sn1, admin2, etc. Use the -policy flag to show global policy parameters. Use the -security flag to show global security parameters.

```
show parameters -service sn1
```

When you enable text-searching capability to Oracle NoSQL Database, in-concert with the tables interface, the `show parameter` command also provides information on the Elasticsearch cluster name and transport port as values for the parameters searchClusterMembers and searchClusterName.

For more information, see Integration with Elastic Search for Full Text Search in the *Integrations Guide*.

show perf

```
show perf
```

Displays recent performance information for each Replication Node.

show plans

```
show plans [-last] [-id <id>] [-from <date>] [-to <date>][-num
<howMany>]
```

Shows details of the specified plan or list all plans that have been created along with their corresponding plan IDs and status.

- The `-last` option shows details of the most recently created plan.
- The `-id <n>` option details the plan with the given id. If -num <n> is also given, list <n> plans, starting with plan #<id>.
- The `-num <n>` option sets the number of plans to the list. The default is 10.
- The `-from <date>` option lists plans after <date>.
- The `-to <date>` option lists plans before <date>.

Combining `-from` with `-to` describes the range between the two dates. Otherwise `-num` applies.

The following date formats are accepted. They are interpreted in the UTC time zone.

- *yyyy-MM-dd HH:mm:ss.SSS*
- *yyyy-MM-dd HH:mm:ss*
- *yyyy-MM-dd HH:mm*
- *yyyy-MM-dd*
- *MM-dd-yyyy HH:mm:ss.SSS*
- *MM-dd-yyyy HH:mm:ss*
- *MM-dd-yyyy HH:mm*
- *MM-dd-yyyy*

- *HH:mm:ss.SSS*
- *HH:mm:ss*
- *HH:mm*

For more information on plan review, see Reviewing Plans.

show pools

```
show pools
```

Lists the Storage Node pools.

show snapshots

```
show snapshots [-sn <id>]
```

Lists snapshots on the specified Storage Node. If no Storage Node is specified, one is chosen from the store. You can use this command to view the existing snapshots.

show regions

```
show regions
```

Displays the list of all the remote regions included in a Multi-Region Oracle NoSQL Database setup.

```
kv-> execute 'show regions'
regions
  DEN
```

show tables

```
show tables -name table_name
```

Displays the table information. Use `-original` flag to show the original table information if you are building a table for evolution. The flag is ignored for building table for addition. For more information, see plan add-table and plan evolve-table

Use `show table -name table_name` statement to list the full text index. This command provides the table structure including the indexes that have been created for that table. For more information, see Creating FTI in the *Integrations Guide*.

show topology

```
show topology [-zn] [-rn] [-an] [-sn] [-store] [-status] [-json] [-verbose]
```

Displays the current, deployed topology. By default it shows the entire topology, including the number of shards. The first set of optional flags restrict the display to one or more zones,

Replication Nodes, Storage Nodes, Arbiter Nodes, store name, or to specify service status. Use –json to display the results in JSON format. If you specify -verbose, then additional information will be displayed, including Replication Node storage directories, storage directory sizes, log directories, and JE HA ports.

You can also obtain the zone ID to which you can deploy Storage Nodes.

```
kv-> show topology
store=mystore  numPartitions=1000 sequence=2376
  zn: id=zn1 name=myzone repFactor=3 type=PRIMARY allowArbiters=false
masterAffinity=false
  sn=[sn1] zn:[id=zn1 name=myzone] nodeA:5000 capacity=1 RUNNING
    [rg1-rn1] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=0.0 ms
  sn=[sn2] zn:[id=zn1 name=myzone] nodeB:5000 capacity=1 RUNNING
    [rg1-rn2] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=0.0 ms
  sn=[sn3] zn:[id=zn1 name=myzone] nodeC:5000 capacity=1 RUNNING
    [rg1-rn3] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=0.0 ms
  sn=[sn4] zn:[id=zn1 name=myzone] nodeD:5000 capacity=1 RUNNING
    [rg2-rn1] RUNNING
          No performance info available
  sn=[sn5] zn:[id=zn1 name=myzone] nodeE:5000 capacity=1 RUNNING
    [rg2-rn2] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=0.0 ms
  sn=[sn6] zn:[id=zn1 name=myzone] nodeF:5000 capacity=1 RUNNING
    [rg2-rn3] RUNNING
             single-op avg latency=0.0 ms   multi-op avg latency=0.0 ms

  numShards=2
  shard=[rg1] num partitions=500
    [rg1-rn1] sn=sn1
    [rg1-rn2] sn=sn2
    [rg1-rn3] sn=sn3
  shard=[rg2] num partitions=500
    [rg2-rn1] sn=sn4
    [rg2-rn2] sn=sn5
    [rg2-rn3] sn=sn6
```

show upgrade-order

```
show upgrade-order [-json]
```

Lists the Storage Nodes which need to be upgraded in an order that prevents disruption to the store's operation.

This command displays one or more Storage Nodes on a line. Multiple Storage Nodes on a line are separated by a space. If multiple Storage Nodes appear on a single line, then those nodes can be safely upgraded at the same time. When multiple nodes are upgraded at the same time, the upgrade must be completed on all nodes before the nodes next on the list can be upgraded.

If at some point you lose track of which group of nodes should be upgraded next, you can always run the show upgrade-order command again.

```
kv-> show upgrade-order -json
{
    "operation" : "show upgrade-order",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
    "singleTextResult" : "Calculating upgrade order, target version:
12.2.4.6.0, prerequisite:
    12.1.3.0.5\nUnable to contact sn3 Unable to connect to the storage node
agent at host localhost, port
    22000, which may not be running; nested exception is:
\n\tjava.rmi.ConnectException: Connection refused
    to host: localhost; nested exception is: \n\tjava.net.ConnectException:
Connection refused (Connection
    refused)\nThere are no nodes that need to be upgraded"
    }
}
```

show users

```
show users -name <name>
```

Lists the names of all users, or displays information about a specific user. If no user is specified, lists the names of all users. If a user is specified using the `-name` option, then lists detailed information about the user.

show versions

```
show versions [-json]
```

Lists the client and server version information.

For example

```
kv-> show versions
Client version: 12.1.3.4.0
Server version: 12.1.3.4.0

kv-> show versions -json
{
    "operation" : "show version",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "clientVersion" : "12.2.4.6.0",
        "serverVersion" : "12.2.4.6.0"
    }
}
```

show zones

```
show zones [-zn <id>] | -znname <name>] [-json]
```

Lists the names of all zones, or display information about a specific zone.

Use the `-zn` or the `-znname` flag to specify the zone that you want to show additional information; including the names of all of the Storage Nodes in the specified zone, and whether that zone is a primary of secondary zone.

```
kv-> show zones -json
{
    "operation" : "show zone",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "zones" : [ {
            "zone" : {
            "id" : "zn1",
            "name" : "1",
            "repfactor" : 1,
            "type" : "PRIMARY",
            "allowArbiters" : false
        }
        }, {
            "zone" : {
            "id" : "zn2",
            "name" : "2",
            "repfactor" : 1,
            "type" : "PRIMARY",
            "allowArbiters" : false
        }
        }, {
            "zone" : {
            "id" : "zn3",
            "name" : "3",
            "repfactor" : 1,
            "type" : "PRIMARY",
            "allowArbiters" : false
        }
        } ]
    }
}
```

snapshot

Encapsulates commands that create and delete snapshots, which are used for backup and restore. The subcommands are as follows:

- snapshot create
- snapshot remove

snapshot create

```
snapshot create -name <name>
```

Creates a new snapshot using the specified name as the prefix.

Use the `-name` option to specify the name of the snapshot that you want to create.

Snapshots should not be taken while any configuration (topological) changes are being made, because the snapshot might be inconsistent and not usable.

snapshot remove

```
snapshot remove -name <name> | -all
```

Removes the named snapshot. If -all is specified, remove all snapshots.

Use the `-name` option to specify the name of the snapshot that you want to remove.

If the `-all` option is specified, remove all snapshots.

To create a backup of your store using a snapshot see Taking a Snapshot.

To recover your store from a previously created snapshot you can use the load utility or restore directly from a snapshot. For more information, see Using the Load Program or Restoring Directly from a Snapshot.

table

Deprecated with exception of table-size. See execute instead.

table-size

```
table-size -name <name> -json <string>
    [-rows <num> [[-primarykey | -index <name>] -keyprefix <size>]]
```

Calculates key and data sizes for the specified table using the row input, optionally estimating the NoSQL DB cache size required for a specified number of rows of the same format. Running this command on multiple sample rows can help determine the necessary cache size for desired store performance.

- -json specifies a sample row used for the calculation.
- -rows specifies the number of rows to use for the cache size calculation
- Use the -index or -primarykey and -keyprefix to specify the expected commonality of index keys in terms of number of bytes.

This command mainly does the following:

1. Calculates the key and data size based on the input row in JSON format.
2. Estimates the DB Cache size required for a specified number of rows in the same JSON format.

The output contains both detailed size info for primary key/index and the total size; internally it calls JE's DbCacheSize utility to calculate the cache size required for primary key and indexes with the input parameters:

```
java -jar $KVHOME/dist/lib/je.jar DbCacheSize
-records <num> -key <size> -data <size> -keyprefix
<size> -outputproperties -replicated <JE properties...>
-duplicates]
```

where:

- -records <num>: The number of rows specified by -row <num>.
- -key <size>: The size of key get from step 1.
- -data <size>: The size of data get from step1.
- -keyprefix <size>: The expected commonality of keys, specified using -primarykey | -index <name> -keyprefix <size>
- -duplicates: Used only for table index.
- -<JE properties...>: The JE configuration parameters used in kvstore.

For example:

```
kv-> execute "create table user (id integer, address string,
zip_code string, primary key(id))"
kv-> execute "create index idx1 on user (zip_code)"
```

See the following cases:

1. Calculates the key size and data size based on the input row in JSON.

```
kv-> table-size -name user -json '{"id":1,
"address": "Oracle Building ZPark BeiJing China",
"zip_code":"100000"}'

=== Key and Data Size ===

      Name         Number of Bytes
-----------------  ---------------
Primary Key               8
Data                     47
Index Key of idx1         7
```

2. Calculates the key/data size and the cache size of the table with 10000 rows.

```
kv-> table-size -name user -json '{"id":1,
"address": "Oracle Building ZPark BeiJing China",
"zip_code":"100000"}'
-rows 10000
=== Key and Data Size ===

      Name         Number of Bytes
-----------------  ---------------
Primary Key               8
```

```
Data                     47
Index Key of idx1         7

=== Environment Cache Overhead ===

16,798,797 minimum bytes

=== Database Cache Sizes ===

Name   Number of Bytes             Description
-----  ---------------  ----------------------------------
             1,024,690  Internal nodes only
Table        1,024,690  Internal nodes and record versions
             1,024,690  Internal nodes and leaf nodes
-----  ---------------  ----------------------------------
               413,728  Internal nodes only
idx1           413,728  Internal nodes and record versions
               413,728  Internal nodes and leaf nodes
-----  ---------------  ----------------------------------
             1,438,418  Internal nodes only
Total        1,438,418  Internal nodes and record versions
             1,438,418  Internal nodes and leaf nodes
```

For more information, see the DbCacheSize javadoc.

Note:

The cache size is calculated in the following way:

- Cache size of table

```
java -jar KVHOME/lib/je.jar DbCacheSize -records
   10000 key 8 -data 47 -outputproperties -replicated
   <JE properties...>
```

  The parameters are:

  – Record number: 10000
  – Primary key size: 8
  – Data size: 47

- Cache size of table

```
 java -jar KVHOME/lib/je.jar DbCacheSize -records
   10000 -key 7 -data 8 -outputproperties -replicated
   <JE properties...> -duplicates
```

  The parameters are:

  – Record number: 10000
  – Index key size: 7
  – Data size: 8. The primary key size is used here, since the data of secondary index is the primary key.
  – Use -duplicates for index.

- Total size = cache size of table + cache size of idx1.

3. Calculates the cache size with a key prefix size for idx1

```
kv-> table-size -name user -json
'{"id":1, "address":"Oracle Building ZPark BeiJing China",
"zip_code":"100000"}' -rows 10000 -index idx1 -keyprefix 3

=== Key and Data Size ===


      Name         Number of Bytes
-----------------  ---------------
Primary Key                 8
Data                       47
Index Key of idx1           7

=== Environment Cache Overhead ===

16,798,797 minimum bytes

=== Database Cache Sizes ===
```

```
Name   Number of Bytes             Description
-----  ---------------  ----------------------------------
             1,024,690  Internal nodes only
Table        1,024,690  Internal nodes and record versions
             1,024,690  Internal nodes and leaf nodes
-----  ---------------  ----------------------------------
               413,691  Internal nodes only
idx1           413,691  Internal nodes and record versions
               413,691  Internal nodes and leaf nodes
-----  ---------------  ----------------------------------
             1,438,381  Internal nodes only
Total        1,438,381  Internal nodes and record versions
             1,438,381  Internal nodes and leaf nodes
```

For more information, see the DbCacheSize javadoc.

> **Note:**
>
> A key prefix size is provided for idx1, the idx1's cache size is calculated like this:
>
> ```
> java -jar KVHOME/lib/je.jar DbCacheSize -records
> 10000 -key 7 -data 8 -keyprefix 3 -outputproperties
> -replicated <JE properties...> -duplicates
> ```
>
> The above examples show that the cache size of idx1 is 413,691 and is smaller than 413,728 of case 2. For more information about the usage of keyprefix, see JE DbCacheSize document.

timer

```
timer [on|off]
```

Turns the measurement and display of execution time for commands on or off.

topology

Encapsulates commands that manipulate store topologies. Examples are redistribution/rebalancing of nodes or changing replication factor. Topologies are created and modified using this command. They are then deployed by using the `plan deploy-topology` command. For more information, see plan deploy-topology. The subcommands are as follows:

- topology change-repfactor
- topology change-zone-arbiters
- topology change-zone-type
- topology clone
- topology contract

- topology create
- topology delete
- topology list
- topology preview
- topology rebalance
- topology redistribute
- topology validate
- topology view

topology change-repfactor

```
topology change-repfactor -name <name>  -pool <pool name>
    -zn <id> | -znname <name> -rf <replication factor>
```

Modifies the topology to change the replication factor of the specified zone to a new value. The replication factor may be decreased for secondary zones, but decreasing it for primary zones is not currently supported.

When increasing the replication factor, the command may create Replication Nodes or Arbiter Nodes and may remove Arbiter Nodes only in the zone specified in the command. If the change in replication factor increases the total primary replication factor equal to two and the zone is configured to allow Arbiters, then Arbiters are created in that zone. If the change in replication factor increases the total primary replication factor from two to a number greater than two and if the zone contained Arbiters, then the Arbiters are removed from the zone. If some other zone contained Arbiters, a topology rebalance must be performed to remove the Arbiters from the topology.

For more information on increasing the replication factor, see Increase Replication Factor.

When decreasing the replication factor for a secondary zone, the command will remove the replication nodes from the zone.

If you want to remove a secondary zone, then the replication factor for that secondary zone should be reduced to zero.

After reducing the replication factor to zero, do the following steps to remove the secondary zone:

1. Remove any admins in the zone using `plan remove-admin` command
2. Remove the Storage Nodes in the zone using `plan remove-sn` command
3. Remove the zone using `plan remove-zone` command

topology change-zone-arbiters

```
topology change-zone-arbiters -name <name>
    {-zn <id> | -znname <name>}  {-arbiter | -no-arbiter}
```

Modifies the topology to change the Arbiter Node attribute of the specified zone.

topology change-zone-master-affinity

```
topology change-zone-master-affinity -name <name>
    -zn <{-no-master-affinity | -master-affinity}
```

Modifies the topology of the existing specified zone to `-no-master-affinity`, or to `-master-affinity`. For example:

```
topology change-zone-master-affinity -name new-topo -zn zn1 -no-master-
affinity
```

Use this command after initially deploying a topology (`plan deploy-zone`).

topology change-zone-type

```
topology change-zone-type -name <name>
    {-zn <id> | -znname <name>} -type {primary | secondary}
```

Modifies the topology to change the type of the specified zone to a new type.

If one or more zones have their type changed and the resulting topology is deployed using the plan deploy-topology command, the following rules apply:

- The plan waits for up to five minutes for secondary nodes that are being converted to primary nodes to catch up with their masters.
- The plan will fail, and print details about lagging zones and nodes, if a quorum of secondary nodes in each shard fails to catch up within the required amount of time. This behavior helps to reduce the time that a newly added primary node cannot become a master, and so is not able to contribute to availability.
- Because this command can only be performed successfully if quorum can be maintained, it does not result in data loss.

topology clone

```
topology clone -from <from topology> -name <to topology>
```

or

```
topology clone -current -name <to topology>
```

Clones an existing topology so as to create a new candidate topology to be used for topology change operations.

topology contract

```
topology contract -name <name> -pool <pool name>
```

Modifies the named topology to contract storage nodes. For more information, see Contracting a Topology.

topology create

```
topology create -name <candidate name> -pool <pool name> [-json]
    -partitions <num>
```

Creates a new topology with the specified number of partitions using the specified storage pool.

You should avoid using the dollar sign ('$') character in topology candidate names. The CLI displays a warning when trying to create or clone topologies whose names contain the reserved character.

If the primary replication factor is equal to two, the `topology create` command will allocate Arbiter Nodes on the Storage Nodes in a zone that supports hosting Arbiter Nodes. During topology deployment, an error is issued if there are not enough Storage Nodes for Arbiter Node distribution. A valid Arbiter Node distribution is one in which the Arbiter Node is hosted on a Storage Node that does not contain other members of its Replication Group.

For more information on creating the first topology candidate, see Make the Topology Candidate.

```
kv-> topology create -name mytopo -pool snpool -json -partitions 20
{
    "operation" : "topology create",
    "returnCode" : 5000,
    "description" : "Operation ends successfully",
    "returnValue" : {
        "store" : "mystore",
        "numPartitions" : 20,
        "sequence" : 32,
        "zone" : [ {
        "id" : "zn1",
        "name" : "1",
        "repfactor" : 1,
        "type" : "PRIMARY"
        }, {
    "id" : "zn2",
    "name" : "2",
    "repfactor" : 1,
    "type" : "PRIMARY"
    }, {
    "id" : "zn3",
    "name" : "3",
    "repfactor" : 1,
    "type" : "PRIMARY"
    } ],
    "sns" : [ {
    "id" : "sn1",
    "zone_id" : "zn1",
    "host" : "localhost",
    "port" : 20000,
```

```
    "capacity" : 1,
    "rns" : [ "rg1-rn1" ],
    "ans" : [ ]
    }, {
    "id" : "sn2",
    "zone_id" : "zn2",
    "host" : "localhost",
    "port" : 21000,
    "capacity" : 1,
    "rns" : [ "rg1-rn2" ],
    "ans" : [ ]
    }, {
    "id" : "sn3",
    "zone_id" : "zn3",
    "host" : "localhost",
    "port" : 22000,
    "capacity" : 1,
    "rns" : [ "rg1-rn3" ],
    "ans" : [ ]
    } ],
    "shards" : [ {
        "id" : "rg1",
        "numPartitions" : 20,
        "rns" : [ "rg1-rn1", "rg1-rn2", "rg1-rn3" ],
        "ans" : [ ]
        } ],
    "name" : "mytopo"
    }
}
```

topology delete

```
topology delete -name <name>
```

Deletes a topology.

topology list

```
topology list
```

Lists existing topologies.

topology preview

```
topology preview -name <name> [-start <from topology>]
```

Describes the actions that would be taken to transition from the starting topology to the named, target topology. If -start is not specified, the current topology is used. This command should be used before deploying a new topology.

topology rebalance

```
topology rebalance -name <name> -pool <pool name>
    [-zn <id> | -znname <name>]
```

Modifies the named topology to create a balanced topology. If the optional `-zn` flag is used, only Storage Nodes from the specified zone are used for the operation.

This command may also add, move or remove Arbiter Nodes. Arbiter Nodes are added if the new topology supports Arbiter Nodes and the old topology does not. Arbiter Nodes are removed if the old topology supported Arbiter Nodes and the new one does not. Arbiter Nodes may be moved to a zero Replication Factor datacenter if the Arbiter Nodes are hosted in a non zero Replication Factor datacenter.

For more information on balancing a non-compliant topology, see Balance a Non-Compliant Topology.

topology redistribute

```
topology redistribute -name <name> -pool <pool name>
```

Modifies the named topology to redistribute resources to more efficiently use those available.

For more information on redistributing resources to enhance write throughput, see Increase Data Distribution.

topology validate

```
topology validate [-name <name>]
```

Validates the specified topology. If no topology is specified, the current topology is validated. Validation generates violations and notes.

Violations are issues that can cause problems and should be investigated.

Notes are informational and highlight configuration oddities that can be potential issues or may be expected.

For more information, see Validate the Topology Candidate.

topology view

```
topology view -name <name>
```

Displays details of the specified topology. Also displays any available Arbiter Node information.

verbose

```
verbose [on|off]
```

Toggles or sets the global verbosity setting. This property can also be set on a per-command basis using the `-verbose` flag.

verify

Encapsulates commands to check various store parameters. Specify one of the subcommands, optionally with `-silent` or `-json`:

```
verify {configuration | prerequisite | upgrade} [-silent] [-json]
```

- verify configuration
- verify prerequisite
- verify upgrade

Invoking `verify` without a subcommand or flag, the returns a deprecated message:

```
kv-> verify
The command:

        verify [-silent]

is deprecated and has been replaced by:

        verify configuration [-silent]
```

verify configuration

```
verify configuration [-silent] [-json]
```

Verifies the store configuration by iterating over components and checking their state against what the Admin database contains. On a large store, this command can be time consuming.

The `-json` option specifies that the command display all output in JSON format.

The `-silent` option suppresses verbose verification messages as verification proceeds. Using the `-silent` option displays only the initial startup messages and the final verification message. This option has no effect when the `-json` option is specified.

In some situations, the `verify configuration` command can generate *violations* and *notes*. For example, if:

- The disk reaches a limit exception.
- The available storage size is less than 5 GB.
- The shard has no partitions.

- A replication node or a storage node is not running.

verify prerequisite

```
verify prerequisite [-silent] [-sn snX]*
```

Verifies that the storage nodes are at or above the prerequisite software version needed to upgrade to the current version. This call may take a while on a large store.

As part of the verification process, this command displays the components which do not meet the prerequisites or cannot be contacted. It also checks for illegal downgrade situations where the installed software is of a newer minor release than the current version.

When using this command, the current version is the version of the software running the command line interface.

Use the `-sn` option to specify those storage nodes that you want to verify. If no storage nodes are specified, all the nodes in the store are checked.

The `-silent` option suppresses verbose verification messages that are displayed as the verification is proceeding. Instead, only the initial startup messages and the final verification message is displayed.

verify upgrade

```
verify upgrade [-silent] [-sn snX]*
```

Verifies the storage nodes (and their managed components) are at or above the current version. This call may take a while on a large store.

As part of the verification process, this command displays the components which have not yet been upgraded or cannot be contacted.

When using this command, the current version is the version of the software running the command line interface.

Use the `-sn` option to specify those storage nodes that you want to verify. If no storage nodes are specified, all the nodes in the store are checked.

The `-silent` option suppresses verbose verification messages that are displayed as the verification is proceeding. Instead, only the initial startup messages and the final verification message is displayed.

Admin Utility Command Reference

This appendix describes the following Admin utility commands:

- diagnostics
- generateconfig
- help
- kvlite
- load admin metadata

- makebootconfig
- ping
- restart
- runadmin
- securityconfig
- start
- status
- stop
- version
- xrstart
- xrstop

Oracle NoSQL Database utility commands are stand-alone utilities that do not require the use of the Oracle NoSQL Database Command Line Interface. They are available using one of two jar files. In some cases, kvstore.jar is used. In others, kvtool.jar is required. Both are packaged with the server libraries.

diagnostics

You can troubleshoot your KVStore using the diagnostics tool. You should first run the `diagnostics setup` command in order to set up the tool. You can then use the `diagnostics collect` command to package important information and files to be able to send them to Oracle Support. You can use the `diagnostics verify` command to verify the configuration of the specified Storage Nodes.

For all details on using the diagnostics tool to troubleshoot your KVStore, see Diagnostics Utility.

generateconfig

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar generateconfig [-verbose]
-host <hostname> -port <port>
 -sn <StorageNodeId> -target <zipfile>
[-username <user >]
[-security <security-file-path>]
 [-secdir <overriden security directory>]
```

This command generates configuration files for any Storage Node identifier (value of "sn" parameter) specified in the command.

Parameter	Required	Default value	Description
`host`	Yes		The host name of the Storage Node for which the config file is generated.

Parameter	Required	Default value	Description
`port`	Yes		The registry port of the Storage Node for which the config file is generated.

Note: The user can use the Admin CLI ping command, to get

Parameter	Required	Default value	Description
			the registry port of any Storage Node.

Parameter	Required	Default value	Description
`sn`	Yes		Identifier of the Storage Node.

Note: The user can use the Admin CLI ping command, to get th

Parameter	Required	Default value	Description
			eStorageNodeIdentifierofanyStorageNode.
`target`	Yes		Full path of the zip file to be created.
`username`	No		The name of the user to log in to the secured store. This parameter is only required if your store is configured to require authentication.

Parameter	Required	Default value	Description
security	No		The client security configuration file. This parameter is only required if your store is secure. A fully qualified path to a file containing security information can be specified.
secdir	No	security	The name of the directory within the KVROOT that will hold the security configuration.

help

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar help <commandName>
```

Prints usage info. With no arguments the top-level shell commands are listed. With a command name, additional detail is provided.

kvlite

KVLite is a simplified version of the Oracle NoSQL Database. It provides a single storage node, single shard store, that is not replicated. It runs in a single process without requiring any administrative interface.

Oracle NoSQL Database can be configured securely. In a secure configuration, network communications between NoSQL clients, utilities, and NoSQL server components are encrypted using SSL/TLS, and all processes must authenticate themselves to the components to which they connect.

Start KVLite in secure mode:

KVLite starts in a secure mode by default.

```
java -Xmx64m -Xms64m -jar lib/kvstore.jar kvlite
```

Start KVLite in non-secure mode:

Execute the kvstore.jar file using the `secure-config disable` flag to disable security and start KVLite in non-secure mode.

```
java -Xmx64m -Xms64m -jar lib/kvstore.jar kvlite -secure-config disable
```

To stop KVLite, use Ctrl C (^C) from within the shell where KVLite is running.

To restart the process, simply run the KVLite utility without any command line options and run the command from the original directory, or specify that directory using the -root flag. Do this even if you provided non-standard options when you first started

KVLite. This is because KVLite remembers information such as the port value and the store name in between runs.

load admin metadata

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar load -store <storeName>
-host <hostname> -port <port> -load-admin
-source <admin-backup-dir> [-force]
[-username <user>] [-security <security-file-path>]
```

Loads the admin metadata from the snapshot to the new store. In this case the `-source` directory must point to the environment directory of the admin node from the snapshot. The store must not be available for use by users at the time of this operation.

where:

- `-load-admin` Specifies that only admin metadata will be loaded into the store.

> **Note:**
>
> This option should not be used on a store unless that store is being restored from scratch. If `-force` is specified in conjunction with `-load-admin`, any existing metadata in the store, including tables and security metadata, will be overwritten. See Using the Load Program for more information.

- `-host <hostname>` Identifies the host name of a node in your store.
- `-port <port>` Identifies the registry port in use by the store's Node.
- `-security <security-file-path>` Identifies the security file used to specify properties for login.
- `-source <admin-backup-dir>` The admin snapshot directory containing the contents of the admin metadata that is to be loaded into the store.
- `-store <storeName>` Identifies the new store which is the target of the load.
- `-username <user>` Identifies the name of the user to login to the secured store.

load store data

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar load [-verbose]
-store <storeName> -host <hostname> -port <port>
-source <shard-backup-dir>[, <shard-backup-dir>]*
[-checkpoint <checkpoint-files-directory>]
[-username <user>] [-security <security-file-path>]
```

Loads data into a store from backup directories. The bulk put API is used by this utility to load data into the target store. To recreate the complete contents of the store, you must specify one directory per shard for each shard associated with the store.

The load utility is highly parallelized. To further boost load performance, you can choose to run multiple concurrent invocations of the load utility on different machines, and assign each invocation a non-overlapping subset of the shard directories, using the `-source` argument. The use of these additional machine resources could significantly decrease overall elapsed load times.

> **Note:**
>
> Creating multiple processes on the same machine is unlikely to be beneficial and could be detrimental, since the two processes are likely to be contending for the same CPU and network resources.

where:

- `-checkpoint <checkpoint-files-directory>` The utility used this directory to checkpoint its progress on a periodic basis. If the load process is interrupted for some reason, the progress checkpoint information is used to skip data that had already been loaded when the load utility is subsequently re-executed with the same arguments. If the `-checkpoint` flag is not specified, progress will not be checkpointed and all the data in the partitions that were already loaded will be reread.
- `-host <hostname>` Identifies the host name of a node in your store.
- `-port <port>` Identifies the registry port in use by the store's node.
- `-security <security-file-path>` Identifies the security file used to specify properties for login.
- `-source <shard-backup-dir>[,<shard-backup-dir>]*` These backup directories typically represent the contents of snapshots created using the snapshot commands described at Taking a Snapshot.
- `-store <storeName>` Identifies the name of the store.
- `-username <user>` Identifies the name of the user to login to the secured store.

makebootconfig

```
java -Xmx64m -Xms64m
-jar KVHOME/lib/kvstore.jar makebootconfig [-verbose]
-root <rootDirectory> -host <hostname> -harange <startPort,endPort>
-port <port> [-config <configFile>]
[-store-security <none | configure | enable> ]
[-noadmin]
[-admindir <directory path>]
[-admindirsize <directory size>]
[-storagedir <directory path>]
[-storagedirsize <directory size>]
[-rnlogdir <directory path>]
[-capacity <n_rep_nodes>]
[-num_cpus <ncpus>][-memory_mb <memory_mb>]
[-servicerange <startPort,endPort>]
[-admin-web-port <admin web service port>]
```

```
[-hahost <haHostname>]
[-secdir <security dir>] [-pwdmgr {pwdfile | wallet | <class-name>}]
[-kspwd <password>]
[-external-auth {kerberos}]
  [-krb-conf <kerberos configuration>]
  [-kadmin-path <kadmin utility path>]
  [-instance-name <database instance name>]
  [-admin-principal <kerberos admin principal name>]
  [-kadmin-keytab <keytab file>]
  [-kadmin-ccache <credential cache file>]
  [-princ-conf-param <param=value>]*
[-security-param <param=value>]*
[-mgmt {jmx|none}]
[-dns-cachettl <time in sec>]
      [-force]
```

where:

- `-capacity <n_rep_nodes>` The total number of Replication Nodes a Storage Node can support. The value defaults to "1".

  If capacity is set to 0, then this Storage Node may be used to host Arbiter Nodes.
- `-config <configFile>` Only specified if more than one Storage Node Agent process will share the same root directory. This value defaults to `config.xml`.
- `-dns-cachettl <time in sec>` Specifies the number of seconds that Replication Nodes should cache host name to IP address mappings. The default value is -1, which means mappings should be cached indefinitely. A value of 0 means mappings should not be cached. The value of this flag is used to set the `networkaddress.cache.ttl` and `networkaddress.cache.negative.ttl` security properties.
- `-external-auth {kerberos}` Specifies Kerberos as an external authentication service. If no keytab or credential cache has been specified on the command line, an interactive version of the `securityconfig` utility will run.

  This flag is only permitted when the value of the `-store-security` flag is specified as configure or enable.

  To remove Kerberos authentication from a running store, set the value of the `userExternalAuth` security.xml parameter to `NONE`.

  For more information on Kerberos, see Kerberos Authentication Service in the *Security Guide*.

  where `-external-auth` can have the following flags:

  - `-admin-principal <kerberos admin principal name>`

    Specifies the principal used to login to the Kerberos admin interface. This is required while using kadmin keytab or password to connect to the admin interface.
  - `-kadmin-ccache <credential cache file>`

    Specifies the complete path name to the Kerberos credentials cache file that should contain a service ticket for the kadmin/ADMINHOST. `ADMINHOST` is the fully-qualified hostname of the admin server or kadmin/admin service.

If not specified, the user is prompted to enter the password for principal while logging to the Kerberos admin interface. This flag cannot be specified in conjunction with the `-kadmin-keytab` flag.

– `-kadmin-keytab <keytab file>`

  Specifies the location of a Kerberos keytab file that stores Kerberos admin user principals and encrypted keys. The security configuration tool will use the specified keytab file to login to the Kerberos admin interface.

  The default location of the keytab file is specified by the Kerberos configuration file. If the keytab is not specified there, then the system looks for the file `user.home/krb5.keytab`.

  You need to specify the `-admin-principal` flag when using keytab to login to the Kerberos admin, otherwise the correct admin principal will not be recognized. This flag cannot be specified in conjunction with the `-kadmin-ccache` flag.

– `-kadmin-path <kadmin utility path>`

  Indicates the absolute path of the Kerberos kadmin utility. The default value is `/usr/kerberos/sbin/kadmin`.

– `-krb-conf <kerberos configuration>`

  Specifies the location of the Kerberos configuration file that contains the default realm and KDC information. If not specified, the default value is `/etc/krb5.conf`.

– `-princ-conf-param <param=value>*`

  A repeatable argument that allows configuration defaults to be overridden.

  Use the `krbPrincValidity` parameter to specify the expiration date of the Oracle NoSQL Database Kerberos service principal.

  Use the `krbPrincPwdExpire` parameter to specify the password expiration date of the Oracle NoSQL Database Kerberos service principal.

  Use the `krbKeysalt` parameter to specify the list of encryption types and salt types to be used for any new keys created.

- `-force` Optionally specified to force generating the boot configuration files even if boot config verification finds any invalid parameters.
- `-hahost <haHostname>` Can be used to specify a separate network interface for store replication traffic. This defaults to the hostname specified using the -host flag.

  The host name specified here must be resolvable using DNS or the `/etc/hosts` file on any machine running client code that wants to connect to the node.
- `-harange <startPort,endPort>` A range of free ports that the Replication Nodes and Admins use to communicate among themselves. These ports should be sequential. You must assign at least as many ports as the specified capacity for this node, plus an additional port if the node hosts an Admin.
- `-host <hostname>` Identifies a host name associated with the node on which the command is run. This hostname identifies the network interface used for communication with this node.

The host name specified here must be resolvable using DNS or the `/etc/hosts` file on any machine running client code that wants to connect to the node.

- `-kspwd<password>` For script-based configuration you can use this option to allow tools to specify the keystore password on the command line. If it is not specified, the user is prompted to enter the password.
- `-memory_mb <memory_mb>` The total number of megabytes of memory available in the machine. If the value is 0, the store attempts to determine the amount of memory on the machine, but the value is only available when the JVM used is the Oracle Hotspot JVM. The default value is "0".

  For best results, do not specify this parameter. Oracle NoSQL Database will determine the proper value by default. This parameter should be used sparingly, and only for exceptional situations.
- `-num_cpus <ncpus>` The total number of processors on the machine available to the Replication Nodes. If the value is 0, the system attempts to query the Storage Node to determine the number of processors on the machine. This value defaults to "0".

  For best results, do not specify this parameter. Oracle NoSQL Database will determine the proper value by default. This parameter should be used sparingly, and only for exceptional situations.
- `-port <port>` The TCP/IP port on which Oracle NoSQL Database should be contacted. Sometimes referred to as the registry port. This port must be free on the node on which this command is run.
- `-pwdmgr [ pwdfile | wallet ]`

  Indicates the password manager mechanism used to hold passwords that are needed for access to keystores, and so on.

  where `-pwdmgr` has the following options:

  - `-pwdmgr pwdfile`

    Indicates that the password store is a read-protected clear-text password file. This is the only available option for Oracle NoSQL Database CE deployments. You can specify an alternate implementation.
  - `-pwdmgr wallet`

    Specifies Oracle Wallet as the password storage mechanism. This option is only available in the Oracle NoSQL Database EE version.
- `-root <rootDirectory>` Identifies where the root directory should reside.
- `-secdir <security dir>`

  Specifies the name of the directory within the KVROOT that will hold the security configuration. This must be specified as a name relative to the specified secroot. If not specified, the default value is `security`.
- `-security-param <param=value>*`

  A repeatable argument that allows configuration defaults to be overridden.

  Use the `krbServiceName` parameter to specify the service name of the Oracle NoSQL Database Kerberos service principal.

  Use the `krbServiceKeytab` parameter to specify the keytab file name in security directory of the Oracle NoSQL Database Kerberos service principal.

- `-servicerange <startPort,endPort>` A range of ports that may be used for communication among administrative services running on a Storage Node and its managed services. This parameter is optional and is useful when services on a Storage Node must use specific ports for firewall or other security reasons. By default the services use anonymous ports. The format of the value string is "startPort,endPort."
- `-admin-web-port <admin web service port>` The TCP/IP port on which the admin web service should be started. If not specified, the default port value is `-1`. If a positive integer number is not specified for `-admin-web-port`, then admin web service does not start up along with the admin service. See REST API for Administering Oracle NoSQL Database.
- `-noadmin` Specifies to disable the bootstrap admin service for SNA.
- `-admindir <path>` Specify a path to the directory to be used to store the environment associated with an Admin Node. If no directory is specified, Admin Nodes use a directory under the root directory.
- `-admindirsize <directory size>` Specify the size of the admin storage directory identified by `-admindir`. This parameter is optional. See Managing Admin Directory Size.

  The value specified for this parameter must be a long, followed optionally by a unit string. Accepted unit strings are: KB, MB, and GB, corresponding to 1024, 1024^2, and 1024^3 respectively. Acceptable strings are case insensitive. Valid delimiters between the long value and the unit string are " ", "-", or "_". If you specify the delimiter as " ", your value should be enclosed in double quotes.

  For example:

  ```
  -admindirsize "200 MB"
  -admindirsize 1_gb
  -admindirsize 3000-Mb
  ```

- `-storagedir <path>` Specifies a path to the directory that a Replication Node will use for storage. If your Storage Node will host more than one (1) replication node, specify this argument once for each Replication Node, being sure that the number of arguments does not exceed the Storage Node capacity.

  If you do not specify a storage directory explicitly, Replication Nodes use a directory under the root directory. Be sure to match the number of `-storagedir` arguments to the value of the capacity argument. For example, if your Storage Node hosts four disks, and you are using one disk for each replication node, specify a capacity of four, and have four `-storagedir` arguments, each with a corresponding `-storagedirsize <directory size>` value.
- `-storagedirsize <directory size>` Specifies the size of the directory identified by each `-storagedir` argument. While this parameter is optional, we strongly recommend that you specify its value, since the system takes the `-storagedirsize <directory size>` into consideration when determining store topology. For example, if you have some Storage Nodes each with smaller disk capacity than other store SNs, the system arranges to store less data on those SNs by adjusting partition distribution to shards to match the storage capacity. See Managing Storage Directory Sizes for details.

  Further, it is an error to specify the `-storagedirsize <directory size>` parameter for some named storage directories, but not all.

Specify the `-storagedirsize <directory size>` value as a long, optionally followed by a unit string. The accepted unit strings are: KB, MB, GB, and TB, corresponding to 1024, 1024^2, 1024^3, 1024^4, respectively. Acceptable strings are case insensitive. Valid delimiter characters between the long value and the unit string are " ", "-", or "_". If you specify the delimiter as " ", your value should be enclosed in double quotes.

For example:

```
-storagedirsize "200 MB"
-storagedirsize 4_tb
-storagedirsize 5000-Mb
```

> **Note:**
>
> If you specify the `-storagedir` parameter, but not `-storagedirsize`, `makebootconfig` displays a warning. We strongly recommend specifying both parameters.

- `-storageDirStorageType [hard drive | SSD | NVMe]` Specifies the type of disk on which storage directories reside.
- `-rnlogdir <path>` Specify a path to the directory to be used for storing the Replication Node log files. This flag may be used more than once in the command to specify multiple Replication Node log directories, but the number should not exceed the capacity for the node.

  If no directory is specified, by default, the logs are stored under the root directory.
- `-store-security [none | configure | enable]` Specifies if security will be used or not. If `-store-security none` is specified, no security will be in use. If `-store-security configure` is specified, security will be used, and the makebootconfig process invokes the security configuration utility as part processing. If `-store-security enable` is specified, security will be used. You will need to configure security either by utilizing the security configuration utility or by copying a previously created configuration from another system.

> **Note:**
>
> The `-store-security` command is optional. Even if the user does not specify `-store-security`, security is enabled by default. The user must run `securityconfig` utility to create the security folder before starting up the storage node agent.

- `-mgmt {jmx|none}`
  Specifies the type of monitoring to be enabled for the Storage Node . This parameter is optional. The default value is none when monitoring is disabled. Use this parameter to make Java Management Extensions (JMX) agents available for monitoring.

  If you specify `jmx`, JMX interfaces will be used for monitoring the Storage Node and any NoSQL components like Replication Nodes, Admin Node and Storage Node Agent hosted on that Storage Node. JMX agents in Oracle NoSQL Database are read-only interfaces. These interfaces let you poll a Storage Node for information about the Storage Node and about any Replication Nodes or Admins that the Storage Node hosts. The

information available from polling includes the service status (RUNNING, STOPPED, UNREACHABLE etc.), operational parameters, and performance metrics. Also, JMX can be used to monitor Arbiter Nodes.

JMX agents also deliver event traps and notifications for particular events. For example, JMX sends notifications for every service status state change, and any performance limits that the store exceeds. You can get the total number of operation requests using the metric `TotalReq` and the metric `TotalOps` gives the total number of records returned or processed. See Monitoring for Storage Nodes (SN) for the definitions of the events available for monitoring .

Creates a configuration file used to start a not-yet-deployed Storage Node to be used in an instance of Oracle NoSQL Database. The file cannot pre-exist. To create the initial "boot config" file used to configure the installation see Installation Configuration Parameters.

You can change parameters after setting them with the `makebootconfig` utility. The commands to use are `change-policy –params` and `plan change-parameters -params`. Changing parameters may require restarting a node. For more information, see CLI Command Reference.

ping

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar ping [-verbose] [-json] [-shard <shardId>]
-host <hostname> -port <port> or
-helper-hosts  <host:port>[,host:port]*>
-username <user>
-security <security-file-path>
```

Attempts to contact a store to get status of running services. This utility provides both a concise summary of the health of a store, as well as detailed information about the topology of the store. It can signal a red/yellow/green status, to let you know whether the store is in full health, whether the store has experienced some failures but is operational, or whether the store has critical problems. `ping` uses the nodes specified by the `-helper-hosts` or `-host/-port` arguments to locate topology metadata describing the store. Using that topology, `ping` contacts all the RNs, SNs, Arbiters, and Admin services associated with a store. You can also indicate a specific shard to return its status information.

Specify the `-helper-hosts` flag as an alternative to the existing `-host` and `-port` flags. If multiple helper hosts are in use, this utility has multiple nodes it can use to make an initial point of contact with the store, and will have a greater chance of success if some nodes of the store are unavailable.

Specify `-shard <shardId>` to return a subset of information.

Ping Command Line Parameters

The `ping` utility's command line parameters are:

- `-host` identifies the name of a specific host in the store. Use this option to check whether the SNA on that particular host can be contacted.

If this parameter is specified, then `-port` must also be specified. Further, if the `-host` and `-port` parameters are specified, then the `-helper-hosts` must not be specified.

- `-port` identifies the listening port for a specific host in the store. Use this parameter only if you are also using the `-host` parameter.
- `-helper-hosts` identifies a comma-separated list of one or more host:port pairs in the store. Use this parameter to check the health of the entire store.

  Using the `-helper-hosts` parameter precludes specifying the `-host` and `-port` flags.

  If multiple helper hosts are provided, this utility has multiple nodes it can use to make an initial point of contact with the store, and thus a greater chance of success if some nodes of the store are unavailable. For example:

  ```
  -helper-hosts hst1:5000,hst2:5100, hst3:5100
  ```

- `-username` is the name of the user that you want to ping the store as. This parameter is required if your store is configured to require authentication. This user should have at least `SYSVIEW` access to the store. The built-in `dbadmin` role is sufficient.
- `-security` is the client security configuration file. This parameter is required if your store is configured to require authentication. For information on the parameters contained in this file, see Configuring SSL in the *Java Direct Driver Developer's Guide*. For example:

  ```
  oracle.kv.auth.username=clientUID1
  oracle.kv.auth.pwdfile.file=/home/nosql/login.pwd
  oracle.kv.transport=ssl
  oracle.kv.ssl.trustStore=/home/nosql/client.trust
  ```

  If you are using Kerberos, then this file would look something like this:

  ```
  oracle.kv.auth.kerberos.keytab = kerberos/mykeytab
  oracle.kv.auth.username = krbuser@EXAMPLE.COM
  oracle.kv.auth.external.mechanism=kerberos
  oracle.kv.auth.kerberos.services=
  node01:oraclenosql/node01.example.com@EXAMPLE.COM
  oracle.kv.auth.kerberos.mutualAuth=false
  ```

- `-verbose` is optional. It causes the `ping` utility to provide additional information about the utility's current actions.
- `-json` causes the `ping` utility to write its output in JSON format.
- `-shard <shardId>` is optional and returns a subset of status information about the specific shard ID you supply. .

  For example:

  ```
  bash-4.1$ java -jar $KVHOME/lib/kvstore.jar ping -host
  mynode.mycompany.com
  -port 5000 -shard rg2 Pinging components of store mystore based upon
  topology
  sequence #2376    shard rg2
  500 partitions and 3 storage nodes
  Time: 2018-09-28 06:57:10 UTC   Version: 18.3.2
  Shard Status: healthy
  ```

```
Admin Status: healthy
Zone [name=myshardzone id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online:3 offline:0 maxDelayMillis:0 maxCatchupTimeSecs:0
Storage Node [sn10] on nodeA:5000    Zone: [name=myshardzone id=zn1
type=PRIMARY allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 18.3.2 2018-09-17 09:33:45 UTC
Build id: a72484b8b33c Edition: Enterprise
         Rep Node [rg2-rn1]      Status: RUNNING,MASTER
sequenceNumber:71,166
haPort:5010 available storage size:12 GB
Storage Node [sn11] on nodeB:5000    Zone: [name=myshardzone id=zn1
type=PRIMARY allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 18.3.2 2018-09-17 09:33:45 UTC
Build id: a72484b8b33c Edition: Enterprise
         Rep Node [rg2-rn2]      Status: RUNNING,REPLICA
sequenceNumber:71,166
haPort:5011 available storage size:14 GB delayMillis:0
catchupTimeSecs:0
Storage Node [sn12] on nodeC:5000    Zone: [name=myshardzone id=zn1
type=PRIMARY allowArbiters=false masterAffinity=false]
Status: RUNNING   Ver: 18.3.2 2018-09-17 09:33:45 UTC
Build id: a72484b8b33c Edition: Enterprise
         Rep Node [rg2-rn3]      Status: RUNNING,REPLICA
sequenceNumber:71,166
haPort:5012 available storage size:24 GB delayMillis:0
catchupTimeSecs:0
```

Ping Exit Codes

The following exit codes can be returned by this utility. Exit codes can be returned both as a process exit code, and as part of the JSON output.

Name	Code	Description
EXIT_OK	0	All services in the store could be located and are in a known, good state (for example, RUNNING).
EXIT_OPERATIONAL	1	One or more services in the store could not be reached, or are in an unknown or not usable state. In this case the store should support all data operations across all shards, as well as all administrative operations, but may be in a state of degraded performance. Some action should be taken to find and fix the problem before part of the store becomes unavailable.

Name	Code	Description
EXIT_NO_ADMIN_QUORUM	2	The Admin Service replication group does not have quorum or is not available at all, and it is not possible to execute administrative operations which modify store configuration. The store supports all normal data operations despite the loss of admin quorum, but this state requires immediate attention to restore full store capabilities.
EXIT_NO_SHARD_QUORUM	3	One or more of the shards does not have quorum and either cannot accept write requests, or is completely unavailable. This state requires immediate attention to restore store capabilities. The exit code takes precedence over `EXIT_NO_ADMIN_QUORUM`, so if this exit code is used, it is possible that the administrative capabilities are also reduced or unavailable.
EXIT_USAGE	100	Illegal ping command usage.
EXIT_TOPOLOGY_FAILURE	101	`ping` was unable to find a topology in order to operate. This could be a store problem, a network problem, or it could be a usage problem with the parameters passed to ping. For example, the specified `-host`/`-port` pair are not part of the store, or none of the hosts specified on `-helper-hosts` can be contacted.
EXIT_UNEXPECTED	102	The utility has experienced an unexpected error.
EXIT_STATUS_UNKNOWN	103	The store is operational but some Replication Nodes are in unknown state.

Note:

Exit codes 1 through 3 may indicate a network connectivity issue that should be checked first before concluding that any services have a problem.

Ping Report Text Output

By default, the `ping` utility reports store health in human readable format. For example:

> **Note:**
>
> Extra line breaks have been added so that the command output fits in the available space.

```
$ java -Xmx64m -Xms64m -jar <KVHOME>/lib/kvstore.jar ping -host nodeA -
port 1310
Pinging components of store mystore based upon topology sequence #108
100 partitions and 3 storage nodes
Time: 2021-04-08 10:37:43 UTC Version: 21.1.11
Shard Status: healthy:1 writable-degraded:0 read-only:0 offline:0
total:1
Admin Status: healthy
Zone [name=MyDC id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
RN Status: online:3 read-only:0 offline:0 maxDelayMillis:0
maxCatchupTimeSecs:0
Storage Node [sn1] on nodeA:13100
Zone: [name=MyDC id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
Status: RUNNING Ver: 21.1.11 2021-03-30 05:02:01 UTC
Build id: 0ce629097e92 Edition: Enterprise isMasterBalanced:true
        Admin [admin1]         Status: RUNNING,MASTER
        Rep Node [rg1-rn1]     Status: RUNNING,MASTER
        sequenceNumber:227 haPort:13117 available storage size:16 GB
storage type:HD
Storage Node [sn2] on nodeB:13200
    Zone: [name=MyDC id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING Ver: 21.1.11 2021-03-30 05:02:01 UTC
    Build id: 0ce629097e92
        Admin [admin2]         Status: RUNNING,REPLICA
        Rep Node [rg1-rn2]     Status: RUNNING,REPLICA
            sequenceNumber:227 haPort:13217 available storage size:14
GB storage type:HD delayMillis:0
            catchupTimeSecs:0
Storage Node [sn3] on nodeC:13300
    Zone: [name=MyDC id=zn1 type=PRIMARY allowArbiters=false
masterAffinity=false]
    Status: RUNNING Ver: 21.1.11 2021-03-30 05:02:01 UTC
    Build id: 0ce629097e92
        Admin [admin3]         Status: RUNNING,REPLICA
        Rep Node [rg1-rn3]     Status: RUNNING,REPLICA
            sequenceNumber:227 haPort:13317 available storage size:24
GB storage type:HD delayMillis:0
            catchupTimeSecs:0
```

Ping Report JSON Output

When the `-json` command line parameter is specified, this utility provides its report in JSON formatting.

> **Note:**
>
> Extra line breaks have been introduced to allow this output to fit in the available space.

```
bash-3.2$ java -Xmx64m -Xms64m \
-jar dist/lib/kvstore.jar ping -host node01 \
-port 5000 -json
{
  "operation" : "ping",
  "returnCode" : 5000,
  "description" : "No errors found",
  "returnValue" : {
    "topology" : {
      "storeName" : "orcl",
      "sequenceNumber" : 9,
      "numPartitions" : 0,
      "numStorageNodes" : 2,
      "time" : 1539857069504,
      "version" : "18.3.2"
    },
    "adminStatus" : "healthy",
    "shardStatus" : {
      "healthy" : 1,
      "writable-degraded" : 1,
      "read-only" : 0,
      "offline" : 0,
      "total" : 2
    },
    "zoneStatus" : [ {
      "resourceId" : "zn1",
      "name" : "Atlanta",
      "type" : "PRIMARY",
      "allowArbiters" : false,
      "masterAffinity" : false,
      "rnSummaryStatus" : {
        "online" : 2,
        "offline" : 0,
        "read-only" : 0,
        "hasReplicas" : false
      }
    }, {
      "resourceId" : "zn2",
      "name" : "Boston",
      "type" : "SECONDARY",
      "allowArbiters" : false,
```

```
        "masterAffinity" : false,
        "rnSummaryStatus" : {
          "online" : 1,
          "offline" : 0,
          "read-only" : 0,
          "hasReplicas" : true,
          "maxDelayMillis" : 0,
          "maxCatchupTimeSecs" : 0
        }
      } ],
      "snStatus" : [ {
        "resourceId" : "sn1",
        "hostname" : "node01",
        "registryPort" : 5000,
        "zone" : {
          "resourceId" : "zn1",
          "name" : "Atlanta",
          "type" : "PRIMARY",
          "allowArbiters" : false,
          "masterAffinity" : false
        },
        "serviceStatus" : "RUNNING",
        "version" : "18.3.2 2018-09-17 09:33:45 UTC  Build id:
a72484b8b33c Edition: Enterprise",
        "adminStatus" : {
          "resourceId" : "admin1",
          "status" : "RUNNING",
          "state" : "MASTER",
          "authoritativeMaster" : true
        },
        "rnStatus" : [ {
          "resourceId" : "rg1-rn1",
          "status" : "RUNNING",
          "requestsEnabled" : "ALL",
          "state" : "MASTER",
          "authoritativeMaster" : true,
          "sequenceNumber" : 23,
          "haPort" : 5002,
          "availableStorageSize" : "3 GB"
        }, {
          "resourceId" : "rg2-rn1",
          "status" : "RUNNING",
          "requestsEnabled" : "ALL",
          "state" : "MASTER",
          "authoritativeMaster" : true,
          "sequenceNumber" : 23,
          "haPort" : 5003,
          "availableStorageSize" : "3 GB"
        } ],
        "anStatus" : [ ]
      }, {
        "resourceId" : "sn2",
        "hostname" : "node02",
        "registryPort" : 6000,
        "zone" : {
```

```
        "resourceId" : "zn2",
        "name" : "Boston",
        "type" : "SECONDARY",
        "allowArbiters" : false,
        "masterAffinity" : false
      },
      "serviceStatus" : "RUNNING",
      "version" : "18.3.2 2018-09-17 09:33:45 UTC  Build id: a72484b8b33c
Edition: Enterprise",
      "adminStatus" : {
        "resourceId" : "admin2",
        "status" : "RUNNING",
        "state" : "REPLICA"
      },
      "rnStatus" : [ {
        "resourceId" : "rg1-rn2",
        "status" : "RUNNING",
        "requestsEnabled" : "ALL",
        "state" : "REPLICA",
        "sequenceNumber" : 23,
        "haPort" : 6003,
        "availableStorageSize" : "3 GB",
        "networkRestoreUnderway" : false,
        "delayMillis" : 0,
        "catchupTimeSecs" : 0,
        "catchupRateMillisPerMinute" : 0
      } ],
      "anStatus" : [ ]
    } ],
    "exitCode" : 0
  }
}
```


restart

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar restart
[-disable-services] [-verbose]
-root <rootDirectory> [-config <bootstrapFileName>]
```

> **Note:**
>
> Before restarting the SNA, set the environment variable `MALLOC_ARENA_MAX` to `1`. Setting `MALLOC_ARENA_MAX` to `1` ensures that the memory usage is restricted to the specified heap size.

Stops and then starts the Oracle NoSQL Database Storage Node Agent and services related to the root directory.

To disable all services associated with a stopped SNA use the `-disable-services` flag. For more information, see Disabling Storage Node Agent Hosted Services.

runadmin

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar runadmin
-host <hostname> -port <port> | -helper-hosts <host:port[,host:port]*>
[-store <storeName> ]
[-username <user>] [-security <security-file-path>]
[-admin-username <adminUser>]
[-admin-security <admin-security-file-path>]
[-timeout <timeout ms>]
[-consistency <NONE_REQUIRED(default) | ABSOLUTE |
               NONE_REQUIRED_NO_MASTER>]
[-durability <COMMIT_SYNC(default) | COMMIT_NO_SYNC |
              COMMIT_WRITE_NO_SYNC>]
[-dns-cachettl <time in sec>]
[-registry-open-timeout <time in ms>]
[-registry-read-timeout <time in ms>]
```

The `runadmin` command starts the Admin command line interface (CLI) utility on the host Storage Node (SN) of your choice. You use the CLI to perform configuration activities for your store.

You can start the CLI on a single host, using the following flags. You can specify any storage node as a single host, including an Admin-only host without any replica nodes:

```
-host <hostname> -port <port>
```

To have more than one host support the Admin command line interface, use the `-helper-hosts` option with two or more hosts:

```
-helper-hosts <host:port[,host:port]*>
```

> **Note:**
>
> The `runadmin -admin-host <adminHost> -admin-port <adminPort>` options are deprecated. Entering either option results in an error. If you are using these options in scripts, replace them with either the `-host` or `-helper-hosts` options (and their port specifications), as noted in the syntax statement.

Use the `-timeout`, `-consistency`, and `-durability` flags to override the connect configuration settings.

where:

- `-timeout`

  Specifies the store request time-out in milliseconds. There is no default.

- `-consistency`

  Indicates the store request consistency. The default value is `NONE_REQUIRED`.

- `-durability`

  Indicates the store request durability. The default value is `COMMIT_SYNC`.

securityconfig

A KVStore can be configured securely. In a secure configuration, network communications between NoSQL clients, utilities, and NoSQL server components are encrypted using SSL/TLS, and all processes must authenticate themselves to the components to which they connect. To set up security when configuring a KVStore, you need to create an initial security configuration. To do this, run securityconfig tool before, after, or as part of the makebootconfig process. You should not create a security configuration at each node. Instead, you should distribute the initial security configuration across all the Storage Nodes in your store. If the stores do not share a common security configuration they will be unable to communicate with one another.

```
java -Xmx64m -Xms64m -jar lib/kvstore.jar securityconfig
```

Various commands used in the securityconfig tool:

- `config create`
- `config add-security`
- `config verify`
- `config update`
- `config show`
- `config remove-security`

You invoke the `config create` command to create the security configuration.

Use the `config create` command with the `-pwdmgr` option to specify the mechanism used to hold password that is needed for accessing the store. In the example below, Oracle Wallet is used.

```
security-> config create -pwdmgr wallet -root KVROOT
```

Enter a password for your store and then reenter it for verification. The configuration tool will automatically generate some security related files.

For more information on `config create` command, see Creating the security configuration.

Use the `config add-security` command to add the security configuration you just created.

```
security-> config add-security -root KVROOT -secdir security -config
config.xml
```

You can use the `config verify` command to verify the consistency and correctness of the security configuration.

```
security-> config verify -secdir <security dir>
```

You can use the config update command to update the security parameters of a security configuration. You can specify a list of security parameters to update.

```
security-> config update -secdir <security dir> [-param <param=value>]*
```

You can use the config show command to print out all security configuration information.

```
security-> config show -secdir <security dir>
```

If you want to disable security for some reason in an existing installation, you can use the config remove-security command.

```
security-> config remove-security -root <kvroot> [-config >config.xml>]
```

For more information on configuring security using securityconfig tool, see Configuring Security with securityconfig.

start

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar start
[-disable-services] [-verbose]
-root <rootDirectory>
[-config <bootstrapFileName>][-restore-from-snapshot]<snapshot-
time_snapshot-dir-name> [-update-config {true | false} ]
```

Starts the Oracle NoSQL Database Storage Node Agent (and if configured, store) in the root directory.

To disable all services associated with a stopped SNA use the `-disable-services` flag. For more information, see Disabling Storage Node Agent Hosted Services.

You can optionally start from an existing snapshot, instead of using `-config <bootstrapFileName>`.

To start from a snapshot, use the `-restore-from-snapshot` option, followed by the snapshot directory name with its `snapshot-time` prefix. Specify `-update-config true` to override the existing configuration as part of restoring snapshot data.

status

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar status
```

```
-root <rootDirectory> [-config <bootstrapFileName>]
[-verbose] [-disable-services]
```

Attempts to connect to a running Oracle NoSQL Database Storage Node Agent and prints out its status.

For example:

```
java -Xmx64m -Xms64m -jar KVHOME/lib/kvstore.jar  \
status -root KVROOT
SNA Status : RUNNING
```

stop

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar stop
[-disable-services] [-verbose]
-root <rootDirectory> [-config <bootstrapFileName>]
```

Stops the Oracle NoSQL Database Storage Node Agent and services related to the root directory.

To disable all services associated with a stopped SNA use the `-disable-services` flag. For more information, see Disabling Storage Node Agent Hosted Services.

version

```
java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar version
```

Prints version.

xrstart

In a multi-region setup, you must start the XRegion service in each region using the `xrstart` command providing the complete path to the JSON config file. As this service is a long-running process, it is recommended to invoke it as a background process by appending the & at the end of the command. You use the Linux `nohup` command (short for no hang up) to keep the processes running even after exiting the shell or terminal.

> **Note:**
>
> The local KVStore must be started before starting the XRegion Service. If the KVStore in the local region has not started or is not reachable, the XRegion Service will not start.

```
nohup java -Xms256m -Xmx2048m -jar $KVHOME/lib/kvstore.jar xrstart \
-config <complete path to the json.config file> > \
<complete path to the home directory for the XRegion Service>/
nohup.out &
```

Table 7-7 Parameters used in xrstart command

Parameter	Description
`-config`	Specifies the complete path where the `json.config` file is placed.
`>`	Instructs to redirect the output to the file specified next to it.
`nohup.out`	Specifies the file to be used for logging the status messages.

Optionally, you can view the status of the `xrstart` command execution by reading the contents of `nohup.out`.

```
cat <complete path to the home directory for the XRegion Service>/
nohup.out
```

You can even check the detailed logs in the service log, that is available in the XRegion Service home directory specified in the XRegion Service configuration file (`json.config`) earlier.

xrstop

In a multi-region setup, you can stop any running Xregion service using `xrstop` command. For example, you must stop an Xregion Service if you want to relocate the service to another host machine. Then you must shut it down in the current machine and restart it in the new host machine.

```
java -Xmx1024m -Xms256m -jar $KVHOME/lib/kvstore.jar xrstop \
-config <complete path to the json.config file>
```

The parameter config specifies the complete path where the `json.config` file is placed.

Initial Capacity Planning

To deploy a store, you must specify a replication factor, the desired number of partitions, and the Storage Nodes on which to deploy the store. The following sections

describe how to calculate these values based on your application's requirements and the characteristics of the hardware available to host the store.

The resource estimation is a two step process:

1. Determine the storage and I/O throughput capacity of a representative shard, given the characteristics of the application, the disk configuration on each machine, and the disk throughput. As part of this step, you should also estimate the amount of physical memory that each machine requires, and its network throughput capacity.
2. Use the shard level storage and I/O throughput capacities as a basis for extrapolating throughput from one shard to the required number of shards and machines, given the storewide application requirements.

Oracle NoSQL Database distribution includes a spreadsheet for you to use in the capacity planning process. The spreadsheet is located here: `<KVHOME>/doc/misc/InitialCapacityPlanning.xls`.

The spreadsheet has two main sections:

- 1. Shard Capacity
- 2. Store Sizing

The two main sections both have some required parameters for you to complete, as well as parameters with default options.

The next sections in this appendix correspond to named columns in the spreadsheet:

- Column A lists cell names associated with the values in column B.
- Dark purple, bold text labels represent required values for you to provide as input.
- Dark blue, bold text labels indicate default values that you can optionally change. The supplied default values are adequate for most estimates.
- Column C has descriptions of the value or computation associated with the value in column B.
- The first three sections cover Shard Capacity: Application Characteristics, Hardware Characteristics Machine Physical Memory contain required inputs.

The spreadsheet computes all other cells using the following formulas.

- After filling in the required inputs, the *StoreMachines* cell indicates how many Storage Nodes should be available in the Storage Node pool.
- The *StorePartitions* cell indicates how many partitions to specify when creating the store.

The spreadsheet calculations also account for JVM overhead. Keep in mind that these computations yield estimates. The underlying model used as a basis for the estimation makes certain simple assumptions. These assumptions are necessary because it is difficult to provide a simple single underlying model that works well under a wide range of application requirements. Use these estimates only as an initial starting point, and refine them as necessary under a simulated or actual load.

Shard Capacity

To determine the shard capacity, first determine the application and hardware characteristics described in this section. Having determined these characteristics, enter them into the accompanying spreadsheet. The spread sheet will then calculate the capacity of a shard on the basis of the supplied application and hardware characteristics.

Application Characteristics

Replication Factor

In general, a *Primary Replication Factor* of 3 is adequate for most applications and is a good starting point, because 3 replicas allow write availability if a single primary zone fails. It can be refined if performance testing suggests some other number works better for the specific workload. Do not select a *Primary Replication Factor* of 2 because doing so means that even a single failure results in too few sites to elect a new master. This is not the case if you have an Arbiter Node, as a new master can still be elected if the Replication Factor is two and you lose a Replication Node. However, if you have multiple failures before both Replication Nodes are caught up, you may not be able to elect a new master. A *Primary Replication Factor* of 1 is to be avoided in general since Oracle NoSQL Database has just a single copy of the data; if the storage device hosting the data were to fail the data could be lost.

Larger *Primary Replication Factor* provide two benefits:

1. Increased durability to better withstand disk or machine failures.
2. Increased read request throughput, because there are more nodes per shard available to service those requests.

However, the increased durability and read throughput has costs associated with it: more hardware resources to host and serve the additional copies of the data and slower write performance, because each shard has more nodes to which updates must be replicated.

> **Note:**
>
> Only the Primary Replication Factor affects write availability, but both Primary and Secondary Replication Factors, and therefore the Store Replication Factor, have an effect on read availability.

The *Primary Replication Factor* is defined by the cell *RF*.

Average Key Size

Use knowledge of the application's key schema and the relative distributions of the various keys to arrive at an average key length. The length of a key on disk is the number of UTF-8 bytes needed to represent the components of the key, plus the number of components, minus one.

This value is defined by the cell *AvgKeySize*.

Average Value Size

Use knowledge of the application to arrive at an average serialized value size. The value size will vary depending upon the particular serialization format used by the application.

This value is defined by the cell *AvgValueSize*.

Read and Write Operation Percentages

Compute a rough estimate of the relative frequency of store level read and write operations on the basis of the KVS API operations used by the application.

At the most basic level, each KVS get() call results in a store level read operation and each put() operation results in a store level write operation. Each KVS multi key operation (KVStore.execute(), multiGet(), or multiDelete()) can result in multiple store level read/write operations. Again, use application knowledge about the number of keys accessed in these operations to arrive at an estimate.

Express the estimate as a read percentage, that is, the percentage of the total operations on the store that are reads. The rest of the operations are assumed to be write operations.

This value is defined by the cell *ReadOpsPercent*.

Estimate the percentage of read operations that will likely be satisfied from the file system cache. The percentage depends primarily upon the application's data access pattern and the size of the file system cache. Sizing Advice contains a discussion of how this cache is used.

This value is defined by the cell *ReadCacheHitPercent*.

Hardware Characteristics

Determine the following hardware characteristics based on a rough idea of the type of the machines that will be used to host the store:

- The number of disks per machine that will be used for storing KV pairs. This value is defined by the cell *DisksPerMachine*. The number of disks per machine typically determines the Storage Node Capacity as described in Storage Node Parameters.
- The usable storage capacity of each disk. This value is defined by the cell *DiskCapacityGB*.
- The IOPs capacity of each disk. This information is typically available in the disk spec sheet as the number of sustained random IO operations/sec that can be delivered by the disk. This value is defined by the cell *DiskIopsPerSec*.

The following discussion assumes that the system will be configured with one RN per disk.

Shard Storage and Throughput Capacities

There are two types of capacity that are relevant to this discussion: 1) Storage Capacity 2) Throughput Capacity. The following sections describe how these two measures of capacity are calculated. The underlying calculations are done automatically by the attached spreadsheet based upon the application and hardware characteristics supplied earlier.

Shard Storage Capacity

The storage capacity is the maximum number of KV pairs that can be stored in a shard. It is calculated by dividing the storage actually available for live KV pairs (after accounting for the storage set aside as a safety margin and cleaner utilization) by the storage (including a rough estimation of Btree overheads) required by each KV pair.

The KV Storage Capacity is computed by the cell: *MaxKVPairsPerShard*.

Shard I/O Throughput capacity

The throughput capacity is a measure of the read and write ops that can be supported by a single shard. In the calculations below, the logical throughput capacity is derived from the disk IOPs capacity based upon the percentage of logical operations that actually translate into disk IOPs after allowing for cache hits. The Machine Physical Memory section contains more detail about configuring the caches used by Oracle NoSQL Database.

For logical read operations, the shard-wide IOPs is computed as:

```
(ReadOpsPercent * (1 - ReadCacheHitPercent))
```

Note that all percentages are expressed as fractions.

For logical write operations, the shard-wide IOPs is computed as:

```
(((1 - ReadOpsPercent) / WriteOpsBatchSize) * RF)
```

The writeops calculations are very approximate. Write operations make a much smaller contribution to the IOPs load than do the read ops due to the sequential writes used by the log structured storage system. The use of WriteOpsBatchSize is intended to account for the sequential nature of the writes to the underlying JE log structured storage system. The above formula does not work well when there are no reads in the workload, that is, under pure insert or pure update loads. Under pure insert, the writes are limited primarily by acknowledgement latency which is not modeled by the formula. Under pure update loads, both the acknowledgement latency and cleaner performance play an important role.

The sum of the above two numbers represents the percentage of logical operations that actually result in disk operations (the **DiskIopsPercent** cell). The shard's logical throughput can then be computed as:

```
(DiskIopsPerSec * RF)/DiskIopsPercent
```

and is calculated by the cell **OpsPerShardPerSec**.

Memory and Network Configuration

Having established the storage and throughput capacities of a shard, the amount of physical memory and network capacity required by each machine can be determined. Correct configuration of physical memory and network resources is essential for the proper operation of the store. If your primary goal is to determine the total size of the store, skip ahead to Estimate total Shards and Machines but make sure to return to this section later when it is time to finalize the machine level hardware requirements.

> **Note:**
>
> You can also set the memory size available for each Storage Node in your store, either through the `memory_mb` parameter of the `makebootconfig` utility or through the `memorymb` Storage Node parameter. For more information, see Installation Configuration Parameters and Storage Node Parameters respectively.

Machine Physical Memory

The shard storage capacity (computed by the cell *MaxKVPairsPerShard*) and the average key size (defined by the cell *AvgKeySize* cell) can be used to estimate the physical memory requirements of the machine. The physical memory on the machine backs up the caches used by Oracle NoSQL Database.

Sizing the in-memory cache correctly is essential for meeting store's performance goals. Disk I/O is an expensive operation from a performance point of view; the more operations that can be serviced from the cache, the better the store's performance.

Before continuing, it is worth noting that there are two caches that are relevant to this discussion:

1. The JE cache. The underlying storage engine used by Oracle NoSQL Database is Berkeley DB Java Edition (JE). JE provides an in-memory cache. For the most part, this is the cache size that is most important, because it is the one that is simplest to control and configure.
2. The file system (FS) cache. Modern operating systems attempt to improve their I/O subsystem performance by providing a cache, or buffer, that is dedicated to disk I/O. By using the FS cache, read operations can be performed very quickly if the reads can be satisfied by data that is stored there.

Sizing Advice

JE uses a Btree to organize the data that it stores. Btrees provide a tree-like data organization structure that allows for rapid information lookup. These structures consist of interior nodes (INs) and leaf nodes (LNs). INs are used to navigate to data. LNs are where the data is actually stored in the Btree.

Because of the very large data sets that an Oracle NoSQL Database application is expected to use, it is unlikely that you can place even a small fraction of the data into JE's in-memory cache. Therefore, the best strategy is to size the cache such that it is large enough to hold most, if not all, of the database's INs, and leave the rest of the node's memory available for system overhead (negligible) and the FS cache.

Both INs and LNs can take advantage of the FS cache. Because INs and LNs do not have Java object overhead when present in the FS cache (as they would when using the JE cache), they can make more effective use of the FS cache memory than the JE cache memory.

Of course, in order for the FS cache to be truly effective, the data access patterns should not be completely random. Some subset of your key-value pairs must be favored over others in order to achieve a useful cache hit rate. For applications where the access patterns are not random, the high file system cache hit rates on LNs and INs can increase throughput and decrease average read latency. Also, larger file system caches, when properly tuned, can

help reduce the number of stalls during sequential writes to the log files, thus decreasing write latency. Large caches also permit more of the writes to be done asynchronously, thus improving throughput.

Determine JE Cache Size

To determine an appropriate JE cache size, use the `com.sleepycat.je.util.DbCacheSize` utility. This utility requires as input the number of records and the size of the application keys. You can also optionally provide other information, such as the expected data size. The utility then provides a short table of information. The number you want is provided in the `Cache Size` column, and in the `Internal nodes and leaf nodes: MAIN cache` row.

For example, to determine the JE cache size for an environment consisting of 100 million records, with an average key size of 12 bytes, and an average value size of 1000 bytes, invoke `DbCacheSize` as follows:

```
java -Xmx64m -Xms64m \
-d64 -XX:+UseCompressedOops -jar je.jar DbCacheSize \
-key 12 -data 1000 -records 100000000 -replicated

      === Environment Cache Overhead ===

      2,536,302 minimum bytes

      To account for JE daemon operation, record locks, HA network
          connections, etc, a larger amount is needed in practice.

      === Database Cache Size ===

       Number of Bytes  Description
       ---------------  -----------
         3,896,520,528  Internal nodes only:
         4,660,565,808  Internal nodes and records version
       110,107,803,216  Internal nodes and leaf nodes
```

Please make note of the following jvm arguments (they have a special meaning when supplied to DbCacheSize):

1. The above example command assumes using Java 11 or later. It is recommended to use Java 17 version. Only 64-bit JVMs are supported by NoSQL DB.
2. The -XX:+UseCompressedOops causes cache sizes to account for CompressedOops mode, which is used by NoSQL DB by default. This mode uses more efficient 32 bit pointers in a 64-bit JVM thus permitting better utilization of the JE cache.
3. The -replicated is used to account for memory usage in a JE ReplicatedEnvironment, which is always used by NoSQL DB.

These arguments when supplied to `Database Cache Size` serve as an indication that the JE application will also be supplied these arguments and `Database Cache Size` adjusts its calculations appropriately. The arguments are used by Oracle NoSQL Database when starting up the Replication Nodes which uses these caches.

The output indicates that a cache size of 3.6 GB is sufficient to hold all the internal nodes representing the Btree in the JE cache. With a JE cache of this size, the IN nodes will be fetched from the JE cache and the LNs will be fetched from the off-heap cache or the disk.

For more information on using the `DbCacheSize` utility, see this Javadoc page. Note that in order to use this utility, you must add the `<KVHOME>/lib/je.jar` file to your Java classpath. <KVHOME> represents the directory where you placed the Oracle NoSQL Database package files.

Having used `DbCacheSize` to obtain the JE cache size, the heap size can be calculated from it. To do this, enter the number obtained from `DbCacheSize` into the cell named *DbCacheSizeMB* making sure to convert the units from bytes to MB. The heap size is computed by the cell *RNHeapMB* as below:

```
(DBCacheSizeMB/RNCachePercent)
```

where *RNCachePercent* is the percentage of the heap that is used for the JE cache. The computed heap size should not exceed 32GB, so that the java VM can use its efficient CompressedOops format to represent the java objects in memory. Heap sizes with values exceeding 32GB will appear with a strikethrough in the *RNHeapMB* cell to emphasize this requirement. If the heap size exceeds 32GB, try to reduce the size of the keys to reduce the JE cache size in turn and bring the overall heap size below 32GB.

The heap size is used as the basis for computing the memory required by the machine as below:

```
(RNHeapMB * DisksPerMachine)/SNRNHeapPercent
```

where *SNRNHeapPercent* is the percentage of the physical memory that is available for use by the RN's hosted on the machine. The result is available in the cell *MachinePhysicalMemoryMB*.

Machine Network Throughput

We need to ensure that the NIC attached to the machine is capable of delivering the application I/O throughput as calculated earlier in Shard I/O Throughput capacity, because otherwise it could prove to be a bottleneck.

The number of bytes received by the machine over the network as a result of write operations initiated by the client is calculated as:

```
(OpsPerShardPerSec * (1 - ReadOpsPercent) *
 (AvgKeySize + AvgValueSize)) * DisksPerMachine
```

and is denoted by *ReceiveBytesPerSec* in the spreadsheet. Note that whether a node is a master or a replica does not matter for the purposes of this calculation; the inbound write bytes come from the client for the master and from the masters for the replicas on the machine.

The number of bytes received by the machine as a result of read requests is computed as:

```
((OpsPerShardPerSec * ReadOpsPercent)/RF) *
 (AvgKeySize + ReadRequestOverheadBytes) * DisksPerMachine
```

where *ReadRequestOverheadBytes* is a fixed constant overhead of 100 bytes.

The bytes sent out by the machine over the network as a result of the read operations has two underlying components:

1. The bytes sent out in direct response to application read requests and can be expressed as:

```
((OpsPerShardPerSec *  ReadOpsPercent)/RF) *
 (AvgKeySize + AvgValueSize) * DisksPerMachine
```

2. The bytes sent out as replication traffic by the masters on the machine expressed as:

```
(OpsPerShardPerSec * (1 - ReadOpsPercent) *
 (AvgKeySize + AvgValueSize) * (RF-1)) * MastersOnMachine
```

The sum of the above two values represents the total outbound traffic denoted by SendBytesPerSec in the spreadsheet.

The total inbound and outbound traffic must be comfortably within the NIC's capacity. The spreadsheet calculates the kind of network card, GigE or 10GigE, which is required to support the traffic.

Estimate total Shards and Machines

Having calculated the per shard capacity in terms of storage and throughput, the total number of shards and partitions can be estimated on the basis of the maximum storage and throughput required by the store as a whole using a simple extrapolation. The following inputs must be supplied for this calculation:

1. The maximum number of KV pairs that will stored in the initial store. This value is defined by the cell *MaxKVPairs*. This initial maximum value can be increased subsequently by using the topology transformation commands described in Transforming the Topology Candidate.
2. The maximum read/write mixed operation throughput expressed as operations/sec for the entire store. The percentage of read operations in this mix must be the same as that supplied earlier in the *ReadOpsPercent* cell. This value is defined by the cell *MaxStorewideOpsPerSec*.

The required number of shards is first computed on the basis of storage requirements as below:

```
MaxKVPairs/MaxKVPairsPerShard
```

This value is calculated by the cell *StorageBasedShards*.

The required number of shards is then computed again based upon IO throughput requirements as below:

```
MaxStorewideOpsPerSec/OpsPerShardPerSec
```

This value is calculated by the cell named *OpsBasedShards*.

The maximum of the shards computed on the basis of storage and throughput above is sufficient to satisfy both the total storage and throughput requirements of the application.

The value is calculated by the cell *StoreShards*. To highlight the basis on which the choice was made, the smaller of the two values in *StorageBasedShards* or *OpsBasedShards* has its value crossed out.

Having determined the number of required shards, the number of required machines is calculated as:

```
MAX(RF, (StoreShards*RF)/DisksPerMachine)
```

Number of Partitions

Every shard in the store must contain at least one partition, but it is best to configure the store so that each shard always contains more than one partition. The records in the KVStore are spread evenly across the KVStore partitions, and as a consequence they are also spread evenly across shards. The total number of partitions that the store should contain is determined when the store is initially created. This number is static and cannot be changed over the store's lifetime, so it is an important initial configuration parameter.

The number of partitions must be more than the largest number of shards the store will contain. It is possible to add shards to the store, and when you do, the store is re-balanced by moving partitions between shards (and with them, the data that they contain). Therefore, the total number of partitions is actually a permanent limit on the total number of shards your store is able to contain.

Note that there is some overhead in configuring an excessively large number of partitions. That said, it does no harm to select a partition value that provides plenty of room for growing the store. It is not unreasonable to select a partition number that is 10 times the maximum number of shards.

The number of partitions is calculated by the cell *StorePartitions*.

```
StoreShards * 10
```

Tuning

The default tuning parameters available for the Oracle NoSQL Database software should in general be acceptable for production systems, and so do not require any tuning. However, the underlying operating system will have default values for various kernel parameters which require modification in order to achieve the best possible performance for your store's installation.

This appendix identifies the kernel parameters and other system tuning that you should manage when installing a *production* store. By this, we mean any store whose performance is considered critical. Evaluation systems installed into a lab environment probably do not need this level of tuning unless you are using those systems to measure the store's performance.

> **Note:**
>
> Oracle NoSQL Database is most frequently installed on Linux systems, and so that is what this appendix focuses on.

Turn off the swap

For best performance on a dedicated Oracle NoSQL Database server machine, turn off the swap on the machine. Oracle NoSQL Database processes are careful in their management of the memory they use to ensure that they do not exceed the RAM available on the machine.

The performance gains come from two sources:

1. The I/O overhead due to swap is eliminated. This is especially important if the disk normally used for swap also holds the store's log files used to persist data.
2. Reduces the CPU overhead associated with kswapd.

To turn off the swap, do not mount any swap partitions at boot time. You do this by eliminating all swap related mount entries from `/etc/fstab`. These are all the rows with the entry "swap" in their mount point (column 2) and file system type (column 3) entries.

You can verify that no swap space is being used by running the `free` command. Do this after the `/etc/fstab` has been modified and the machine has been rebooted:

```
-bash-4.1$ free -m
             total       used       free     shared    buffers     cached
Mem:         72695      72493        202          0        289       2390
-/+ buffers/cache:      69813       2882
Swap:            0          0          0
```

The `Swap/total` cell in the above table should read `0`.

Linux Page Cache Tuning

Tune your page cache to permit the OS to write asynchronously to disk whenever possible. This allows background writes, which minimize the latency resulting from serial write operations such as fsync. This also helps with write stalls which occur when the file system cache is full and needs to be flushed to disk to make room for new writes. We have observed significant speedups (15-20%) on insert-intensive benchmarks when these parameters are tuned as described below.

Place the following commands in `/etc/sysctl.conf`. Run

```
sysctl -p
```

to load the new settings so they can take effect without needing to reboot the machine.

```
# Set vm.dirty_background_bytes to 10MB to ensure that
# on a 40MB/sec hard disk a fsync never takes more than 250ms and takes
```

```
# just 125ms on average. The value of vm.dirty_background_bytes
# should be increased on faster SSDs or I/O subsytems with higher
# throughput. You should increase this setting by the same proportion
# as the relative increase in throughput. For example, for a typical SSD
# with a throughput of 160MB/sec, vm.dirty_background_bytes should be set
# to 40MB so fsync takes ~250ms. In this case, the value was increased by
# a factor of 4.
vm.dirty_background_bytes=10485760

# IO calls effectively become synchronous(waiting for the underlying
# device to complete them). This setting helps minimize the
# possibility of a write request stalling in JE while holding the
# write log latch.
vm.dirty_ratio=40

# Ensures that data does not hang around in memory longer than
# necessary. Given JE's append-only style of writing, there is
# typically little benefit from having an intermediate dirty page
# hanging around, because it is never going to be modified. By
# evicting the dirty page earlier, its associated memory is readily
# available for reading or writing new pages, should that become
# necessary.
vm.dirty_expire_centisecs=1000
```

Earlier versions of the Linux kernel may not support `vm.dirty_background_bytes`. On these older kernels you can use `vm.dirty_background_ratio` instead. Pick the ratio that gets you closest to 10MB. On some systems with a lot of memory this may not be possible due to the large granularity associated with this configuration knob. A further impediment is that a ratio of 5 is the effective minimum in some kernels.

```
vm.dirty_background_ratio=5
```

Use `sysctl -a` to verify that the parameters described here are set as expected.

OS User Limits

When running a large Oracle NoSQL Database store, the default OS limits may be insufficient. The following sections list limits that are worth reviewing.

File Descriptor Limits

Use `ulimit -n` to determine the maximum number of files that can be opened by a user. The number of open file descriptors may need to be increased if the defaults are too low. It's worth keeping in mind that each open network connection also consumes a file descriptor. Machines running clients as well as machines running RNs may need to increase this limit for large stores with 100s of nodes.

Add entries like the ones below in `/etc/security/limits.conf` to change the file descriptor limits:

```
$username soft nofile 10240
$username hard nofile 10240
```

where `$username` is the username under which the Oracle NoSQL Database software runs.

Note that machines hosting multiple replication nodes; that is, machines configured with a capacity > 1; will need larger limits than what is identified here.

Process and Thread Limits

Use `ulimit -u` to determine the maximum number of processes (threads are counted as processes under Linux) that the user is allowed to create. Machines running clients as well as machines running RNs may need to increase this limit to accommodate large numbers of concurrent requests.

Add entries like the ones below in `/etc/security/limits.conf` to change the thread limits:

```
$username soft nproc 8192
$username hard nproc 8192
```

where `$username` is the username under which the Oracle NoSQL Database software runs.

Note that machines hosting multiple replication nodes; that is, machines configured with a capacity > 1; will need larger limits than what is identified here.

Linux Network Configuration Settings

Before continuing, it is worth checking that the network interface card is configured as expected during the initial setup of each SN, because it is harder to debug these problems later when such configuration problems show up under load.

Use the following command to determine which network interface is being used to access a particular subnet on each host. This command is particularly useful for machines with multiple NICs:

```
$ ip addr ls to 192.168/16
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    state UP qlen 1000
    inet 192.168.1.19/24 brd 192.168.1.255 scope global eth0
```

Use the following command to get information about the configuration of the NIC:

```
$ ethtool -i eth2
driver: enic
version: 2.1.1.13
firmware-version: 2.0(2g)
bus-info: 0000:0b:00.0
```

Use the following command to get information about the NIC hardware:

```
$ lspci -v | grep "Ethernet controller"
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit
Ethernet Controller (rev 02)
```

Use the following command to get information about the network speed. Note that this command requires sudo:

```
$ sudo ethtool eth0 | grep Speed
    Speed: 1000Mb/s
```

You may want to consider using 10 gigabit Ethernet, or other fast network implementations, to improve performance for large clusters.

Server Socket Backlog

The typical default maximum server socket backlog, typically set at 128, is too small for server style loads. It should be at least 1K for server applications and even a 10K value is not unreasonable for large stores.

Set the `net.core.somaxconn` property in `sysctl.conf` to modify this value.

Isolating HA Network Traffic

If the machine has multiple network interfaces, you can configure Oracle NoSQL Database to isolate HA replication traffic on one interface, while client request traffic uses another interface. Use the `-hahost` parameter of the `makebootconfig` command to specify the interface to be used by HA as in the example below:

```
java -Xmx64m -Xms64m \
-jar kvstore.jar makebootconfig -root /disk1/kvroot \
-host sn10.example.com -port 5000 -harange 5010,5020 \
-admindir /disk2/admin -admindirsize 2 GB
-storagedir /disk2/kv -hahost sn10-ha.example.com
```

In this example, all client requests will use the interface associated with `sn10.example.com`, while HA traffic will use the interface associated with `sn10-ha.example.com`.

Receive Packet Steering

When multiple RNs are located on a machine with a single queue network device, enabling Receive Packet Steering (RPS) can help performance by distributing the CPU load associated with packet processing (soft interrupt handling) across multiple cores. Multi-queue NICs provide such support directly and do not need to have RPS enabled.

Note that this tuning advice is particularly appropriate for customers using Oracle Big Data Appliance.

You can determine whether a NIC is multi-queue by using the following command:

```
sudo ethtool -S eth0
```

A multi-queue NIC will have entries like this:

```
 rx_queue_0_packets: 271623830
     rx_queue_0_bytes: 186279293607
     rx_queue_0_drops: 0
     rx_queue_0_csum_err: 0
```

```
     rx_queue_0_alloc_failed: 0
     rx_queue_1_packets: 273350226
     rx_queue_1_bytes: 188068352235
     rx_queue_1_drops: 0
     rx_queue_1_csum_err: 0
     rx_queue_1_alloc_failed: 0
     rx_queue_2_packets: 411500226
     rx_queue_2_bytes: 206830029846
     rx_queue_2_drops: 0
     rx_queue_2_csum_err: 0
     rx_queue_2_alloc_failed: 0
...
```

For a 32 core Big Data Appliance using Infiniband, use the following configuration to distribute receive packet processing across all 32 cores:

```
echo ffffffff > /sys/class/net/eth0/queues/rx-0/rps_cpus
```

where `ffffffff` is a bit mask selecting all 32 cores.

For more information on RPS please consult:

1. About the Unbreakable Enterprise Kernel
2. Receive packet steering

MTU Size

When using machines connected to networks running at 1000Mb/s or higher speeds, it is recommended that you enable jumbo frames on the machines that are hosting the RNs. HA replication benefits from the use of Jumbo frames such that the feeder (via the HA parameter: feederBatchBuffKb) uses a default batch buffer size of 8K, which is well matched to use a Jumbo frame.

Setting the MTU to 9000 is also helps in improving network performance on KV client machines with high speed networks, especially if the request or response payloads frequently exceed the default MTU size of 1500.

To enable jumbo frames, set the MTU to 9000 on each machine. Also, verify that this MTU is supported on the entire network path between machines hosting the RNs.

For example, to determine the speed of the *ens3* interface, use the following command:

```
# ip link show ens3
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc mq state UP
mode DEFAULT group default qlen 1000
    link/ether 00:00:17:01:2c:b6 brd ff:ff:ff:ff:ff:ff
```

If required, change the MTU configuration using the *ip* command:

```
# ip link set ens3 mtu 9000
```

Check AES Intrinsics Settings

While most modern hardware systems enable AES Intrinsics by default, you can check these settings yourself to confirm their use.

An Oracle NoSQL Database installation using the SSL/TLS encryption gets better performance if it can take advantage of hardware acceleration available on the host machine.

Most SSL cipher suites use the AES encryption algorithm, and most modern processors support hardware acceleration for AES. To confirm that a Java installation is taking advantage of AES hardware acceleration, check to see if AES intrinsics are enabled. You can get that information by printing flag values for the Java virtual machine from your terminal using the `-XXPrintFlagsFinal` flag, as follows. Then, search for the two boolean flags `UseAES`, and `UseAESIntrinsics`. In this example, results show that AES intrinsics are enabled.

```
java -XX:+PrintFlagsFinal -version | grep 'AES\|Intrinsics'
bool UseAES = true {product} {default}
bool UseSSE42Intrinsics = true {ARCH product} {default}
java version "10.0.2" 2018-07-17
Java(TM) SE Runtime Environment 18.3 (build 10.0.2+13)
Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.2+13, mixed mode)
```

Setting AES Intrinsics

For best performance, enable AES intrinsics on all machines that support them. If not enabled when you run the check just described, you must specify `-XX:+UseAES` and `-XX:+UseAESIntrinsics` for every JVM command line that uses SSL, using these flags:

```
java -XX:+UseAES -XX:+UseAESIntrinsics [...]
```

You can add these two flags to the JVM options for RNs by setting the `configProperties` parameter. See Replication Node Parameters .

Client applications that make calls to the NoSQL API, should specify these system properties on the Java command line for the application.

Viewing Key Distribution Statistics

As you might already know, Oracle NoSQL Database stores the data by distributing the rows across all the partitions by hashing each row's shard key. Based on the activity in your store's tables, Oracle NoSQL Database collects the key distribution data into internally managed system tables. As needed, you can access these statistics by querying these system tables.

As an Oracle NoSQL Database administrator, you may encounter many situations where you need to view the key distribution statistics. To discuss one such use-case, consider a situation where you are not able achieve the expected amount of throughput for your Oracle NoSQL Database in spite of having multiple shards in your cluster. This might happen if the data in your store is not distributed across the shards evenly. In order to confirm if this is the reason behind low throughput, you need a mechanism to understand how the data is distributed across the Oracle NoSQL Database cluster. The Key Distribution Statistics provided by the Oracle NoSQL Database can help you understand the data distribution across multiple partitions and shards in your store.

The two system tables into which the Oracle NoSQL Database collects the key distribution statistics are:

- `SYS$TableStatsPartition`
- `SYS$TableStatsIndex`

Oracle NoSQL Database manages and maintains these system tables internally. When you enable security on your store, these system tables are read-only. Regardless of security, the schema for system tables is immutable. The name of system tables is defined with the prefix `SYS$`. You are not allowed to create any other table name using this reserved prefix.

`SYS$TableStatsPartition`

This table stores the table key statistics at the partition level. It contains a row for each partition for every table. For example, if you created a store with 100 partitions, this table contains 100 rows for every table in your store. The statistics stored per partition for each table in your store are:

1. The number of rows stored
2. The average size of keys in bytes
3. The size in bytes consumed by the rows

The structure of the `SYS$TableStatsPartition` table is as below:

Column	Data Type	Description
`tableName`	string	Name of the table whose Key Distribution Statistics are being stored.
`partitionId`	integer	Partition ID
`shardId`	integer	Shard ID
`count`	long	Number of rows stored.
`avgKeySize`	integer	The average size of keys in bytes.
`tableSize`	long	The size in bytes consumed by the rows.
`tableSizeWithTombstones`	long	The table storage in bytes including tombstones.

> **Note:**
>
> The tombstone is a small piece of storage when a row in a multi-region table is deleted. This per-row storage overhead is used to keep some metadata of the deleted row, which will be used in conflict resolution when another row is replicated from remote regions. The tombstone will expire in 7 days after it is created and the storage will be released. For tables without tombstones (For example, non-multi-region tables, system tables, etc,) the metric `tableSizeWithTombstones` would be the same as the metric `tableSize` in the system table. The difference between the two metrics is the total storage size of tombstones in the table.

`SYS$TableStatsIndex`

This table stores the index key statistics at the shard level. This table contains a row for each shard for every index. You do not have direct control over the number of shards created in your store, but you can always view the store topology to know how many shards are created in your store. For more information, see show topology.

The statistics stored per shard for each table in your store are:

1. The number of index rows
2. The average size of the index keys in bytes
3. The size in bytes consumed by the index rows

The structure of `SYS$TableStatsIndex` system table is as below:

Column	Data Type	Description
`tableName`	string	Name of the table whose Key Distribution Statistics are being stored.
`indexName`	string	Name of the index
`shardId`	integer	Shard ID
`count`	long	Number of index rows stored.
`avgKeySize`	integer	Average size of index keys in bytes.
`indexSize`	long	The size in bytes consumed by the index rows.

Gathering the Key Distribution Statistics

The gathering of the key distribution statistics into the system tables is determined by two parameters:

- `rnStatisticsEnabled`:
  In Oracle NoSQL Database, the Key Distribution Statistics are enabled by default for all newly created stores. You can disable the capturing of these statistics by executing the following command from Admin Command Line Interface (CLI):

  ```
  plan change-parameters -wait -all-rns -params "rnStatisticsEnabled=false"
  ```

- `rnStatisticsGatherInterval`:
  In Oracle NoSQL Database, the default time interval between two consecutive updates on `SYS$TableStatsPartition` and `SYS$TableStatsIndex` is 24 hours. You can change the time interval between the capture of these statistics by modifying the `rnStatisticsGatherInterval` parameter. The time unit specified must be in days, hours, or minutes.

  For example, to instruct Oracle NoSQL Database to collect the Key Distribution Statistics after every minute, execute the following command from Admin Command Line Interface (CLI):

  ```
  plan change-parameters -wait -all-rns -params
  "rnStatisticsGatherInterval=1 min"
  ```

> **Note:**
>
> Enabling the Key Distribution Statistics does not immediately trigger the collection of statistics. Oracle NoSQL Database initiates the statistics collection at a time based on the collection interval defined by the `rnStatisticsGatherInterval` parameter.

`rnStatisticsGatherInterval`

Reading the Key Distribution Statistics
You can query the system tables to get key distribution data or review the gathering process.

In order to get a complete set of statistics for a given table, you must aggregate the per-partition values stored for that table in the `SYS$TableStatsPartition` system table.

For example, to get the total number of rows in a table named `myTable`, you must sum the values in the count column for all the rows in the `SYS$TableStatsPartition` table where tableName = myTable.

Example Query:

```
sql-> select * from SYS$TableStatsPartition where tableName =
'myTable';
```

Result:

```
{"tableName":"myTable","partitionId":8,"shardId":3,"count":0,"avgKeySiz
e":0,"tableSize":0}
{"tableName":"myTable","partitionId":9,"shardId":4,"count":0,"avgKeySiz
e":0,"tableSize":0}
{"tableName":"myTable","partitionId":1,"shardId":1,"count":0,"avgKeySiz
e":0,"tableSize":0}
{"tableName":"myTable","partitionId":4,"shardId":2,"count":0,"avgKeySiz
e":0,"tableSize":0}
{"tableName":"myTable","partitionId":7,"shardId":3,"count":50,"avgKeySi
ze":15,"tableSize":103}
{"tableName":"myTable","partitionId":10,"shardId":4,"count":50,"avgKeyS
ize":15,"tableSize":103}
{"tableName":"myTable","partitionId":5,"shardId":2,"count":0,"avgKeySiz
e":0,"tableSize":0}
{"tableName":"myTable","partitionId":6,"shardId":2,"count":0,"avgKeySiz
e":0,"tableSize":0}
{"tableName":"myTable","partitionId":2,"shardId":1,"count":0,"avgKeySiz
e":0,"tableSize":0}
{"tableName":"myTable","partitionId":3,"shardId":1,"count":0,"avgKeySiz
e":0,"tableSize":0}
```

In the above result, observe that there are 50 keys each in `"partitionId":7,"shardId":3` and `"partitionId":10,"shardId":4` whereas all the other partitions and shards are empty. This shows that the key data is not distributed evenly across all the partitions and shards.

Similarly, you can query the `SYS$TableStatsIndex` system table to read the index key distribution statistics for a given table at the shard level.

For example, to get the total number of index rows in a table named `myTable`, you must sum the values in the count column for all the index rows in the `SYS$TableStatsIndex` table where tableName = myTable.

Example Query:

```
sql-> select * from SYS$TableStatsIndex where tableName = 'myTable';
```

Result:

```
{"tableName":"myTable","indexName":"idx_shard_key","shardId":3,"count":50,"av
gKeySize":1,"indexSize":75}
{"tableName":"myTable","indexName":"idx_shard_key","shardId":4,"count":50,"av
gKeySize":1,"indexSize":75}
{"tableName":"myTable","indexName":"idx_shard_key","shardId":1,"count":0,"avg
KeySize":0,"indexSize":0}
{"tableName":"myTable","indexName":"idx_shard_key","shardId":2,"count":0,"avg
KeySize":0,"indexSize":0}
```

As you can see from the above result, there are 50 index keys each in `"shardId":3` and `"shardId":4` whereas all the other shards are empty. This shows that the index key data is not distributed evenly across all the shards.

Retention of the Key Distribution Statistics
After collecting the key distribution statistics, they are retained in the system tables for a fixed time period. This value is determined by the `rnStatisticsTTL` parameter. By default, these statistics are retained for 60 days. However, you can change this value by executing the change-parameters plan from the Admin CLI. The time unit specified must be in days or hours.

For example, execute the following command from Admin Command Line Interface (CLI) to retain the Key Data Statistics in the system tables for 90 days:

```
plan change-parameters -wait -all-rns -params "rnStatisticsTTL=90 days"
```

Few points to note are:

- Any changes that you make to the `rnStatisticsTTL` parameter will not be applied to the existing rows in the `SYS$TableStatsPartition` and `SYS$TableStatsIndex` tables. They will take effect only after the next gathering scan.
- If you disable the collection of Key Distribution Statistics, all the rows present in the system tables will expire after the current Time to Live (TTL) period.
- If you drop any tables or indexes in your store, their statistics rows present in the system tables will also expire after the TTL period.
- Even if you change the `rnStatisticsTTL` to a value less than `rnStatisticsGatherInterval`, all the existing statistics rows will only expire as the TTL value defined during the last scan.
- `rnStatisticsTTL` can be set to 0 days. However, this is not recommended as it disables automatic removal of the statistics rows.

Examples: Key Distribution Statistics

Key distribution statistics can also be used to provide estimates of other information about tables that may prove useful.

Example 7-4 Key Distribution Statistics

To estimate the number of elements in each table, perform the following query:

```
SELECT tableName,
    sum(count) AS count
FROM SYS$TableStatsPartition
WHERE NOT contains (tableName, "$")
GROUP BY tableName
```

The clause `WHERE NOT CONTAINS (tableName, "$")` filters out system tables by only including tables whose names do not contain the "$" character.

The clause `GROUP BY tableName` is what causes the sums to be computed over all of the partition entries for the same table.

Example 7-5 Key Distribution Statistics

To estimate the average key size for each table, perform the following query:

```
SELECT tableName,
    CASE WHEN sum(count) = 0
        THEN 0
        ELSE sum(avgKeySize*count)/sum(count)
    END AS avgKeySize
FROM SYS$TableStatsPartition
WHERE NOT contains(tableName, "$")
GROUP BY tableName
```

The case clause skips entries whose count is zero, and otherwise weights each entry by the element count, dividing the result by the total count.

Example 7-6 Key Distribution Statistics

To estimate the number of elements in each index, perform the following query:

```
SELECT tableName,
       indexName,
       sum(count) AS count
FROM SYS$TableStatsIndex
WHERE NOT contains(tableName, "$")
GROUP BY tableName, indexName
```

Example 7-7 Size of the tables

The clause WHERE NOT CONTAINS (tableName, "$") filters out system tables by only including tables whose names do not contain the "$" character.

```
SELECT tableName,TableSize,
tableSizeWithTombstones FROM SYS$TableStatsPartition
WHERE NOT contains(tableName,"$");
```

For tables without tombstones (For example, non-multi-region tables, system tables, etc,), the metric `tableSizeWithTombstones` would be the same as the metric tableSize in the system table. The difference between the two metrics is the total storage size of tombstones in the table.

Example 7-8 Determine the size before a table export

You want to export a gigantic table to another place (another disk, kvstore, etc.), You can use `tableSize` to determine the size of the data. You can determine the size of live data without tombstone for that table since export does not copy tombstones.

```
SELECT tableName,TableSize FROM SYS$TableStatsPartition
WHERE NOT contains(tableName,"$");
```

Solid State Drives (SSDs)

If you are planning on using Solid State Drives (SSDs) for your Oracle NoSQL Database deployment, a special consideration should be taken. Because of how SSDs work, I/O latency can become an issue with SSDs over time. Correct configuration and use of `trim` can help minimize these latency issues.

Trim requirements

In general, for TRIM to be effective, the following requirements must be met:

- The SSD itself must support trim.
- Linux-kernel 2.6.33 or later.
- Filesystem ext4 (ext3 does not support trim).

Enabling Trim

The `trim` support must be explicitly enabled for the ext4 file system. You should mount the file system with trim enabled.

Diagnostics Utility

In order to catch configuration errors early, you can use this tool when troubleshooting your KVStore. Also, you can use this tool to package important information and files to send them to Oracle Support, for example.

The usage for the utility is:

```
> java -Xmx64m -Xms64m \
-jar KVHOME/lib/kvstore.jar diagnostics {setup | collect} [args]
```

Setting up the tool

You should first run the `diagnostics setup` command in order to setup the tool. This command generates the configuration file `sn-target-list` with the Storage Node target list, which contains the IP/hostname, registry ports, and root directory of SNAs in the remote machines.

The usage of this command is:

```
diagnostics setup {-add |
-list |
-delete |
-clear} [args]
```

where:

- -add

  Adds the specified information of each SNA to the `sn-target-list`. The usage is:

  ```
  setup -add -store <store name>
  -sn <SN name>
  -host <host>
  -rootdir <kvroot directory>
  [-sshusername <SSH username>]
  [-configdir <directory of configuration>]
  ```

  In the `sn-target-list`, the SNA information has the following format:

  ```
  <store name>|<sn name>|<SSH username@host>|<root directory>
  ```

  For example:

  ```
  mystore|sn3|lroot@localhost|/scratch/tests/kvroot
  ```

  > **Note:**
  >
  > You can also create and edit the `sn-target-list` manually in your preferred text editor to add or delete any SNA information.

- -list

Lists and tests the SNAs information of the `sn-target-list`. The usage is:

```
setup -list [-configdir <configuration file directory>]
            [-sshusername <SSH username>]
```

This command checks if:

- The host name is reachable or not.
- The root directory exists or not.

- -delete

  Specified to delete the information of the specified SNA from the `sn-target-list`.

  The usage of this command is:

```
diagnostics setup -delete
[-store <store name>]
[-sn <SN name>]
[-host <host>]
[-rootdir kvroot directory>]
[-sshusername <SSH username>]
[-configdir <configuration file directory>]
```

- -clear

  Specified to clear all the SNA information in the `sn-target-list`.

  The usage of this command is:

```
diagnostics setup -clear [-configdir <configuration file directory>]
```

- -configdir

  Optionally specified to change the default directory where the `sn-target-list` file is saved. If the flag is not specified, the default directory is the working directory.

Packaging Information and Files

After completing the `diagnostics setup`, you can use the `diagnostics collect` tool to package important information and files to be able to send them to Oracle Support, for example.

The usage of this command is:

```
diagnostics collect -logfiles
[-host <host name of a SN in topology>]
[-port <registry port of a SN in topology>]
[-sshusername <SSH username>]
[-username <store username>]
[-security <security-file-path>]
[-configdir <location of Storage Node target file>]
[-savedir <destination directory for log files>]
[-nocompress]
```

where:

- -logfiles

  Specified to gather log files of KVStore and pack them up into a compressed file. These files can be a part of the `KVROOT` directory or the `rnlogdir` directory, depending on what was specified when running the `makebootconfig` file.

  > **Note:**
  >
  > In old servers, `je.[info, config, stat]` files will still be a part of the environment directory.

  Available disk space in all the hosting machines and the client machine is required. If available disk space is not enough, an error message is prompted. Log files are helpful to analyze some sophisticated issues.

- -host

  Specifies the host of a Storage Node. If specified, it detects a running topology in order to update the `sn-target-list` without having to run `diagnostics setup` first. It needs to be specified with -port.

- -port

  Specifies the host of a Storage Node. If specified, it detects a running topology in order to update the `sn-target-list` without having to run `diagnostics setup` first. It needs to be specified with -host.

- -sshusername

  Specifies a SSH username to log on as in a Storage Node.

- -username

  Specifies a username to log on as in a secure deployment.

- -security

  In a secured deployment, specifies a path to the security file. If not specified in a secure store, updating the `sn-target-list` will fail.

- -configdir

  Specifies the directory which contains the `sn-target-list`. If the flag is not specified, the default directory is the working directory.

- -savedir

  Optionally used to specify the path of the directory to contain all the log files. If the flag is not specified, the default directory is the working directory.

- -nocompress

  Specifies that log files should be copied directly instead of being compressed. If the log files size is large, copying can take a while. You should use -nocompress if the remote servers do not have an unzip tool or if compress mode encounters errors.

Verifying Storage Node configuration

You can use the `diagnostics verify` tool to verify the configuration of the specified Storage Nodes. You can also check if the configuration of each Storage Node is consistent with other members of the cluster.

The usage of this command is:

```
diagnostics verify { -checkLocal | -checkMulti }
[-host <host name of a SN in topology>]
[-port <registry port of a SN in topology>]
[-sshusername <SSH username>]
[-username <store username>]
[-security <security-file-path>]
[-configdir <location of Storage Node target file>]
```

where:

- -checkLocal

  If specified, verifies the configuration of the specified Storage Nodes.

- -checkMulti

  If specified, verifies that the configuration of each Storage Node is consistent with other members of the cluster.

- -host

  Specifies the host of a Storage Node. If specified, it detects a running topology in order to update the `sn-target-list` without having to run `diagnostics setup` first. It needs to be specified with -port.

- -port

  Specifies the host of a Storage Node. If specified, it detects a running topology in order to update the `sn-target-list` without having to run `diagnostics setup` first. It needs to be specified with -host.

- -sshusername

  Specifies a SSH username to log on as in a Storage Node.

- -username

  Specifies a username to log on as in a secure deployment.

- -security

  In a secured deployment, specifies a path to the security file. If not specified in a secure store, updating the `sn-target-list` will fail.

- -configdir

  Specifies the directory which contains the `sn-target-list`. If the flag is not specified, the default directory is the working directory.