Oracle® NoSQL Database

Run Book

Release 12.2.4.5
E85376-01
February 2018

Oracle NoSQL Database Run Book, Release 12.2.4.5

E85376-01

Contents

Preface

Conventions Used in This Book v

1 Introduction

2 Software Monitoring

System Log File Monitoring 2-1
- Java Management Extensions (JMX) Monitoring 2-2

Monitoring for Storage Nodes (SN) 2-2
- Metrics 2-3
- Java Management Extensions (JMX) Notifications 2-4

Monitoring for Replication Nodes (RN) 2-18
- Metrics 2-18

Monitoring for Arbiter Nodes 2-23
- Metrics 2-23

Monitoring for Administration (Admin) Nodes 2-25
- Metrics 2-25

3 Hardware Monitoring

Monitoring for Hardware Faults 3-1
- The Network 3-1
  - Correlating Network Failure to NoSQL Log Events 3-2
  - Recovering from Network Failure 3-2
- Persistent Storage 3-2
  - Detecting and Correlating Persistent Storage Failures to NoSQL Log Events 3-3
  - Resolving Storage Device Failures 3-3
  - Procedure for Replacing a Failed Persistent Storage Device 3-6
  - Example 3-7

Servers 3-12

Detecting and Correlating Server Failures to NoSQL Log Events 3-13
Resolving Server Failures 3-13
Terminology Review 3-13
Assumptions 3-16
Replacement Procedure 1: Replace SN with Identical SN 3-16
Replacement Procedure 2: New SN Takes Over Duties of Removed SN 3-18
Examples 3-21
Setup 3-21
Example 1: Replace a Failed SN with an Identical SN 3-25
    Verification 3-29
Example 2: New SN Takes Over Duties of Existing SN 3-30
    Verification 3-34

A Third Party Licenses

Preface

This document describes how to monitor and troubleshoot 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.

1 Introduction

There are several important aspects to maintaining a highly available NoSQL Database that can service requests with predictable latency and throughput. This book outlines these aspects and describes how to maintain NoSQL Database to achieve these goals by focusing on the following topics:

- How to monitor the hardware and software in a NoSQL Database cluster.
- How to detect hardware and software failures.
- How to diagnose a hardware or software failure.
- How to restore a component, or set of components, once a failure has been detected and resolved.

The following chapters focus on the management and monitoring aspects of the Oracle NoSQL Database. The purpose here is to monitor, detect, diagnose, and resolve run-time issues that may occur with the NoSQL Database and the underlying hardware.

Though the guide briefly touches on guidelines and best practices for the application layer's role in monitoring and diagnosis, it does not provide any specific guidance as this will be dictated by the application and its requirements as set forth by the business.

2

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 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 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.
- System Monitoring Agents – Oracle NoSQL Database publishes MIBs for integration with Java Management Extensions (JMX) Management Beans for integration with JMX based monitoring solutions.
- Application Monitoring – A good proxy for the “health” of the 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 Integrating Oracle Enterprise Manager (OEM) with Oracle NoSQL Database in the *Oracle NoSQL Database Administrator's Guide*.

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 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. The following steps can be used to find the machines that are running the components of the cluster:

1. `java -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 JMX based tools, the Oracle NoSQL MIB is found in lib directory of the installation along with the JAR files for the product. For more information on JMX, see Standardized Monitoring Interfaces in the *Oracle NoSQL Database Administrator's Guide*.

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 *Oracle NoSQL Database Concepts Manual*.

See the following sections:

- Metrics
- Java Management Extensions (JMX) Notifications

Metrics

- 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 an 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) – Although this state exists in the category, it is typically never seen 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 represents the network interface name that is utilized by the replication subsystem 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 Storage Node Parameters section in the *Oracle NoSQL Database Administrator's Guide*. If users

encounter a problem that suggests that the HA host name has been specified incorrectly, they should first check that they have provided the correct value in the call to the `makebootconfig` command. The user can also change the value later by modifying the `haHostname` parameter. For more information, see makebootconfig section in the *Oracle NoSQL Database Administrator's Guide*.

- snHaPortRange – The range of ports that replication nodes use to communicate amongst 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 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, but can be modified by the administrator if the hardware configuration is changed after the store is initialized.
- snCollectorInterval – The interval that all nodes are utilizing 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 via 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 section in the *Oracle NoSQL Database Concepts Manual*.

See the following section:

- Metrics

Metrics

- 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 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 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 unavaialble. For more information, see Arbiter Nodes section in the *Oracle NoSQL Database Concepts Manual*.

See the following section:

- Metrics

Metrics

- 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 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 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 section in the *Oracle NoSQL Database Concepts Manual*.

See the following section:

- Metrics

Metrics

The following metrics are accessible through JMX and are intended to be used for the monitoring of the 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 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.

Note that the corruption situations are difficult to handle, but this is rare and require 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 sections Admin Parameters and Admin Restart in the *Oracle NoSQL Database Administrator's Guide*.
- adminLogFileLimit – A logging config parameter that represents the maximum size of a single log file in bytes. For more information on `adminLogFileLimit`, see sections Admin Parameters and Admin Restart in the *Oracle NoSQL Database Administrator's Guide*.
- 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.

3

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 -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  \
       -storagedir /disk1/ondb/data \
       -storagedir /disk2/ondb/data
```

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 -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 \
     -storagedir /tmp/sn1/disk1/ondb/data \
     -storagedir /tmp/sn1/disk2/ondb/data
```

On Win_B

```
java -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 \
     -storagedir /tmp/sn2/disk1/ondb/data \
     -storagedir /tmp/sn2/disk2/ondb/data
```

On Win_C

```
java -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    \
     -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.

On Win_A

```
> nohup java -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot -config config1.xml &
```

On Win_B

```
> nohup java -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot -config config2.xml &
```

On Win_C

```
> nohup java -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 -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 -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.

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. Combined with the information contained in the Oracle NoSQL Database Administrator's Guide, 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 -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:

   ```
   > nohup java -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 -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 -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  \
       -storagedir /disk1/ondb/data
```

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,

   ```
   > nohup java -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 -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 -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 \
     -storagedir /tmp/sn1/disk1/ondb/data
```

On Win_B

```
java -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 \
     -storagedir /tmp/sn2/disk1/ondb/data
```

On Win_C

```
java -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 \
     -storagedir /tmp/sn3/disk1/ondb/data
```

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.

Win_A

```
> nohup java -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config1.xml &
```

Win_B

```
> nohup java -jar /opt/ondb/kv/lib/kvstore.jar start \
             -root /opt/ondb/var/kvroot \
             -config config2.xml &
```

Win_C

```
nohup java -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 -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/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 -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 -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 -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:

```
> nohup java -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 -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 -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 -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 \
     -storagedir /tmp/sn2/disk1/ondb/data
```

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,

```
> nohup java -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 -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/dsk1/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/dsk1/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
```

A

Third Party Licenses

All of the third party licenses used by Oracle NoSQL Database are described in the License document.