

Getting Started with **Oracle NoSQL Database**

Maqsood Alam
Ashok Joshi
Aalok Muley
Chaitanya Kadaru

Foreword by Andrew Mendelsohn
Executive Vice President, Oracle Database Server Technologies

Oracle
Press™

ORACLE®

Oracle Press™

Getting Started with Oracle NoSQL Database

Maqsood Alam

Aalok Muley

Ashok Joshi

Chaitanya Kadaru



New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Copyright © 2014 by McGraw-Hill Education (Publisher). All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-0-07-183572-5

MHID: 0-07-183572-5

e-book conversion by Cenveo® Publisher Services

Version 1.0

The material in this e-book also appears in the print version of this title: ISBN: 978-0-07-181653-3,

MHID: 0-07-181653-4

McGraw-Hill Education e-books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative, please visit the Contact Us pages at www.mhprofessional.com.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. All other trademarks are the property of their respective owners, and McGraw-Hill Education makes no claim of ownership by the mention of products that contain these marks.

Screen displays of copyrighted Oracle software programs have been reproduced herein with the permission of Oracle Corporation and/or its affiliates.

Information has been obtained by McGraw-Hill Education from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill Education, or others, McGraw-Hill Education does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

Oracle Corporation does not make any representations or warranties as to the accuracy, adequacy, or completeness of any information contained in this Work, and is not responsible for any errors or omissions.

TERMS OF USE

This is a copyrighted work and McGraw-Hill Education (“McGraw-Hill”) and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill’s prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED “AS IS.” MCGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

About the Authors

Maqsood Alam is a Director of Product Management at Oracle and has over 17 years of experience in architecting, building, and evangelizing enterprise and system software. Maqsood is a pure technologist at heart and has a wide range of expertise, ranging from parallel and distributed systems to high performance database applications and big data. His current initiatives at Oracle are focused on Oracle NoSQL Database, Oracle Exadata, Oracle Database 12c, and the Oracle Big Data Appliance. He is a coauthor of the book *Achieving Extreme Performance with Oracle Exadata* published by McGraw-Hill Education, and also the author of several whitepapers and best practices dealing with various Oracle technologies. He is an Oracle Certified Professional and holds both bachelor's and master's degrees in computer science.

Aalok Muley is a Senior Director of Product Management at Oracle. He is responsible for driving adoption of Oracle's family of database products: Oracle NoSQL Database, Oracle Big Data Connectors, Oracle Database 12c, and engineered systems such as Oracle Big Data Appliance and Oracle Exadata. Aalok has over 19 years of experience; he has led teams working on database industry standard benchmarks, database product development, and Fusion Middleware technologies. He has been part of the technology integration of many Oracle acquisitions. As part of the product development organization, Aalok is currently focused on working closely with partners and customers to design high-throughput, highly available enterprise-grade solutions. He holds a master's degree in computer engineering from Worcester Polytechnical Institute in Massachusetts.

Ashok Joshi is the Senior Director of Development for Oracle NoSQL Database, Berkeley DB, and Database Mobile Server. Ashok has been involved in database systems technology for over two decades as an individual contributor, as well as in a management role. Ashok has made extensive contributions to indexing, concurrency control, buffer management, logging and recovery, and performance optimizations in a variety of products, including Oracle Rdb, Oracle Database, and Sybase SQL Server. He is the author or coauthor of several papers as well as 12 patents on database technology. Ashok graduated from the Indian Institutes of Technology, Bombay with a bachelor's degree in electrical engineering and received a master's degree in computer science from the University of Wisconsin, Madison.

Chaitanya Kadaru is an accomplished software professional with over 12 years of industry experience. He has spent the majority of his time with Oracle, working in databases, middleware, and Oracle applications in various roles, including developer, evangelist, pre-sales, consulting, and training. He recently co-founded Extuit, a premier

Oracle consulting company, and has architected solutions involving engineered systems, such as Oracle Exadata, Oracle Exalogic, and Oracle Big Data Appliance, for a wide range of customers. He is currently responsible for a large-scale Oracle Database consolidation to Oracle Exadata for a large financial services company. Chaitanya holds a bachelor's degree in engineering from BITS, Pilani, and a master's degree in information systems from Carnegie Mellon University.



Contents

Foreword	vii
Introduction	ix
4 Oracle NoSQL Database Installation and Configuration	1
Oracle NoSQL Database Installation	2
Download Oracle NoSQL Database Software	4
Software Installation	4
Oracle NoSQL Database Administration Service	6
Create the Boot Configuration	8
Perform Sanity Checks	13
Oracle NoSQL Database Configuration	13
Plans	14
Configuration Steps	15
Automating the Configuration Steps	21
Verifying the Deployment	22
Summary	25
5 Getting Started with Oracle NoSQL Database Development	27
Developing on KVLite	28
A Basic Hello World Program	31
How to Model Your Key Space	34
The Basics of Reading and Writing a Single Key-Value Pair	37
Consistency and Durability from the Programmer's Perspective	38
Durability	39
Consistency	41
Summary	44

9 Advanced Topics	45
Hadoop Integration	46
RDF Graph	49
Integration with Complex Event Processing	51
Database External Tables	53
Define an External Table	55
Edit the Configuration File	56
Publish the Configuration	56
Test the nosql_stream Script	56
Use the External Table to Read Data from Oracle NoSQL Database	57
Summary	57
Summary	59



Foreword

We are at the dawn of the era of the Cloud and Big Data. The functioning of virtually every organization depends on the availability of their data. To service this explosion of data, users are turning to a wide range of data management solutions ranging from file systems to NoSQL databases to relational databases.

NoSQL databases are increasingly becoming part of the enterprise developer's toolkit. There are three main categories of NoSQL databases: key-value, document, and graph. They all have their pros and cons, and you need to pick the right database for the problem at hand. In general, NoSQL databases offer a cost-effective way to solve simpler data management problems with a less advanced feature set. They all complement, rather than replace, the relational database.

Oracle NoSQL Database is a key-value database, which is the most general purpose of the three types of NoSQL databases. Key-value databases apply when the use case is to manage a large collection of data objects consisting of a primary key and a string of bytes. For example, key-value databases are used for managing user profiles and shopping carts on e-commerce sites, detecting fraud in credit card transactions, and logging and providing rapid access to sensor data. So if you've ever shopped on the web, paid with a credit card, or flown on an airplane, then Oracle NoSQL Database may have played a small part in that experience.

We have focused on making sure that Oracle NoSQL Database meets the needs of the enterprise developer and operations by including the following capabilities rarely found in other key-value databases:

- ACID transactions—because code to handle recovery from failures belong in the database, not in your applications.
- Enterprise stack integration—because no organization needs yet another silo of data, we have integrated Oracle NoSQL Database with Oracle Database (use Oracle SQL to query NoSQL data!), Oracle Enterprise Manager, Hadoop, and Oracle Coherence.
- Scalable performance with predictable latency—because your new database must continue to deliver as your customer base and data grow.

This ebook covers the installation of Oracle NoSQL Database, guides you through the basics of developing a NoSQL database application, and concludes with topics on integration of Oracle NoSQL Database with external systems. I would encourage you to visit Oracle.com and perform a free download of the product so you can try it for yourself.

ANDREW MENDELSON
Executive Vice President
Oracle Database Server Technologies



Introduction

The roots of NoSQL databases can be traced back to the mid-60s when databases such as MUMPS (aka M Database) and PICK (aka MultiValue) came into existence. The main purpose at that time was to build a schema-less implementation of the relational database management system (RDBMS) that would be lightweight and optimized, highly scalable, provide high-transaction throughput, and most importantly, provide an alternative method for data access than the traditional SQL interface.

The term “NoSQL” was initially coined by Carlo Strozzi in 1998 when he named his lightweight, open source relational database management system as NoSQL. Although his database still used the relational database paradigm, his main intention was to provide an alternative interface for data access besides SQL. The term “NoSQL” later resurfaced in 2009 as an attempt to categorize the large number of emerging databases that defied the attributes of traditional RDBMS systems. The key attributes of NoSQL databases are mainly to support non-relational structures; provide a distributed implementation that is highly scalable; and at most times, to not support the key transaction guarantee features inherent to RDBMS systems, such as ACID properties (atomicity, consistency, isolation, and durability).

Berkeley DB (BDB) originated at University of California, Berkeley (1986–1994) as a byproduct of the effort to convert BSD 4.3 (aka Berkeley Unix) to BSD 4.4. In 1996, Netscape requested a few additional enhancements to BDB in order to make it usable in the browser, which led to the formation of Sleepycat Software. The purpose of Sleepycat was to provide enterprise-level support to BDB and to make further enhancements to the product. Sleepycat Software was later acquired by Oracle in February 2006.

Oracle NoSQL Database is a distributed key-value database that uses the BDB engine underneath the covers, and provides a variety of additional features such as dynamic partitioning, load balancing, predictable latency, monitoring, and other features that enable Oracle NoSQL Database to be used in enterprise-level deployments.

Here is an overview of the chapters covered in this book:

Chapter 4: Oracle NoSQL Database Installation and Configuration

This chapter covers the installation and configuration steps of Oracle NoSQL Database. You start with downloading the software, proceed through the software installation process, and finally wrap up by configuring a distributed cluster of Oracle NoSQL Database.

Chapter 5: Getting Started with Oracle NoSQL Database Development

In this chapter, you are introduced to the basics of NoSQL development. You start with a basic Hello World program and learn about modeling the key space. The basics of reading and writing data are also covered in this chapter.

Chapter 9: Advanced Topics

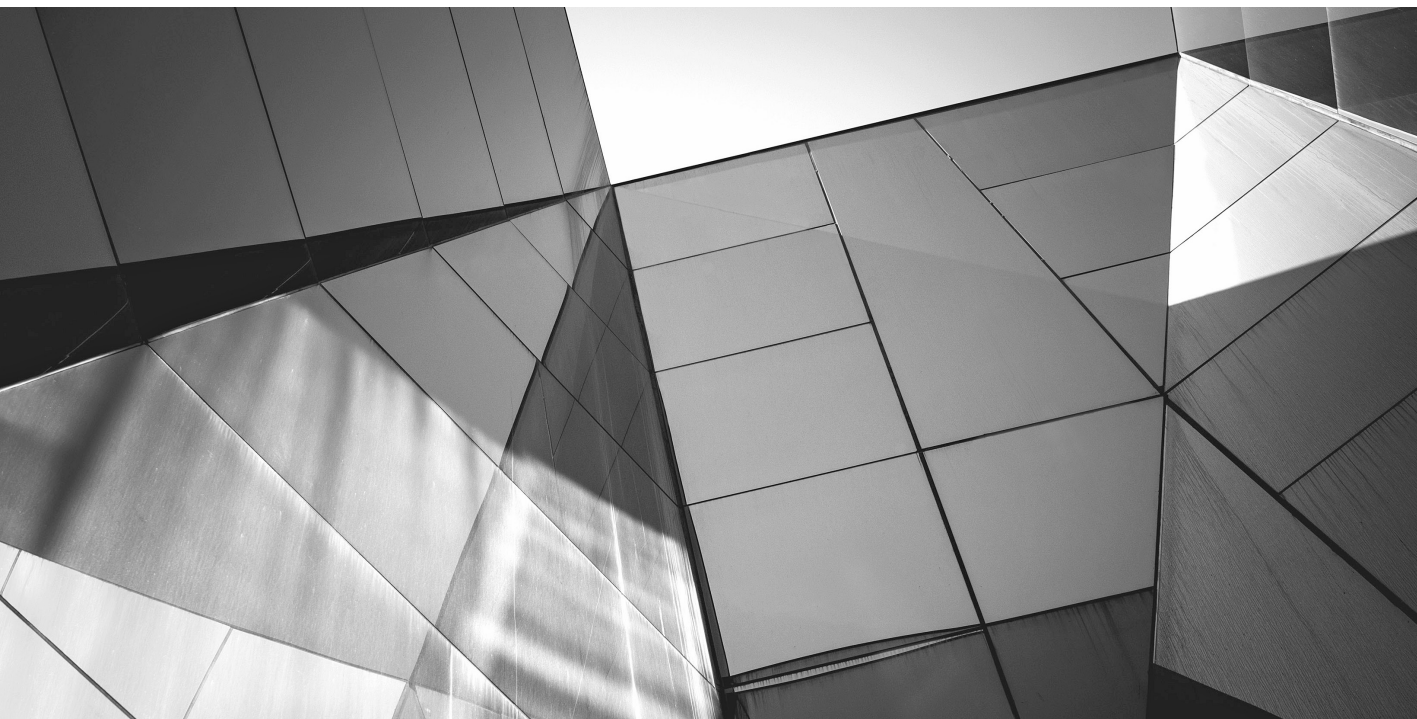
In this chapter, we cover topics related to integration of Oracle NoSQL Database with other products commonly found in enterprise datacenters, such as the Oracle Relational Database Management System, Oracle Event Processing, and Hadoop.

Intended Audience

This book is suitable for the following readers:

- Developers who need to write NoSQL applications using Oracle NoSQL Database
- Big data architects looking for different methods of storing unstructured data for real-time analysis
- Database administrators who like to get into installation, administration, and maintenance of NoSQL databases
- Technical managers or consultants who need an introduction to Oracle NoSQL Database and to see how it compares to other NoSQL databases

No prior knowledge of Oracle NoSQL Database, big data, or any NoSQL database technology is assumed.



CHAPTER 4

Oracle NoSQL Database Installation and Configuration

Prior to deploying Oracle NoSQL Database, it is imperative that a right-sized database topology is architected using the best practices of sizing Oracle NoSQL databases that satisfies the availability, reliability, and performance requirements as set forth by the Oracle NoSQL Database application. The total number of Storage Nodes and their respective hardware configurations, such as CPU, memory, and disks, must be defined by now. Otherwise, poorly sized systems are prone to performance and stability issues when subjected to production workloads. Please refer to Chapter 8 for the best practices of sizing Oracle NoSQL Database deployments.

The Oracle NoSQL Database deployment is typically a two-phase process. The first phase comprises the steps to install the Oracle NoSQL Database software on the individual Storage Nodes and starts the required processes. If your intention was only to use *KVLite*, you are done after this phase and may continue with developing your applications. *KVLite* is a single node and non-distributed version of Oracle NoSQL Database suitable for development and learning purposes. Further details on *KVLite* and application development are provided in Chapter 5.

The steps outlined in the second phase build a *distributed* and *clustered* version of Oracle NoSQL Database using a set of Storage Nodes. A distributed version is essential for achieving high availability, scalability, reliability, and performance requirements—the key characteristics of an enterprise-grade production system.

Oracle NoSQL Database Installation

Oracle NoSQL Database installation in itself is a simple process and can be completed very quickly, but the steps that occur before the installation may take most of your time. Verifying the installation prerequisites such as the hardware, network, and operating system is essential for ensuring a successful and, more important, stable installation. Furthermore, the operating system must be configured with the appropriate set of packages that are required by the Oracle NoSQL Database software. Hence, you must ensure that all Storage Nodes of the key-value store satisfy the following requirements:

- **Operating system** Oracle Linux and Oracle Solaris are the officially supported operating systems for Oracle NoSQL Database. It may be true that Oracle NoSQL Database, as a Java application, can run on any platform that supports a Java Virtual Machine (JVM), but the chances of running into issues on unsupported platforms are much higher as Oracle Corporation tests its software products only on supported platforms. If you run into issues, you need to reproduce the problem on a supported platform before Oracle support can investigate and analyze the root cause. Therefore, it is very important to ensure that the underlying operating system (OS) is fully supported by Oracle.

- **Clock synchronization** A distributed computing cluster, such as the Oracle NoSQL Database cluster, requires the system clocks of individual Storage Nodes to be synchronized with a global time clock. This is essential to ensure inter-node communications between processes running on different Storage Nodes have the same understanding of time. The clock *drift* (time difference) between all the nodes in the cluster should ideally be close to zero. NTP (Network Time Protocol) is a reliable mechanism for synchronizing time between multiple nodes and readily available on many OS installations, including Oracle Linux and Oracle Solaris. As a best practice, ensure that NTP is set on all the Storage Nodes with an optimal synchronization interval, thereby reducing the clock drift to as close to zero as possible.
- **Java** Ensure that Java SE 6 (JDK 1.6.0 u25) or later is installed on the Storage Nodes. Otherwise, install the correct version of Java from Oracle's download site (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>). You may use the `java -version` command to check the version of Java installed on the system.
- **File system for KVHOME and KVROOT** *KVHOME* is the file system location storing the Oracle NoSQL Database *software binaries* and *KVROOT* is the location for storing Oracle NoSQL Database configuration files and also serves as the default location for key-value pair *data*. Identify the directories on the file system to be used for these locations and ensure that they have enough space to hold their contents. It is recommended that you have both these locations on the local file system and not on a shared file system such as the Network File System (NFS), as sharing of I/O resources by multiple applications often leads to contention issues.
- **Network ports** The Storage Nodes and Replication Nodes of the key-value store communicate with each other using the TCP/IP protocol over Ethernet ports. Ensure that enough ports are available across all the Storage Nodes, and that the ports are free and unallocated and not blocked by network firewalls. The port numbers are user configurable and set during the initial configuration of Oracle NoSQL Database. Further details on assigning network ports are discussed later in this chapter.

**NOTE**

If you were to use the integration features of Oracle NoSQL Database such as the integration with Hadoop, Oracle Loader for Hadoop, and the Oracle 11gR2 Database via database external tables, you would require an installation of Oracle NoSQL database software on those systems as well.

Download Oracle NoSQL Database Software

The Oracle NoSQL Database software comes in two editions, the Community Edition (CE) and the Enterprise Edition (EE). The Enterprise Edition includes all features of the Community Edition, plus a few additional features related to the integration of NoSQL with other products, such as the Oracle Database and Oracle Event Processing. The EE also provides access to enterprise-level Oracle Support. Refer to Chapter 2 for further details on CE vs. EE.

No matter what edition you choose to install, the steps for installation and configuration do not deviate much. The Enterprise Edition may require additional steps to configure the integration-specific features, but those steps are covered later in the book.

You may download the Oracle NoSQL Database Community Edition and the Enterprise Edition from the Oracle Technology Network, and to suit your needs, you have an option of downloading a Tar or a Zip archive.



NOTE

Oracle NoSQL Database Community Edition is preinstalled on the Oracle Big Data Appliance (BDA) and configured during the BDA deployment process upon the customer's request.

Software Installation

By now you must have picked the directory locations for KVHOME, KVROOT, and the Oracle NoSQL Database *data* directory (unless using KVROOT as the default) on each Storage Node. Although it is not an absolute requirement, it is preferred that you have both KVHOME and Oracle NoSQL Database *data* directories on a local file system instead of a shared file system such as NFS. For KVHOME, this ensures that patching and upgrading of the software are done in a rolling fashion and with minimal downtime, and moreover, eliminate a single point of failure as one set of binaries is shared across all nodes. Using a local file system for the data directory ensures that the disks are dedicated solely for NoSQL, thereby eliminating unforeseen IO bottlenecks that may be introduced by other applications.



NOTE

KVROOT is the default location for storing Oracle NoSQL Database data and can be overwritten by using the `-storagedir` parameter of `makebootconfig`. Refer to the section “Create the Boot Configuration” for further details.

You must also ensure that KVHOME and Oracle NoSQL Database data are located in separate directories, and are consistent (that is, follow the same paths) across all the Storage Nodes in the key-value store by following standard naming conventions. This is a recommended best practice for avoiding configuration errors and helps with easier manageability. Also ensure that there is enough space allocated by the OS for these locations. The Oracle NoSQL Database data directory requires much more space than KVHOME as it stores the application key-value pair data. KVHOME, on the other hand, requires less than 50MB of disk space (for the current 11gR2 release).



NOTE

You do need superuser privileges to install Oracle NoSQL Database. The installation can be performed as a regular OS user.

Once you have downloaded the software, copy the Zip archive to all the Storage Nodes and move them to the root directory of KVHOME (the mount point or the parent folder of the intended KVHOME). Extract the package contents using the appropriate unzip utilities (gunzip followed by tar for *.gz and unzip for *.zip), and when the extraction completes, the KVHOME directory is created automatically. Repeat this process on all Storage Nodes, and once the extraction succeeds on all the nodes, you have completed the Oracle NoSQL Database software installation. The installation process is really that simple; there are no screens or parameters to configure.

The following example outlines the steps to install Oracle NoSQL Database version 2.0.39 Enterprise Edition using a *.gz file. The root directory of KVHOME used in the example is /opt/kvhomes, and at the end of the installation, KVHOME is created as /opt/kvhomes/kv-2.0.39.




```
$ cd /u01/kvhomes
$ gunzip kv-ee-2.0.39.tar.gz
$ tar xvf kv-ee-2.0.39.tar
$ ls -F /u01/kvhomes/kv-2.0.39
build.xml doc/ examples/ exttab/ lib/ LICENSE.txt README.txt
```

The actual packages and directories created in KVHOME depend upon the release, version, and the Oracle NoSQL Database edition (CE or EE). The KVHOME directory itself depends on the release and the version and typically follows the convention kv-M.N.O, where M is the software release, and N and O the major and minor release numbers, respectively. This naming scheme ensures that future upgrades do not overwrite existing installations and that previous installations can easily be identified.

6 Getting Started with Oracle NoSQL Database

Now that the installation is complete, you may quickly verify the installation by running a supplied test application called `kvclient.jar`, which is a part of the Oracle NoSQL Database software. The `kvclient.jar` application prints the current release and version of the Oracle NoSQL Database software on the screen. You may run `kvclient.jar` from one of the nodes (but ideally on all nodes) and ensure that the output you get follows the format of `version.M.N.O`, where the version is 11gR2 for the current release. The following example illustrates the use of `kvclient.jar`:



```
$ java -jar /u01/kvhomes/kv-2.0.39/lib/kvclient.jar
11gR2.2.0.39
```

Oracle NoSQL Database Administration Service

The Oracle NoSQL Database Administration Service is a process that runs on the Storage Nodes, and is in charge of a variety of administration activities on the Oracle NoSQL Database, such as instance startup/shutdown, initial configuration, ongoing modifications to the configuration, and monitoring system performance, without the need for manually writing complex scripts and commands. The Administration Service is also responsible for collecting and maintaining database performance statistics, and also for logging important system events, and thereby assisting with online monitoring and helping tune database performance. The Administration Service internally uses a database called the *Administration Database* to store configuration and monitoring data.



NOTE

The Administration Database under the covers uses a key-value store.

The availability of the Administration Service is critical to performing maintenance operations on the key-value store; therefore, it is important to have multiple Administration Services deployed across the store to ensure high availability. The best practice is to have a minimum of three Administration Services so at least one service is predicted to be available in a given time. The availability of the Administration Service is not to be confused with the availability of Oracle NoSQL Database itself, as normal database activities such as reads, writes, and replication occur without the intervention of the Administration Service.


The Administration Service is accessible from a command line interface called the *Administration CLI* or *CLI*, and a web-based console called the *Web Administration Console*.

Administration Command Line Interface (CLI)


The Administration CLI supports all administration activities on the key-value store. To start the Administration CLI, execute the `runadmin` class of `kvstore.jar`. The `runadmin` interacts with the Administration Service and provides the CLI prompt (the `kv->` prompt). However, when you are configuring the key-value store for the first time, you do not have any Administration Services running, but you still need to start the CLI and to configure other parameters, for example the key-value store name. It seems like a catch-22, and the solution to that is to use the built-in Administration Service of the Storage Node Agent process called the Bootstrap Administration Service. A default Administration Database gets created by the Bootstrap Administration Service and later, when the Administration Service gets deployed by the CLI commands, the final set of Administration Services gets created and the Bootstrap Administration Service is stopped and no longer used.

The Bootstrap Administration Service is automatically started upon starting the Storage Node Agent process when the administration port is specified in the boot configuration file (boot configuration is discussed next). Therefore, you need to ensure that the Storage Node Agent service is started prior to executing `runadmin` because `runadmin` communicates with the SNA using the registry port.

The CLI can be invoked mainly in three modes: an interactive mode, a single command mode, and a script mode. The *interactive mode* is most commonly used and is the only mode that provides a command prompt (the `kv->` prompt). Users input commands at the prompt, one at a time, and the commands are executed in the background (although they may run in the foreground using the `-wait` flag). The following is an example of starting the CLI using `runadmin` to access the Storage Node Agent on `node01` with the registry port 5000:

```
 java -jar KVHOME/lib/kvstore.jar runadmin -host node01 -port 5000
kv->
```

The *single command mode*, on the other hand, runs a single command directly at the OS command prompt while invoking the CLI. It passes the CLI command as a parameter to `runadmin`. Once the single CLI command completes its execution, the control is returned back to the OS. If the command completes successfully, the exit code returned to the OS is 0, and if it encounters an error, the exit code is a value other than 0. The general usage of invoking the CLI in the single command mode is as follows:


```
 java -jar KVHOME/lib/kvstore.jar runadmin
-host <hostname> -port <port> [single command and arguments]
```

where `<hostname>` and `<port>` are the SNA hostname and the registry port, respectively.

Last, the *script mode* is very similar to the single command mode, but it runs a script containing multiple CLI commands instead of running only a single CLI

8 Getting Started with Oracle NoSQL Database

command. With the script mode, it becomes easy to automate repetitive tasks, or run tasks in a batch mode, without requiring direct supervision of an administrator. As with the single command mode, the control returns to the OS prompt once the script completes. The script file is specified using the `load -file` switch when invoking `runadmin`. The following is a typical example of CLI in script mode:

```
 java -jar KVHOME/lib/kvstore.jar runadmin  
-host <hostname> -port <port> load -file <path-to-script>
```

The configuration steps in this chapter are mainly performed using the interactive mode. The CLI allows a number of commands, and some commands may even have subcommands. Commands are also grouped by the specific set of functions they perform; for example, the `show` command displays the state of the key-value store and its components, whereas the `ddl` command manipulates key-value store schemas. You may use the `help` command to discover all commands allowed by the CLI, or append a `-help` flag to a specific command to display its usage syntax. For a complete listing of all CLI commands, refer to the *Oracle NoSQL Database Administration Guide* provided by Oracle. Only the important CLI commands that get you through the configuration steps are covered in this chapter.

Web Administration Console

Besides the Administration CLI, the *Web Administration Console* can also be used to administer the Oracle NoSQL Database. The current release of the Web Administration Console supports mainly read-only type administration activities such as browsing the key-value store topology, monitoring plan executions, and browsing at the cluster-wide log file; however, it does not support activities related to the store configuration or modification. These are the tasks well handled by the Administration CLI.

The Web Administration Console is part of the Administration Services and uses a port to listen for HTTP requests from web clients, called the *Administration Port*. To access the Web Administration Console, point an HTML-based web interface to the host running the Administration Service and specify the Administration Port. For instance, you would use the URL `http://node01:5001` to access the Web Administration Console with the Administration Service running on `node01` and listening to port `5001`.

Create the Boot Configuration

So far, you only have the Oracle NoSQL Database software installed on the Storage Nodes. The next step is to perform a few additional tasks, such as specifying the network ports for Storage Node Agents (SNAs), the Web Administration Console and replication ports, and the `KVROOT` location to store the configuration files and, optionally, the data files. Although these tasks are related to configuration, they are still part of the first phase and are required to be completed before starting the main

configuration steps, which is mainly focused on building a distributed cluster of Oracle NoSQL Database.

The Storage Node Agent (SNA) process and Administration Service utilize a configuration file at startup for setting up the network ports and other initialization parameters. This configuration file is also referred to as the “boot config” file and is located in the KVROOT directory with a default name of `config.xml`. For a freshly installed Storage Node, the boot config file does not exist, and needs to be created manually using the `makebootconfig` utility. The `makebootconfig` utility has the following syntax:

```
java -jar KVHOME/lib/kvstore.jar makebootconfig [-verbose]
-root <rootDirectory> -host <hostname> -hahostname <hostname>
-harange <startPort,endPort> -port <port> [-admin <adminPort>]
[-config <configFile>] [-storagedir <path>] [-capacity <n_rep_nodes>]
[-num_cpus <ncpus>] [-memory_mb <memory_mb>]
[-servicerange <startPort,endPort>]
[-mgmt {snmp|jmx|none}] [-pollport <snmp poll port>]
[-traphost <snmp trap/notification hostname>]
[-trapport <snmp trap/notification port>]
```

The following are the details on the commonly used parameters of `makebootconfig`. It is a good idea to ensure that you have this information before creating the initial boot configuration.

- **-root <rootDirectory>** This is the KVROOT location that stores the configuration files and, optionally, the key-value pair data (unless using the `-storagedir` clause to overwrite data storage location). If using KVROOT to store data, ensure that the disk space is large enough to accommodate the key-value pairs destined for the Storage Node. The storage location should also guarantee the I/O performance to satisfy the application requirements. The examples in this book assume that the KVROOT directory is built on `/u02/kvroot` on each Storage Node.
- **-port <port>** Each Storage Node runs a Storage Node Agent (SNA) process to facilitate communications between other SNA processes and the client applications. The SNA listens to a *registry port* specified using the `-port` parameter. The registry port typically used in the examples is 5000.

NOTE

All ports required for Oracle NoSQL Database should be unallocated and unused by other applications, and not blocked by network firewalls. Ensure this is true on all the servers that are part of the Oracle NoSQL Database cluster.

- **-admin <adminPort>** This is the port used by the Administration Service to listen for HTTP connections from web clients, also called the *Administration Port*. For initial configuration, use this option only on the first Storage Node to configure the Bootstrap Administration Service. Once the database is fully deployed, configure multiple Administration Services for HA and ensure that they all use the same Administration Port. Otherwise, it becomes difficult for users to identify the port of the next healthy Administration Service when the Storage Node running the primary fails. The Administration Port typically used in the examples is 5001.
- **-harange <startPort, endPort>** Each Storage Node requires a set of ports (specified using a range, called the *HA Range* ports) to be used by the Replication Nodes and Administration Services for facilitating the replication of user data (key-value pairs). The SNA internally manages these ports and reserves one for the Administration Service and one for each Replication Node (RN) hosted by the Storage Node (SN). If there are multiple RNs per SN (yes, this is possible, as discussed in the corresponding note), then you need to ensure that the range specified has enough ports that are equal to the maximum number of RNs that the SN may ever host, plus one for the Administration Service, as there could be at most one Administration Service per SN. The ports are specified as a range using “startPort, endPort.” The examples in this book use “5010, 5020” as the HA Range ports.



NOTE

Multiple Replication Nodes can be configured per Storage Node. Although this is not recommended as a best practice, in certain cases where there is ample CPU, memory, and I/O resources on the physical server, it could be justified.

- **-servicerange <startPort, endPort>** The Storage and Replication Node services internally initiate Java-based RMI calls across the nodes in the cluster (separate from data replication operations). Using the **-servicerange** flag, you may specify a second range of free ports to be used specifically for such RMI invocations. If **-servicerange** is not used, the RMI ports are randomly allocated, thereby making it difficult for network administrators to configure firewall rules. This parameter is useful when there is a network firewall configured between the clients and the Storage Nodes and it restricts access to specific ports. Using this parameter forces RMI calls over a predefined range of ports upon which firewall rules can be proactively defined.

- **SNMP configuration** You could optionally configure SNMP monitoring tools (such as Oracle Enterprise Manager or other third-party SNMP- or JMX-based tools) to capture critical events and alerts that arise within the Oracle NoSQL Database cluster. For this to occur, the critical events need to be propagated outside the Oracle NoSQL Database processes and memory structures through the built-in SNMP/JMX agent (specified using the `-mgmt` flag) via an *SNMP port* (specified using the `-pollPort` flag) to communicate with an external SNMP management system (specified using `-trapHost` and `-trapPort`).
- **`-num_cpus <ncpus>`** This parameter is used when you have multiple Replication Nodes on a Storage Node. The `-num_cpus` specifies the total number of processors on the server available to all the Replication Nodes so the CPU resources can be appropriately allocated. This value defaults to 0, in which case the OS is queried to get the number of processors on the machine. The best practice, however, is to not default it to 0 because there is a chance that the user installing Oracle NoSQL Database may not have access to OS utilities such as `/proc/cpuinfo` (for Linux) as the file is typically owned by root, which may result in the query failing.
- **`-memory_mb <memory_mb>`** The Replication Node cache and heap sizes are set accordingly with the total memory available on the server. Use this parameter to specify the total memory on the server in megabytes. If not specified, it defaults to 0. This makes the system query the OS to get the actual value but only when Oracle Hotspot JVM is used. The best practice is to follow the same guidelines mentioned earlier and specify the total memory using this flag instead of relying on the operating system. This value becomes even more important when the Storage Node hosts multiple Replication Nodes and the system needs to effectively manage the memory between multiple Replication Nodes.
- **`-hahostname <hostname>`** This flag is used for specifying a separate network interface for routing replication traffic. This comes in handy when segregating client requests from internal replication traffic within the Replication Nodes. If `-hahostname` is not set, it defaults to the hostname specified using the `-host` flag.
- **`-capacity <capacity>`** This parameter is optional and, when set, specifies the total number of Replication Nodes a Storage Node can support. Capacity is set to values greater than 1 usually when the Storage Node has sufficient disk, CPU, and memory, and can support multiple Replication Nodes.

12 Getting Started with Oracle NoSQL Database

- **-storagedir <path>** This parameter is used to override the default Oracle NoSQL Database data directory for the Replication Node. As a best practice, always specify `-storagedir` instead of defaulting to `KVROOT`, and decouple the Oracle NoSQL Database configuration data location with the key-value pair data. When multiple Replication Nodes are to be configured per Storage Node, use this parameter along with the `-capacity` parameter. If `-storagedir` is not specified and you have specified `-capacity` greater than 1, the data storage directory for each Replication Node gets created under the `KVROOT` directory with default names. You may use this parameter multiple times in the command to specify multiple directories, one each per Replication Node, not to exceed the `-capacity` parameter. For example, if the Storage Node houses eight disks, you would specify `-capacity 8` and have eight `-storagedir` arguments, one per each Replication Node.

Once you have obtained the preceding information, proceed with creating the boot config file, as shown in the following example. The example calls the `makebootconfig` command with `/u02/kvroot` as the `KVROOT`, the SNA running on port 5000, the Administration Service running on port 5001, the range for `harange` ports as 5010–5020, and the capacity of 1.

```
$> mkdir -p /u02/kvroot
$> java -jar /u01/kvhomes/kv-2.0.39/lib/kvstore.jar makebootconfig
    -root /u02/kvroot \
    -host <hostname> \
    -port 5000 \
    -admin 5001 \
    -harange 5010,5020 \
    -capacity 1 \
    -num_cpus 0 \
    -memory_mb 0
```

The next step is to start the Oracle NoSQL Database Storage Node agent (SNA) processes on each of the Oracle NoSQL Database nodes. As mentioned earlier, the SNA automatically starts the Bootstrap Administration Service if the `-admin` parameter is specified at the time of the creation of the boot config file. You can use the `start` utility to start SNA processes, and also remember to start the SNA on all Storage Nodes that will be used to configure the Oracle NoSQL Database in the next section.

The following example starts the SNA process using `/u02/kvroot` as the `KVROOT`:

```
$> nohup java -jar /u02/kvroot/lib/kvstore.jar start -root /u02/kvroot &
```

**NOTE**

It is important to run the `start` command in the background and preferably use `nohup` to avoid process hang-ups. Also, you should configure the nodes to start the SNA process automatically at boot time, using OS startup utilities such as `init.d` for Linux.

Perform Sanity Checks

It is important to test the installation before proceeding with the configuration steps. There are several ways to perform sanity checks on the Storage Nodes and ensure that the nodes are running the required services and are void of any setup- and installation-related issues. The tests that you may perform at this stage should ensure that the host, operating system, and the Oracle NoSQL Database software processes are alive and healthy.

Use the JVM process status tool (`jps`) to check the Java processes running on the host. The SNA Agent process, the Administration Service, and the Replication Nodes will each have a Java process that runs on the operating system and should be visible on the output. At a minimum, you should see the Storage Node Agent process running with the name `ManagedService` and a class of `RepNode`, and if you have configured the Administration Service, you should see a second `ManagedService` process with a class of `Admin`. You may also see a `kvstore.jar` process if you have configured a distributed key-value store and a `kvlite.jar` process if you have started the KVLite database.

At the OS command prompt, run `jps -m`, as shown here:



```
$> jps -m
5705 kvstore.jar start -root /u02/nosql/kvroot
5945 ManagedService -root /u02/nosql/kvroot/movielite/snl -store
      movielite -class RepNode -service rg1-rn1
5757 ManagedService -root /u02/nosql/kvroot -class Admin -service
      BootstrapAdmin.5000 -config config.xml
25478 Jps -m
```

Further sanity checks are performed after the configuration steps in the next section are complete.

Oracle NoSQL Database Configuration

After completing the installation steps, Oracle NoSQL Database needs to be configured before it can be accessed by client applications, unless you plan on using only KVLite, in which case the configuration is already complete and you may proceed to Chapter 5 and start developing Oracle NoSQL Database applications.

Prior to understanding the configuration process, it is important to understand *plans* as they are used quite frequently during the initial configuration.


Plans

Plans are a set of commands that perform a series of predefined administrative tasks on the Oracle NoSQL Database cluster. Plans encapsulate multiple operations that may query or modify the state of the key-value store; interact with the key-value store Administration Service, the Storage Nodes, or the Replication Nodes; issue requests that require modifications to store parameters; or simply look up Storage Node configuration parameters. Plans can sometimes be long-running operations and may touch every Storage Node in the cluster, or sometimes run on specific Storage Nodes and complete very quickly.

Plans are created and executed by using the `plan` command from the Administration CLI. The `plan` command takes in a subcommand as the input parameter, which is a prebuilt administrative operation to be performed on the Storage Nodes. All subcommands are preprogrammed to perform documented and specific actions. For example, there are subcommands to create a datacenter and a Storage Node, and to reconfigure the parameters on a Replication Node. Examples of commonly used subcommands are

- **deploy-datacenter** Deploys a datacenter to the key-value store
- **deploy-sn** Deploys a Storage Node to a specific host in a datacenter
- **deploy-admin** Deploys the Administration Service on a Storage Node
- **execute** Executes a previously created plan

When a `plan` subcommand gets executed, the Administration Service stores the subcommand in the Administration Database and assigns an integer, internally referred to as the `plan_id`. You may list all available plans created in the system by using the `plan` command without arguments. For a complete list of all subcommands, you may use `help plan`, as shown in the following example:

```
 kv-> plan  
kv-> help plan
```

Plans are executed using the `plan` command from the Administration CLI. By default, the `plan` command runs asynchronously in the background and the prompt returns immediately. You may optionally use the `-wait` flag to make the plan run synchronously, in which case the command line prompt will only return after the

plan completes. The following example illustrates the use of the `plan` command with the `-wait` flag:

```
kv-> plan deploy-datacenter -name "Dallas" -rf 3 -wait
Executed plan 1, waiting for completion...
Plan 1 ended successfully
```

The `plan wait` command (not the same as the `plan` command with the `-wait` flag) can be used to wait until the specified plan completes, or for a specified time period. The `-last` flag refers to the most recent plan that was created. The complete syntax for the `plan wait` command is

```
kv-> plan wait -id <id> | -last [-seconds <timeout in seconds>]
```

You can also create plans and defer their execution by using the optional `-noexecute` flag. The `-noexecute` flag saves the plan in the system and returns the `plan_id`. The plan can be executed later as required by using the `plan execute -id <id>` command.

The `-wait` and `-noexecute` options, when coupled with the `plan wait` command, provide the capability to program multiple plan executions using scripts. Using these flags, you can run a series of interdependent plans by ensuring that the current plan completes before the next plan is started. Moreover, you can capture the return code of the `plan` command within the script and take appropriate actions. Furthermore, you can also save plans and run them multiple times by retrieving them from a list of stored plans. By using these powerful options available for the `plan` command, the administrators can build a complex set of batch scripts that can be scheduled to run automatically and virtually unattended.

Configuration Steps

Finally, it's time to run the configuration steps. After all configuration steps are completed, a set of Storage Nodes is configured to act and work as one distributed cluster of the key-value store. The key value store is built in accordance with the topology you require, and contains the appropriate number of shards and Replication Nodes that you have identified as part of the capacity planning and sizing activity.

Before proceeding with the configuration steps, ensure that the individual Storage Node Agent (SNA) processes have started on all the Storage Nodes. If you followed the post-install steps from the installation section of this chapter, the SNA processes should already have been started.

The key-value store configuration process comprises the following steps:

1. Start the Administration CLI.
2. Name the key-value store.


3. Create a datacenter.
4. Deploy the first Storage Node.
5. Create an Administration Service.
6. Create a Storage Node Pool.
7. Create the remaining Storage Nodes.
8. Create and deploy Replication Nodes.

Start the Administration CLI

The configuration steps are performed using the Administration CLI. Prior to invoking the CLI, select the Storage Node that would serve as the primary administration node during the configuration process, and also the node that holds the master copy of the Administration Database. The Administration Database stores critical data about the key-value store and its topology, and ensuring its availability is important for the proper functioning of the Oracle NoSQL Database. By default, the first Storage Node you would ever connect to using the Administration CLI becomes the primary administration node (and the only node, until other Administration Services are added).

It is important to note that *all* steps for the configuration of the key-value store should be run on the *same* Storage Node. The Storage Nodes cannot be switched in the middle of the configuration. If that happens for any reason, you will have to start over by manually cleaning up an incomplete configuration.

Log in to the Storage Node you have identified as being the primary administration node. Start the CLI by invoking `runadmin`, as shown in the following example:

```
 > java -jar KVHOME/lib/kvstore.jar runadmin -port 5000 -host node01  
kv->
```

The CLI invocation assumes that the Storage Node `node01` is configured with the Storage Node Agent to listen on port 5000, known as the registry port. Also, `KVHOME` is the directory where Oracle NoSQL Database software is installed.



NOTE

The configuration steps described next can also be coded into a script file and run collectively by passing the file using the `-script` flag of CLI. This is very helpful for avoiding typos and when running the configuration on multiple environments.

Name the Key-Value Store

One of the first attributes you will configure is the name for the key-value store. The name you choose should be suitable to the key-value store function, application, and/or contents. The key-value store name is used to build a directory path on the file system, under which subdirectories will be created to store the actual key-value pairs. Therefore, syntactically speaking, any name would be valid as long as it is allowed by the operating system for naming directories. The valid characters supported for a key-value store name are alphanumeric characters, a minus sign (-), an underscore (_), and a period (.).

At the `kv->` prompt, use the `configure -name` command to name the store. This command takes in the store name as a parameter (the only parameter) and ensures that the name is syntactically valid and allowed by the system. Otherwise, an error is flagged. In the example shown here, a key-value store is given the name `movieDBstore`:

```
kv-> configure -name movieDBstore
```

Create a Datacenter

Conceptually speaking, a *datacenter* is referred to as the facility that houses computer equipment and related infrastructure such as network, storage, and power, usually all components residing in one location. In the context of Oracle NoSQL Database, however, a datacenter is simply a set of Storage Nodes that are part of the key-value store and may be geographically distributed. The current release of Oracle NoSQL Database allows only *one* datacenter per key-value store. In future releases, the concept of multiple datacenters may be introduced to indicate physically separate entities that could be utilized by Oracle NoSQL Database to enhance its high availability and recoverability.

The replication factor for the key-value store can be set only at the datacenter level. Determining the appropriate replication factor is very important as the availability and recoverability of the store depends on it. The command used to create the datacenter is also used to define the replication factor.

Use the `plan deploy-datacenter` command to create the datacenter and define the replication factor. The command takes in a datacenter name via the `-name` input parameter and the replication factor via the `-rf` parameter. In the example that follows, a datacenter named `Dallas` is created with a replication factor of 3. The `plan` command returns the plan number and the status of its execution. The `-wait` flag is used to indicate that the prompt should wait for the command to finish before accepting further input.

```
kv-> plan deploy-datacenter -name "Dallas" -rf 3 -wait
Executed plan 1, waiting for completion...
Plan 1 ended successfully
```

Alternatively, if you do not specify the `-wait` option, you may check the status of the plan command by using `show plans`:

```
kv-> show plans
1 Deploy DC SUCCEEDED
```

Deploy the First Storage Node

You need to add the very first Storage Node to the key-value store. Although you have already connected to a Storage Node and started the SNA service, it has not been added to the key-value store. This step is a prerequisite for creating the Administration Service.

Run the `plan deploy-sn` command to add the Storage Node to the key-value store. This command takes in the datacenter ID as the input, which is obtained by the `show topology` command. The example provided here indicates that `dc1` is the datacenter ID of the Dallas datacenter:

```
kv-> show topology
store=movieDBstore numPartitions=0 sequence=1
dc=[dc1] name=Dallas repfactor=3
```

Now that you have the datacenter ID, run `plan deploy-sn` and add the Storage Node `node01` with the registry port of 5000 to the datacenter ID `dc1`, as shown in the following example:

```
kv-> plan deploy-sn -dc dc1 -host node01 -port 5000 -wait
Executed plan 2, waiting for completion...
Plan 2 ended successfully
```

Create the Administration Service

The Administration Service is in charge of maintaining the Administration Database and providing a Web-based Administration Console. The step after creating the first Storage Node is to create the Administration Service using the `plan deploy-admin` command. This command requires the Storage Node ID (obtained from the topology command, as shown in the next example) and the HTTP port number of the Administration Service. As you may recall, the administration port number is used to route HTTP traffic to the Web-based Administration Console. The following example deploys the Administration Service on Storage Node ID `sn1` with the administration port of 5001.

```
kv-> show topology
store=movieDBstore numPartitions=0 sequence=1
dc=[dc1] name=Dallas repfactor=3
sn=[sn1] dc=dc1 node01:5000 capacity=1 RUNNING

kv-> plan deploy-admin -sn sn1 -port 5001 -wait
Executed plan 3, waiting for completion...
Plan 3 ended successfully
```

**NOTE**

You may run the `show topology` command at each step to show the progress of the configuration process.

After a successful execution of the `plan deploy-admin` command, you would have a single Administration Service deployed in the key-value store. This is sufficient for you to continue with the remaining configuration steps, but later on, you should configure additional nodes to run the Administration Service to ensure that the service is available when failures occur.

Create a Storage Node Pool

A Storage Node Pool is a logical grouping of all the Storage Nodes that are present in the key-value store. Storage Nodes are associated with pools in order to facilitate optimal distribution of resources, especially when the Storage Nodes are added or removed from the key-value store.

Once you have created the Administration Service, create a Storage Node Pool using the `pool create` command. The command requires only the pool name as the input and is run only once at pool creation time. Next, add the Storage Nodes to the pool using the `pool join` command. You would run the `pool join` command on all the Storage Nodes, including the Storage Node you have created earlier. The `pool join` command associates a Storage Node to the pool and requires the pool name and Storage Node ID as the input.

The following example illustrates the CLI commands to be run for this step:

```
kv-> pool create -name movieDBpool
kv-> show topology
store=movieDBpool numPartitions=0 sequence=2
  dc=[dc1] name=Dallas repFactor=3
    sn=[sn1] dc=dc1 node01:5000 capacity=1 RUNNING
kv-> pool join -name movieDBpool -sn sn1
Added Storage Node(s) [sn1] to pool movieDBpool
```

Create the Remaining Storage Nodes

So far, you have created only one Storage Node and joined it to the Storage Node Pool. Although, technically speaking, a single node key-value store is allowed by the system (with a replication factor of one), it does not provide the high availability typically required for production deployments. Therefore, you need to deploy additional Storage Nodes to the key-value store and add them to the Storage Node Pool.

Use the `plan deploy-sn` command to deploy the Storage Node to the cluster and the `pool join` command to add the Storage Node to the Storage Node Pool, as you have done in the earlier step. Remember to repeat these commands on all Storage Nodes that you have identified to be a part of the key-value store.

The following example illustrates the addition of Storage Node 02 and Storage Node 03 to the key-value store.

```
kv-> plan deploy-sn -dc dc1 -host node02 -port 5000 -wait
kv-> show topology
kv-> pool join -name movieDBpool -sn sn2
kv-> plan deploy-sn -dc dc1 -host node03 -port 5000 -wait
kv-> pool join -name movieDBpool -sn sn3
....
```

In the preceding example, observe that `show topology` was run immediately after `deploy-sn` to obtain the Storage Node ID of the node just deployed. The ID is used by the `pool join` command for adding the Storage Node to the pool. But if you notice, the Administration Service allocates Storage Node IDs in a sequential manner. For instance, if the Storage Node that was just created has an ID of 5, then the next Storage Node that you create will be allocated an ID of 6. Therefore, you can always predict the Storage Node ID of the node you have just deployed by incrementing the ID of the previous Storage Node by one—which means you can directly run `deploy-sn` without the need to run `show topology`. Note that this is only true as long as only one CLI session is used to run the `deploy-sn` commands.

NOTE

Now that you have deployed all the Storage Nodes in the cluster, you may now create multiple Administration Services to ensure high availability.

Create and Deploy Replication Nodes

The last step in the configuration process is to create and deploy the Replication Nodes on all the Storage Nodes in the key-value store. Although there is not a direct command to create a Replication Node, you would create and deploy a *topology*, and this in turn would create and deploy the correct set of Replication Nodes on the Storage Nodes.

The `topology create` command is used to create the topology and it requires the topology name, Storage Node Pool, and the total number of partitions as the input. The *topology name* is a unique name that you define to identify the topology; the *Storage Node Pool* is the name of the Storage Node Pool you created earlier; and the *total number of partitions* is obtained by undergoing a capacity planning and sizing exercise (refer to Chapter 8 for further details). The total number of partitions is a static parameter and cannot be altered once it is set, so make sure that the number you provide here is more than the maximum number of shards that you would ever expect your key-value store to grow in its lifetime.

The `topology create` command will automatically create an appropriate number of shards and Replication Nodes based upon the number of Storage Nodes and the replication factor. The number of shards in the key-value store is calculated by dividing the total number of Storage Nodes by the replication factor of the datacenter, and the number of partitions per shard is calculated by dividing the total number of partitions provided by the total number of shards.

Finally, deploy the topology on the Storage Nodes by using the `plan deploy-topology`. The command requires the topology name, and upon its completion, it starts the Replication Node processes on all the Storage Nodes.

The following example illustrates the use of `topology create` and `deploy-topology` commands:

```
kv-> topology create -name movietopo -pool movieDBpool -partitions 300
kv-> plan deploy-topology -name movietopo -wait
Executed plan 6, waiting for completion...
Plan 6 ended successfully
```

The key-value store is fully installed and configured once the preceding commands are successfully completed.

Automating the Configuration Steps

Up to this point, you have run the configuration steps by creating and executing plans using the interactive command line interface of `runadmin`: the `kv->` prompt. In some cases, you would need to automate the configuration steps. Perhaps, you will be building test and development environments repeatedly, or you need to avoid potential typographical errors introduced at the command line, or simply run the configuration unattended during off-hours.


There are two ways to run the Administration CLI commands directly at the OS prompt. The first method uses the `load -file` flag at the time of executing `runadmin` to specify a script file containing a sequence of CLI commands. For example, you may create a script named `moviedeploy.kvs` with the following contents:

```
configure -name movieDBstore
plan deploy-datacenter -name Dallas -rf 3 -wait
plan deploy-sn -dcname Dallas -host node01 -port 5000 -wait
plan deploy-admin -sn sn1 -port 5001 -wait
```

Execute the preceding script by issuing the following command using the `load -file` flag:

```
java -jar kvstore.jar runadmin -host node01 -port 5000 \
load -file moviedeploy.kvs
```

Second, you could run a single CLI command at the OS prompt by specifying the CLI command directly at the prompt, and thereby providing the capability to run multiple CLI commands by running multiple OS commands. The following example of a shell script illustrates the use of this method. Notice that each invocation of `runadmin` would start a separate instance.



```
#!/bin/sh
HOST=node01
PORT=5000
HTTPPORT=5001
KVADMIN="java -jar lib/kvstore.jar runadmin -host $HOST -port $PORT"

# Each of the following CLI command below starts a new instance of runAdmin
$KVADMIN configure -name moviestore
$KVADMIN plan deploy-datacenter -name Dallas -rf 3 -wait
$KVADMIN plan deploy-sn -dcname Dallas -host $HOST -port $PORT -wait
$KVADMIN plan deploy-admin -sn sn1 -port $HTTPPORT -wait
```



NOTE

On UNIX systems, you can also use a [here document](#) as a method of scripting CLI commands at the OS command prompt or within UNIX shell scripts. The [here document](#) provides a mechanism for streaming input strings to a command. Refer to your UNIX scripting guide for further details.

Verifying the Deployment

Several methods are available to test the sanity of the newly built key-value store—mainly, the Web Administration Console, the Administration CLI, the supplied sample programs, and finally the `ping` command provided by the SNA. You can always program your own verification method, but the ones outlined here are quick and readily available, and also quite effective.

Verification Using the Web Administration Console and CLI

The Web Administration Console and the CLI can both be used to validate the key-value store topology and observe the plan execution results. If using the Web Administration Console, point your browser to the machine and port running the Administration Service. For instance, if you used a host named `node01` and listening on port 5001 for HTTP requests, point to the URL `http://node01:5001` to launch the Web Administration Console.

The landing page shown is normally the topology section of the interface, as depicted in Figure 4-1 (if not, just click the topology link). The Topology section

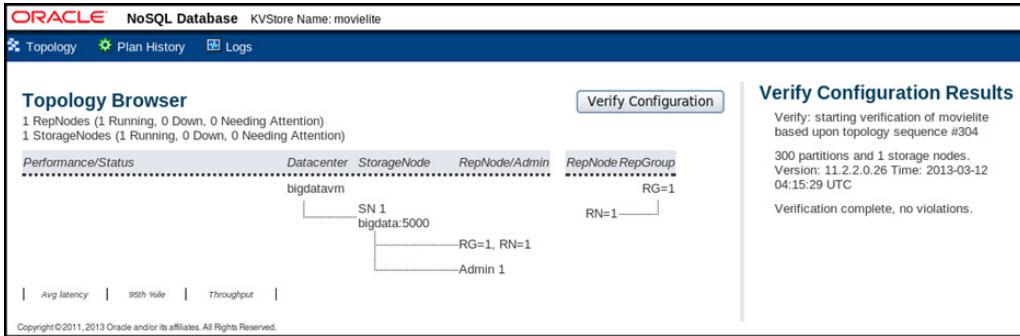


FIGURE 4-1. Topology Browser

displays the details of the key-value store topology and the current status of its components. Ensure that the datacenter, Storage Node, Replication Node, and Administration Node information is correctly displayed and processes have RUNNING status.

Verify Configuration is the function of the web-based console that verifies the topology using an internal topology sequence and alerts you about potential violations. Click the Verify Configuration button and observe the output, as shown in Figure 4-2. The verification step should inform you that the store is currently in RUNNING state and has no violations.

As shown in Figure 4-3, you may click Plan History and observe the list of plans that were run in the process of configuring the store, along with their respective status. Observe that the output is very similar to the `show plans` CLI command.

Alternatively, you may also use the Administration CLI for verifying the configuration. At the CLI prompt, run the `show plans` and `show topology` commands, as shown in the following example. The commands should display the

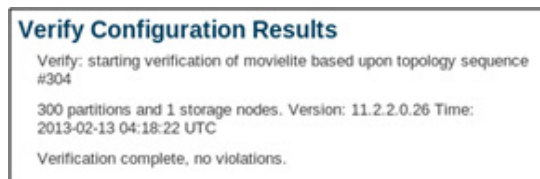
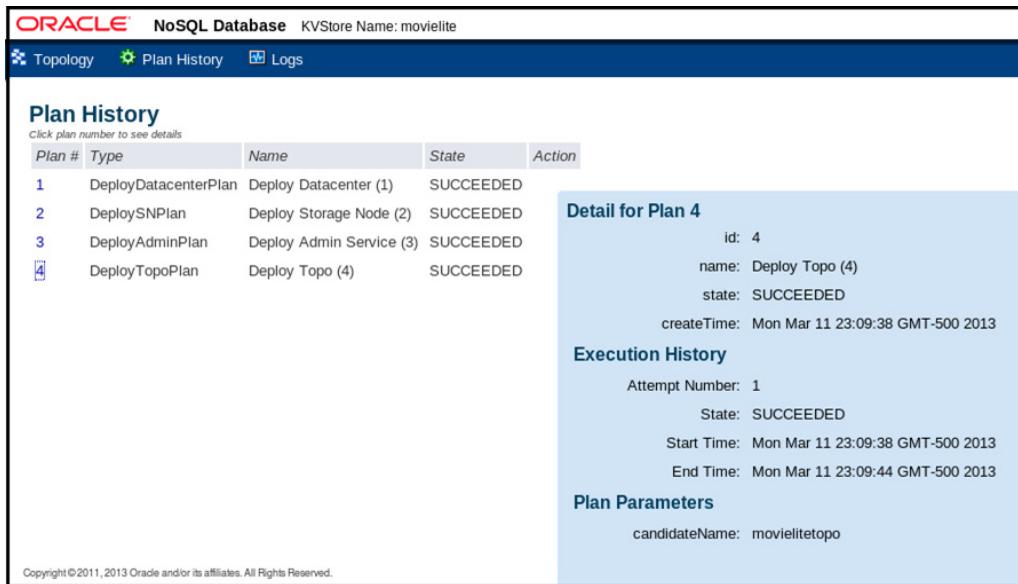


FIGURE 4-2. Verify Configuration

FIGURE 4-3. *Plan History*

status of SUCCEEDED for the plan executions and the status of RUNNING for SNA and Administration Services.

```
kv-> show plans
1 Deploy Datacenter (1) SUCCEEDED
2 Deploy Storage Node (2) SUCCEEDED
3 Deploy Admin Service (3) SUCCEEDED
4 Deploy Topo (4) SUCCEEDED
kv-> show topology
store=movielite numPartitions=300 sequence=304
dc=[dc1] name=bigdatavm repFactor=1
sn=[sn1] dc=dc1 bigdata:5000 capacity=1 RUNNING
[rg1-rn1] RUNNING
```

Verification Using the Sample Program and Ping

The best method to test the installation is perhaps to compile and run the sample Hello World application supplied by the Oracle NoSQL Database software installation. Running the sample application exercises various software libraries of the installation and ensures the connectivity to the key-value store is fully functional.

The sample program outputs the string Hello Big Data World on the screen, if things work as expected.

The sample application is located in the `KVHOME/examples/hello` directory. The following example illustrates the steps for running this application:

```
# cd to KVHOME
$> cd /u01/kvhome/kv-2.0.39

# Compile the sample Hello World Application
$> javac -g -cp lib/kvclient.jar:examples examples/hello/*.java

# Run the Application. Substitute the <hostname>, <port> and <kvstore>
with your settings
$> java -cp lib/kvclient.jar:examples hello.HelloBigDataWorld \
-host <hostname> -port <port> -store <kvstore>
```

You may also run the `ping` command of `kvstore.jar` from the OS prompt. Remember that this command was also run prior to starting the configuration process and, at that time, it output only the version of Oracle NoSQL Database. But now it should output the status of your Oracle NoSQL Database topology, quite similar to the Verify Topology tool of the Web Administration Console. The following example shows the output from a successful execution of the `ping` command:

```
$ java -jar /u02/nosql/kv-2.0.26/lib/kvstore.jar ping -port 5000 -host node01

Pinging components of store movielite based upon topology sequence #304
movielite comprises 300 partitions and 1 Storage Nodes
Storage Node [sn1] on bigdata:5000

Datacenter: bigdatavm [dc1]

Status: RUNNING   Ver: 11gR2.2.0.26 2013-01-28 12:19:21 UTC

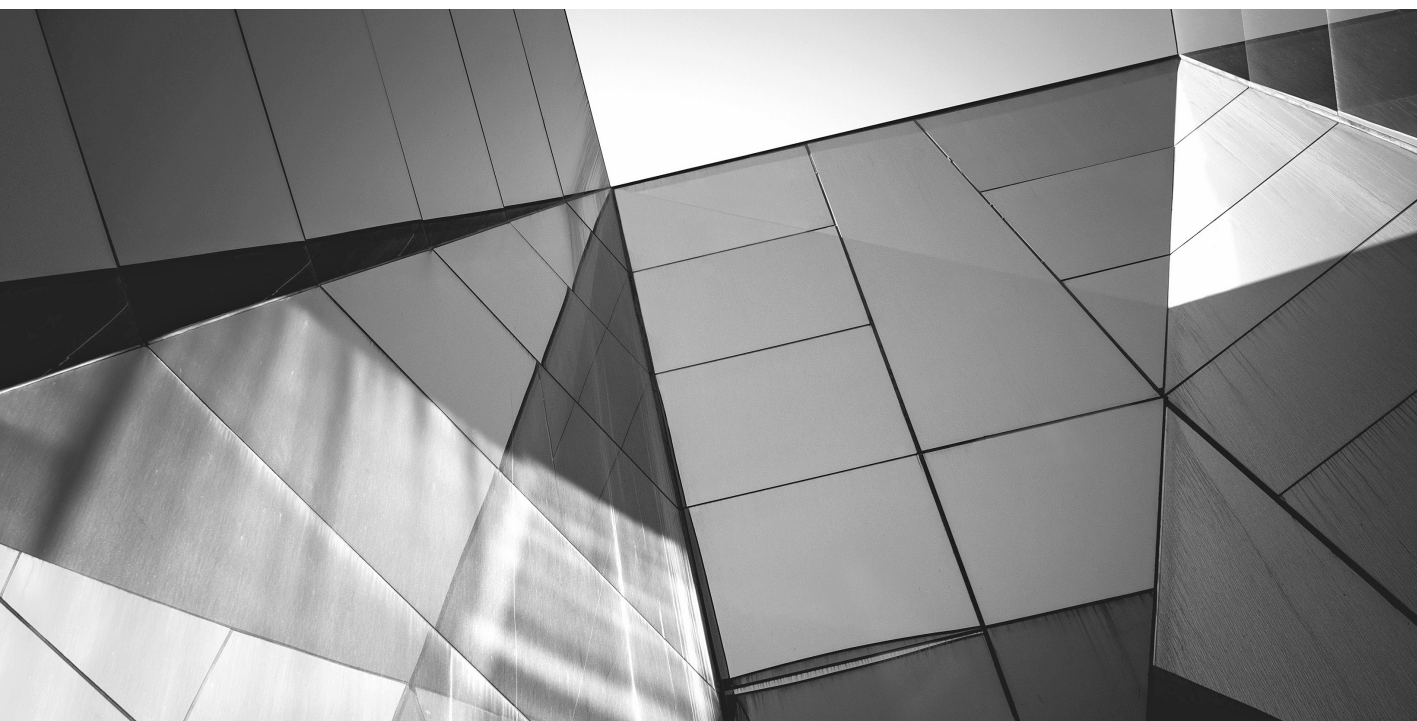
Build id: 99ef986805a3
  Rep Node [rg1-rn1] Status: RUNNING, MASTER at

sequence number: 611 haPort: 5011
```

Summary

In this chapter, you have learned to install and configure a production-grade deployment of Oracle NoSQL Database. Although the actual installation steps are quite simple, to the extent that it's merely a matter of running `unzip` or a `tar` command, the activities that occur both prior to and after the installation process are quite essential, and need to be well planned. This is not only to ensure a successful installation, but also to ensure that the installed software operates at the optimal performance levels and provides the right levels of availability and reliability.

Therefore, the installation of enterprise-grade software should be treated like a mini-project. The project would comprise multiple phases such as planning, implementation, testing, and go-live, not to mention resource allocations that go alongside to manage and execute the project. The instructions outlined in this chapter and in Chapter 8 will help ensure that the Oracle NoSQL Database deployed in the datacenter is fully stable, sized to provide the right capacity, and delivers extreme performance.



CHAPTER 5

Getting Started
with Oracle NoSQL
Database Development

In preceding chapters, we have covered the details surrounding distributed computing concepts, the architecture of Oracle NoSQL Database, and how Oracle NoSQL Database is installed in production environments. In this chapter, we begin to discuss how applications are developed on top of Oracle NoSQL Database. It is important to note up front that application development utilizing a distributed NoSQL database requires the application developer to carefully consider some very important questions, as the answers to these questions will lead directly to how the application will interact with NoSQL Database. More specifically, the following questions should be considered during the application design process:

- What are the latency requirements for the application? Does the application execute according to a service level agreement in which strict latency requirements must be met?
- How tolerant will the application be to inconsistent data? Are there portions of the application that can operate successfully on data that may not be the most recent copy?
- What kind of transactions will be needed in the application? Are there pieces of the application functionality that will need ACID transactions? Can ACID behavior be relaxed for portions of the application in exchange for increased throughput and lower latencies?
- How should the data be modeled in Oracle NoSQL Database such that the expected queries against this data can be satisfied easily and efficiently?

Application developers should think carefully about how to finesse the trade-offs between application throughput, latency, availability, and consistency. These are core concepts in developing successful applications on Oracle NoSQL Database and should be utilized to drive the choice of which API is appropriate for a given task. More specifically, the application designer is strongly encouraged to examine each part of the functional requirements of the application and decide what the trade-offs will be, and then choose the right Oracle NoSQL Database API to deliver on those trade-offs.


Of course, there are other issues to be considered during any application design phase such as threading model, class association, and algorithm design, but these types of questions will not be addressed in the context of this chapter.

Developing on KVLite

It is highly advisable to begin the application development process on top of KVLite, as this implementation of Oracle NoSQL Database provides all of the programmatic API functionality in an extremely simple and easy-to-use package. Once a sufficient

level of comfort has been attained with the Oracle NoSQL Database APIs and modeling the key space, the development process should be moved to a clustered deployment of Oracle NoSQL Database. KVLite is a lightweight version of the NoSQL database server that runs on a single node, has a single replication group, and is packaged inside of the `kvstore.jar` file located in the `lib` folder in the `KVHOME` directory.

KVLite can be launched using the following command:

```
 java -jar KVHOME/lib/kvstore.jar kvlite -root ./kvroot -store <kvstore name> -host <localhost> -port 5000 -admin 5001
```

When you start KVLite successfully, it will either create a new store or open an existing store if it was started previously.

The different parameters to use with the KVLite command line utility are as follows:

- **-admin** If this option is specified, the admin thread that is spawned will listen on the specified port. This will provide a listener for launching the admin command line user interface. This defaults to port 5001 if not specified.
- **-help** Displays a description of the command line parameters.
- **-host** This option specifies the name of the host on which KVLite is running. The DNS registered hostname should be used if the desire is to connect to the KVLite instance from another computer.
- **-logging** Turns on Java application logging. The log files are placed in the examples directory in your Oracle NoSQL Database distribution.
- **-port** Identifies the port on which the KVLite is listening for client connections.
- **-root** Identifies the path of the Oracle NoSQL Database home directory. In the case of KVLite, the database files of the store are located here. The directory has to be present, and if the database files are not present, they will be created.
- **-store** Identifies the name of the store. This option should be used *only* if you are creating a new store.

NOTE

KVLite can be stopped simply by performing a `CTRL-C` in the shell where the KVLite instance was launched.

The APIs for Oracle NoSQL Database can roughly be broken down into the following high-level categories:

- **Writing data** The capability to insert key-value pairs. Several variants exist for the programmer to be able to put a key-value pair into the store. These include
 - **Vanilla put** Store a simple key-value pair, regardless of whether the key exists or not. If the key already exists in the store, the value will be overwritten.
 - **Put if not exists** Only insert the key-value pair if the key does not already exist in the store.
 - **Put if exists** Only insert the key-value pair if the key already exists in the store, effectively overwriting the value associated with the key.
- **Reading data** Reading from Oracle NoSQL Database is further broken down into the following sets of operations:
 - **Read single value** Given a key, the value associated with the key is returned.
 - **Read multiple values** Given a partial key prefix, call key-value pairs that begin with the supplied prefix will be returned. The programmer can further specify that the return occur as a fully materialized set or as a non-materialized iterator. Furthermore, the programmer may specify a range of keys (essentially a “between clause” for those familiar with SQL) to restrict the results being qualified.
- **Deleting data** Given a key, delete the record associated with that key. As with the put methods, several variants of the delete method exist:
 - **Vanilla delete** Delete a simple key-value pair, regardless of whether the key exists or not. If the key exists in the store, the key and the value will be deleted. If the key does not exist in the store, no action will be taken by this method; however, it will return false to the caller.
 - **Delete by version check** Delete only the key-value pair if the version of the key-value pair matches the supplied version.
 - **Multi-delete** Given a parent key, subrange, and depth, this method will delete the descendant key-value pairs associated with the parent key.
- **Mixed operations** The Oracle NoSQL Database API contains a method for the programmer to specify a collection of mixed write and delete operations

along with an associated collection of key-value pairs. This enables ACID transactions across multiple operations, with the caveat that each key in the collection must share the same major path.

A Basic Hello World Program

As an initial exercise, we examine the program `HelloToNoSQLDB`, which is a very simple piece of code that writes a single key-value pair and then reads the value associated with the key. Note that this coding example and the examples that follow in subsequent chapters in this book presume a cursory level of knowledge around the Java programming language. The primary APIs of the Oracle NoSQL Database are written in Java, and any programmer that will interact with these APIs must know the Java programming language. Oracle NoSQL Database also publishes APIs written in the C programming language, but the programming examples in this book focus on Java.

The code for our simple example is as follows:

```
package helloNOSQL;

import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.Key;
import oracle.kv.Value;
import oracle.kv.ValueVersion;
```

We've decided to place our code in a package entitled `helloNOSQL`. Although packages in Java are not mandatory, it's always a good idea to package your code in a namespace that makes it easy for others to find your functionality and potentially reuse your code. Also note that we have imported all of the classes that we will use in this example from the `oracle.kv` package. We have decided to call our class `HelloNOSQLWorld` and make this class public so that anyone can access it. Note the private key-value store instance variable that will act as our main handle to communicate with Oracle NoSQL Database.

```
public class HelloNOSQLWorld {

    private final KVStore store;
```


To run our program and to test it, we need a simple main method, which in our case is simply creating an instance of the `HelloNOSQLWorld` class and then calling the `runExample` method on that instance:

```
public static void main(String args[]) {
    try {
        HelloNOSQLWorld example = new HelloNOSQLWorld (args);
```

32 Getting Started with Oracle NoSQL Database

```
        example.runExample();
    } catch (RuntimeException e) {
        e.printStackTrace();
    }
}
```

The constructor for our class that follows expects three arguments, which are needed to open the store and write to it. We have default values for `storeName`, `hostName`, and `hostPort`. If the user does not provide any command line parameters, the default parameters are used. However, the user has the choice of passing the values for a particular store located on a particular machine, which has been configured with a non-standard port. In the code that follows, we parse through the arguments and then see if one or more arguments are passed through the command line. If it is passed, then it overwrites the defaults.



```
/**
 * Parses command line args and opens the key-value store.
 */
HelloBigDataWorld(String[] argv) {

    String storeName = "kvstore";
    String hostName = "localhost";
    String hostPort = "5000";

    final int nArgs = argv.length;
    int argc = 0;

    while (argc < nArgs) {
        final String thisArg = argv[argc++];

        if (thisArg.equals("-store")) {
            if (argc < nArgs) {
                storeName = argv[argc++];
            } else {
                usage("-store requires an argument");
            }
        } else if (thisArg.equals("-host")) {
            if (argc < nArgs) {
                hostName = argv[argc++];
            } else {
                usage("-host requires an argument");
            }
        } else if (thisArg.equals("-port")) {
            if (argc < nArgs) {
                hostPort = argv[argc++];
            } else {
                usage("-port requires an argument");
            }
        }
    }
}
```

```

    } else {
        usage("Unknown argument: " + this);
    }
}

```

Once the arguments are successfully parsed, a `KVStoreConfig` object is instantiated using the command line parameters passed or the defaults. (Note that we have chosen default values for these parameters which are the same default values that `KVLite` uses.) These parameters are then used to obtain a handle to the key-value store by calling the `getStore` method on the `KVStoreFactory` class. Using this handle, we have access to all of the Oracle NoSQL Database API calls for manipulating data.



```

        store = KVStoreFactory.getStore
            (new KVStoreConfig(storeName, hostName + ":" + hostPort));
    }

```

The function below just prints out the ways in which this `HelloNOSQLWorld` class can be used.

```

private void usage(String message) {
    System.out.println("\n" + message + "\n");
    System.out.println("usage: HelloBigDataWorld ");
    System.out.println("\t-t-store <instance name> (default: kvstore) " +
        "-host <host name> (default: localhost) " +
        "-port <port number> (default: 5000)");
    System.exit(1);
}

```

The function `runExample` that follows will perform the operations of writing a simple record into the NoSQL Database and reading that record back out again. In this example, we insert a key-value pair with the key as the string `Hello` and the value as the string `NOSQL World`. The insert is performed by calling the `put` method on our store object. Note that to do this, we must first create an instance of an `oracle.kv.Key` object and an instance of an `oracle.kv.Value` object. In general, instances of `oracle.kv.Key` objects can be created from Java Strings and instances of `oracle.kv.Value` objects can be created from Java byte arrays. This is important to keep in mind as we begin discussing how to model keys and values in your application later in this chapter.



```

/**
 * Performs example operations and closes the key-value store.
 */
void runExample() {

    final String keyString = "/Hello";
    final String valueString = "NOSQL World!";

```

```

        store.put(Key.fromString(keyString),
Value.createValue(valueString.getBytes()));
    }

```

Now the next step is to obtain the value of the key we have stored to verify that we have successfully inserted the key-value pair into our store. To do this, we try to use the `get` method on the store handle. There are many variations of the `put` and `get` methods to insert and retrieve the key-value pairs into the store, but we use the simplest methods in this chapter as a way to start our quest into the programming world of Oracle NoSQL Database.



```

        final ValueVersion valueVersion = store.get(Key.fromString(keyString));

        System.out.println(keyString + " " +
            new String(valueVersion.getValue().getValue()));

        store.close();
    }
}

```

We complete this example code by closing the handle to Oracle NoSQL Database, cleaning up any resources that are allocated within this handle.

Before moving deeper into the Oracle NoSQL Database APIs and discussing more in-depth programmer topics, it is important to understand how to model the key space for your application.

How to Model Your Key Space

Now that you have seen a simple example of the Oracle NoSQL Database APIs and are familiar with the basic concept of key-value storage, we turn to the topic of key space modeling before moving on to more complex programming examples. As with any database, care must be taken to model the data for your application such that querying your data can be accomplished within the following constraints:

- **Correctness** For applications that must exhibit transactional consistency or isolation, modeling the keys correctly to support this functionality is crucial.
- **Efficiency** Most applications are concerned with retrieving data as fast as possible and using as few system resources as possible.
- **Extensibility** If you're developing a production application, and you're lucky enough to have many users on your application, chances are that you will be asked to enhance your application by adding new functionality in subsequent releases. An extensible data model will afford you the ability to extend your application without entirely redesigning your data model.

- **Normalization** For most transaction processing applications, it is desirable to structure your data model such that queries may access the data by limiting or completely eliminating data duplication.

Keys in the Oracle NoSQL Database are bifurcated into two discrete parts:

- **Major component** The major component of a key denotes the shard that will contain the records for all of the minor keys that follow from a specific major key. This means that records that share the same combination of major key components are guaranteed to be in the same shard, which means they can be efficiently queried. In addition, records with identical major key components can participate in ACID transactions. Keys are distributed across the store by hashing on the key's major component.
- **Minor component** The minor key component of a key can be thought of as the shard local path to the record. Hashing the major component will point you to the shard that contains the data; using the minor component of the key will point you to the record in that shard.

The entire key, major plus minor components, must be unique in Oracle NoSQL Database.

Let's take an example application for the following key space modeling exercise. For this exercise, we choose a simple e-mail application. For this application, we have been given the following simple set of requirements:

1. Users must have the following folders available when they bring up the e-mail application:
 - a. **Inbox** This is where e-mail messages arrive when delivered and must be managed as an ordered container by time.
 - b. **Deleted folder** After a user clicks a delete control on a message, this folder must be populated with the message such that when a user clicks on the deleted folder, the message will be displayed in this folder. This folder must also be managed as an ordered container by time.
 - c. **Sent folder** After a user clicks a send control on a message, this folder must be populated with the message such that when a user clicks on the Sent folder, the message will be displayed in this folder. Again, this folder must also be managed as an ordered container by time.
2. Users must have the ability to search the messages in any folder given a search term and a start and end date.

Given the requirements just stated, we choose to model the keys in the following manner:

```
/users/id/folders/-/inbox/date
/users/id/folders/-/deleted/date
/users/id/folders/-/sent/date
```

As a concrete example, consider a user with ID 34271 and the date 07/13/2013. The preceding three keys would then look like the following strings;

```
/users/34271/folders/-/inbox/20130713
/users/34271/folders/-/deleted/20130713
/users/34271/folders/-/sent/20130713
```

At this point you may be wondering why we chose to structure the keys for this application in this manner. Let's drill down and see what our rationale was for modeling the keys this way:

- **ACID transactions** In Oracle NoSQL Database, ACID transactions are only supported on a shard local basis. Remember that Oracle NoSQL will hash the major portion of the key, and all keys that have the same major path will hash to the same shard. Notice that we have structured the major path such that all of the mail folders for a single user contain the exact same major path, hashing to the same shard, and ultimately able to participate in ACID transactions. Hence, we have just satisfied requirements 1b and 1c simply by structuring our keys appropriately.
- **Extensibility** It will be trivial to add more folders as these requirements materialize. Let's say we get a subsequent requirement to support calendars for all e-mail users. We would simply add another key as `/users/id/folders/-/calendar/20130713`.
- **Ordering in folders** Notice that all of the keys end with a string representation of a day that is formatted as `YYYYMMDD`. We chose this way to model our keys because Oracle NoSQL Database uses the natural sort order of the Java String class, and keys in Oracle NoSQL are simply instances of the Java String class. Hence, if we structure our dates in this manner, we can have Oracle NoSQL order the results of queries for us such that we may satisfy the ordering requirements for 1a, 1b, and 1c.
- **Searching folders** To support requirement 2, we will again make use of the modeling construct we devised by putting a string date format at the end of each key. This can be utilized in API calls to Oracle NoSQL to restrict the results that are returned; furthermore, this restriction can be supplied as a start and end range. To satisfy requirement 2, it's not realistic to build a key

on every word contained in every e-mail. However, being able to restrict the number of e-mails by time and executing this restriction in the NoSQL cluster will be a good option for this application.

In short, having a solid understanding of how to model the keys in your application can go a long way to meeting your application requirements. As you will see in subsequent sections, knowledge of the APIs and how these relate to the data modeling exercise is important to being able to tie all of this together and will help you achieve a successful application implementation.

The Basics of Reading and Writing a Single Key-Value Pair

Now that we have discussed the basics of how to model a key space, the very first task the programmer faces is how to insert data into the Oracle NoSQL Database. Fortunately, the Oracle NoSQL Database API gives you a wide variety of options to tackle this task. There are several ways to write records into the key-value store depending on the complexity of the functional requirements that you are trying to address.

The code fragment that follows explores the different ways of creating a key and storing a value using the Oracle NoSQL Database APIs. In this example, we create a key that will be used to reference notepad data for a specific user; we choose a user with ID 34271. Note that in both examples, the minor portion of the key (the string after the dash) is optional. Also note the initialization of the `KVStore` object, which is created by calling the `getStore` method on the `KVStoreFactory` class. The `KVStoreConfig` is created using the name of the store we wish to connect to as well as one of the host computers in the Oracle NoSQL Database cluster and the port to contact on that host machine. This computer is used only as a bootstrap mechanism to retrieve the topology of the Oracle NoSQL Database cluster. All subsequent API calls will be routed to the appropriate machines in the cluster by the Oracle NoSQL Database driver.



```
String notePadKey = "/users/34271/folders/-/notepad";
String valueString = "A test notepad item that means nothing";
KVStore store = KVStoreFactory.getStore
    (new KVStoreConfig("kvstore", aStoreHost + ":" + port));

Key myKey = Key.fromString(notePadKey);

System.out.println(myKey.getFullPath());

System.out.println(myKey.toString());
```


```
Value myValue = Value.createValue(valueString.getBytes());

store.put(myKey, myValue);
```

In this example, the output of the `System.out.println` would look as follows:

```
[users, 34271, folders, notepad]
/users/34271/folders/-/notepad
```

In a real e-mail application, we would certainly not have the ability to statically declare a path that includes a user ID to a specific folder. This path would be materialized at run time based on the user that is currently logged in, and possibly some notion of that user's authorization to reach the specific folder (in this case, the notepad folder). An alternative to construct the key given these constraints is shown. In the code that follows, we presume that the user ID is returned by an application function that contains an `AuthorizationContext` class for retrieving the currently logged-in user:




```
ArrayList<String> majorList = new ArrayList<String>();
ArrayList<String> minorList = new ArrayList<String>();
int userId = AuthorizationContext.getCurrentUserId();

majorList.add("users");
majorList.add(userId);
majorList.add("folders");

minorList.add("notepad");

Key myKey = Key.createKey(majorList, minorList);
store.put(myKey, myValue);
```

Now let's take a look at how you would use the Oracle NoSQL Database API to read the contents of a user's notepad folder:



```
ValueVersion valueVersion = store.get(myKey);
String notePadContents = new String(valueVersion.getValue().
getValue());
```

Consistency and Durability from the Programmer's Perspective

In this section we introduce the concepts of durability (for writing data) and consistency (for reading data). As you will see, these are key concepts to writing successful applications on the Oracle NoSQL Database.

Durability

If you look up the term “durability” in the Merriam-Webster’s dictionary, you find the following definition:

“Something that is able to exist for a long time without significant deterioration”

Translating this definition to the world of data storage, you can see the parallel to how long data can be stored on any particular media before it deteriorates, or even worse, completely disappears. You can think of durability as the storage of data in the memory of a single computer. The data will be durable until that computer fails or encounters a power outage. Alternatively, you can create a single copy or multiple copies of this data and place it in the memory of another computer or set of computers to gain confidence in the durability of the data; however, should there be a loss of power to all of the computers, the data is lost. You can take this definition several steps further by considering single or multiple copies of the data on disk, or you can increase the confidence in the data’s durability by copying the data to computer memory or disks that reside in separate datacenters that are physically separated.

Oracle NoSQL Database codifies the notion of durability into a policy that can be set by the programmer for each API call that writes data to the store. The stricter the durability policy, the higher your level of confidence that the data can survive a media failure. In Oracle NoSQL Database, there are two distinct but related sets of durability policies:

- **Replica acknowledgment-based policies** Acknowledgment-based policies define how strict the master should behave with respect to how many replicas respond successfully before the master considers the write committed and responds to the caller of the API. There are three flavors of the acknowledgment-based durability:
 - **ALL** This is the most stringent and most durable acknowledgment-based policy and dictates that all replicas must acknowledge successful writes before the master will consider the transaction committed. From the programmer’s perspective, one can think of this as synchronous replication as the caller of the API will wait until all replicas have written the data before the API call will return.
 - **SIMPLE_MAJORITY** This is the next most stringent and is sometimes referred to as *quorum writes*. In this durability policy, the master will asynchronously replicate the data to all replicas and then wait for a successful response only from a majority, or quorum, of replicas before considering the transaction committed. For example, if there is a total of three nodes (a master and two replicas), the master need only wait for a single replica to respond before committing the transaction, as a total of two writes have occurred, making this a majority.

- **NONE** This is the least stringent policy and from the programmer's perspective, this can be thought of as purely asynchronous replication. The master will write the data locally, send the data to the other replicas, and immediately consider the transaction committed without waiting for any responses from the replicas.
- **Synchronization-based policies** Defines the basic guarantee that a write operation has been saved to persistent storage. High levels of synchronization offer a greater guarantee that the transaction is persistent to disk, but trade that guarantee for lower performance. There are three flavors of synchronization policies and these can be specified for master node as well as for non-master node writes:
 - **SYNC** This is the most stringent level of durability and implies the most overhead from a performance perspective. Using this policy will force each node to flush the write to persistent storage before returning success. While this policy gives the programmer a very high level of confidence that a write will never be lost, it comes at a cost of performance.
 - **WRITE_NO_SYNC** This is the next most stringent level of durability and will cause each node to make a system call that will write the data to the file system buffer cache, but not flush the data directly to persistent storage. The data will get flushed to persistent storage by the file system in an asynchronous fashion.
 - **NO_SYNC** This is the least stringent level of durability and will cause each node to write the data to its memory cache. The data will be flushed to persistent storage either on a checkpoint or when the data gets evicted from the node's cache.

When choosing a durability and acknowledgment policy for any particular operation, it's important to think about several issues:

- **The type of data that is being stored** If you're storing high-value operational data that is absolutely critical to the business and you cannot consider any trade-offs with respect to the confidence of durability and the latency of the API call, this should guide your choice of durability and acknowledgment option toward the more stringent durability choices. Our e-mail application discussed earlier is a good example of data that should be stored with a high confidence of durability, as this application does not have extreme SLAs for latency (single- or double-digit milliseconds) and users of our e-mail application expect zero data loss no matter what happens to the underlying systems. On the other hand, if the data is low value and non-critical, you should consider trading off the confidence of durability for lower latency.

- **The type of workflow being implemented** There are some workloads that dictate a very low latency service level agreement (SLA), and for these workloads you should consider a less stringent durability and acknowledgment policy, as this might be the right trade-off in order to achieve the SLAs on latency that are required. For example, in systems that service online display advertising, publishers of online content will require that ad servers return to the browsers in less than 75 milliseconds. This requirement, coupled with the fact that the writes in this workload are tracking consumer browsing behavior, indicates a perfect scenario for choosing a very low durability confidence setting in exchange for very low latency on write operations.

Your application requirements will dictate the strategy of the durability and acknowledgment policies used as a default setting in Oracle NoSQL as well as for specific API calls. The default durability policy can be set for the entire store in the `KVStoreConfig` class, and as you will see later in this chapter, this default configuration can also be overridden in each API call.

Consistency

While *durability* speaks to the resiliency of writing data to Oracle NoSQL Database, *consistency* speaks to the resiliency of reading data from Oracle NoSQL Database. More specifically, consistency refers to the ability for an application to read the most recent copy of data as it has been written to the store. Because a key-value store is typically composed of a cluster of computers (called nodes) that are working together in a distributed fashion, it is possible for a record to be written by the master node in a shard and then subsequently read from another node in the shard. Because there is a time lag between the time that a record is written to the master and the time it takes for the record to be transferred over the network to the other nodes in the shard, the record may not be consistent with the master if read from a node that has not yet received the most recent update from the master. The level of consistency that an application requires between records being read on any node in Oracle NoSQL Database is called the consistency policy. As with the durability and acknowledgment policies in Oracle NoSQL, the consistency policy gives the programmer a powerful tool to be able to trade off performance (lower latency/higher throughput) for stringent consistency. There are four distinct consistency policies in Oracle NoSQL Database:

- **ABSOLUTE** This is the most stringent consistency policy and dictates that the read must be executed at the master node of the shard, thereby guaranteeing that the most recent committed version of the record is returned to the caller of the API. While using this policy gives the programmer a nice guarantee for

the state of the read, it comes at a potential cost in system throughput as well as read latency. Using this policy will prevent the NoSQL Database driver from spreading the read load out across all of the nodes in the shard, thereby possibly overloading the master node and reducing overall system throughput as well as increasing the latency of reads in the system.

- **Time** This is the next most stringent consistency policy and, when supplied by the programmer along with a time ceiling *X* and time unit *Y*, specifies that the read can be performed against any node in the shard as long as that node's version of the record is no longer than *X* time units lag from the version of the record held at the master. The Oracle NoSQL Database driver is always topology-aware and can easily compute a heuristic for how far off any node is from its master in the shard. As an example of how one would use this policy, consider your e-mail application. Let's say you're reading the calendar for the current user. You could supply a time-based read, a ceiling of 500, and a unit of milliseconds, thereby specifying that you would be willing to utilize any node in the shard for this read as long as the record you're reading is no more than 500 milliseconds lagging from the master.
- **Version** This consistency policy is at least as stringent as the time-based consistency policy described previously, but can be utilized in a slightly different way. Version-based consistency allows the programmer to supply a version object to the read call and dictates to the Oracle NoSQL Database driver that it may read from any node that contains at least this version of data and greater. Versions in Oracle NoSQL Database are simply externalized notions of the underlying storage system's log sequence number, and each insert into the store will be tagged with a log sequence number, externalized through the APIs as an instance of a `Version` class. This policy is generally useful for those applications that maintain some state information about previously inserted or updated objects and can use the saved version information for these objects as an optimization hint to the Oracle NoSQL Database driver.
- **NONE_REQUIRED** This consistency policy tells the driver that it can read the record from any node that it thinks is the most optimal node to read from whether or not that node has data that is consistent with its master node. This policy thus places no constraints on the read and is the most optimal policy that can be used when reading records from Oracle NoSQL. This policy is quite useful for those workloads that have very strict latency SLAs and highly favor the ability to return something, even though it may be out of date within say, 10 or 15 milliseconds. Again, we see this type of requirement in the online display advertising world where publishers of online content have placed extremely tight latency restrictions on their ad

serving providers. These providers are reading user behavioral data in an attempt to increase the probability that the user will actually click through an ad that is placed on the publisher's website.

The following code snippet illustrates how to set a default level of durability and consistency as well as how these defaults can be overridden at the individual API call. The first parameter defines the synchronization policy at the master node level, the second parameter defines the synchronization policy at the replication node level, and the third parameter configures the replication acknowledgment policy. Note that in the code snippet that follows, nothing is done with respect to exception handling from the API calls. This is an important topic for the programmer of any NoSQL-based application and will be covered in detail in Chapter 6.

```

Durability defaultDurability = new Durability(
Durability.SyncPolicy.SYNC, // Master sync
Durability.SyncPolicy.NO_SYNC, // Replica sync
Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);
    // Create an instance of the KVStoreConfig class by specifying the name
    // of our store and any machine:port in our cluster of nodes
KVStoreConfig conf = new KVStoreConfig("kvstore", "a_machine:5000");
conf.setDurability(defaultDurability);

conf.setConsistency(Consistency.NONE_REQUIRED);

store = KVStoreFactory.getStore(conf);

```

The code snippet that follows will actually create a key-value pair and insert it into the key-value store based on a new durability policy, which will override the default one.

```

majorList.add("users");
majorList.add(userId);
majorList.add("folders");

minorList.add("notepad");
Key myKey = Key.createKey(majorList, minorList);
String content = "A test notepad value";
Value myValue = Value.createValue(st.getBytes());

// Create durability policy to override the durability policy at the
// configuration object level
Durability durability = new Durability(Durability.SyncPolicy.NO_SYNC,
Durability.SyncPolicy.NO_SYNC, Durability.ReplicaAckPolicy.NONE);
try {
    store.put(myKey, myValue, null, durability, 0, null);
} catch (DurabilityException de) {
    de.printStackTrace();
} catch (RequestTimeoutException re) {
    re.printStackTrace();
}

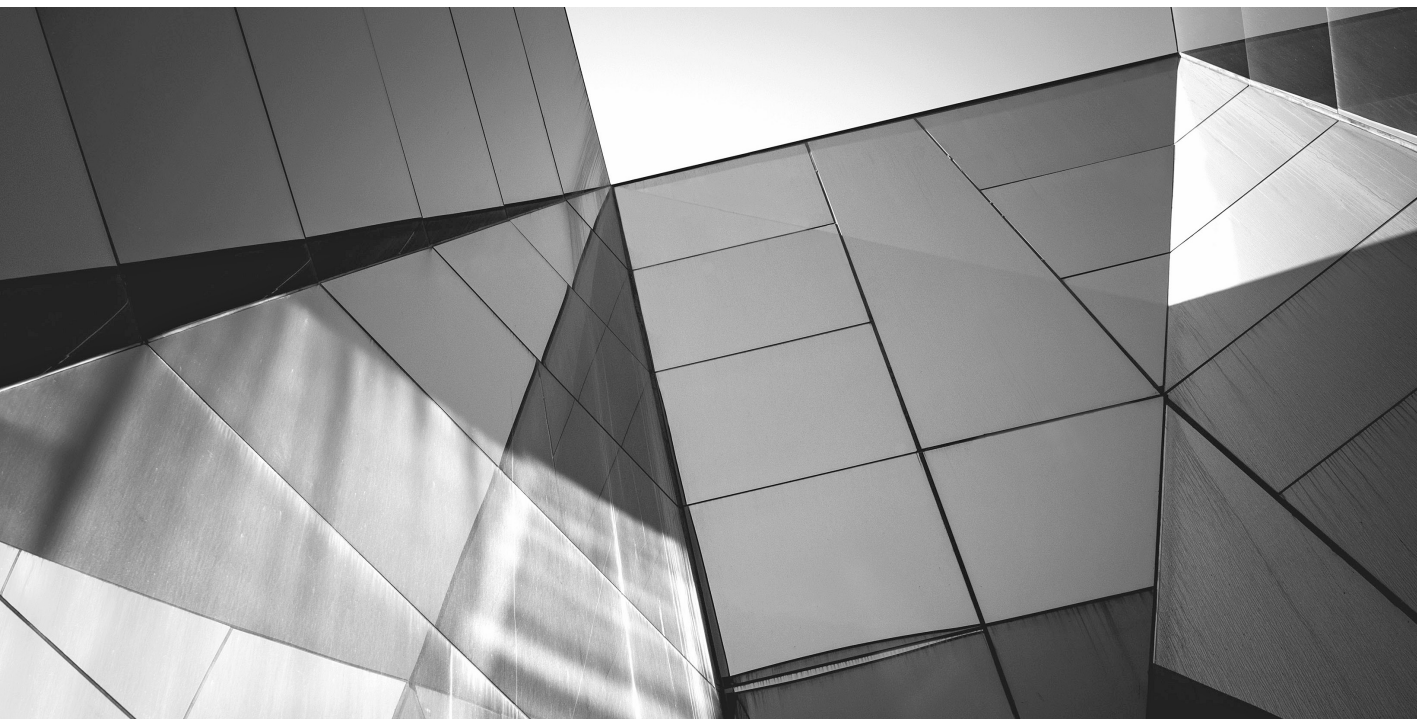
```

44 Getting Started with Oracle NoSQL Database

```
// Override the default consistency policy and specify absolute
// consistency for reading back the record we just wrote
//
ValueVersion vv = store.get(myKey, Consistency.ABSOLUTE, 0, null);
```

Summary

There are many important things to consider when designing an application on top of Oracle NoSQL Database. We have covered issues from the modeling of the key space to the detail settings for durability and consistency, and each area denotes a set of critical design decisions for the programmer. Unlike applications built on top of traditional relational database systems, programmers approaching the design of applications on Oracle NoSQL Database are encouraged to think carefully about how their code will interact with the Oracle NoSQL APIs and how application requirements should drive the API level trade-off decisions that will be crucial to the overall performance and proper functioning of the application.



CHAPTER 9

Advanced Topics

Enterprises have a mix of various technologies deployed in their environment, each serving a specific set of functional needs. Oracle NoSQL Database, when deployed in such a heterogeneous environment, has to work alongside these technologies to provide a complete and seamless integration. Customers deploy software and hardware solutions to solve their business needs. A technology that integrates with ease into existing data management infrastructures, and causes minimal disruptions to the business, leads to an improvement in return on investment (ROI) and lower total cost of ownership (TCO). In this final chapter, we cover multiple use cases that highlight how Oracle NoSQL Database can be integrated with Hadoop, Oracle Database, and Complex Event Processing engines. The chapter also provides details on the new functionality in Oracle NoSQL Database v2 for the support of RDF Graph, Avro format, and the new C-API support.

Hadoop Integration

Big Data deployments typically have two key requirements: The first is to be agile and responsive while receiving large volumes of data in real time; the second is to be able to analyze a large dataset in batch mode. The real-time capability allows the Big Data deployment to support a large number of users, sensors, or inputs, while the batch analysis capability brings the intelligence to make smart decisions for the Big Data problem at hand. These two requirements of a Big Data deployment are best explained through an example of an online retailer.

Let's look at the architecture of a sample online retailer, as shown in Figure 9-1. Customers access the website of the online retailer either through their mobile or desktop devices. These requests are routed to a bank of web servers that load balance these requests to multiple application servers. Fault tolerance and high availability are built into the application tier so that there is no single point of failure. As customers fill their shopping carts, the information needs to be quickly persisted so that it can be available at checkout or on the next visit if the customer does not finish the checkout process. Oracle NoSQL Database, with its high performance clustered architecture, is an ideal database to persist this information. Because of the seasonal nature of the retail business, the amount of customer traffic varies dramatically through the year. It is important to have a persistent store that can grow and shrink based on the load on the retailer's website. The elastic clustered architecture support in Oracle NoSQL Database means that new nodes can be very quickly provisioned, and once the peak load has passed, the number of nodes can be shrunk in an elastic fashion without any downtime. The elastic architecture is also relevant for cloud providers that need to quickly respond to the changing needs of their hosted customers.

Online retailers carry a large variety of products and customers are often lost in the myriads of choices and options available for each product category. It is therefore important to provide personalized recommendation of products to buy to the customers. To achieve this functionality, retailers will periodically send shopping cart information from their NoSQL database to a Hadoop cluster for further analysis.

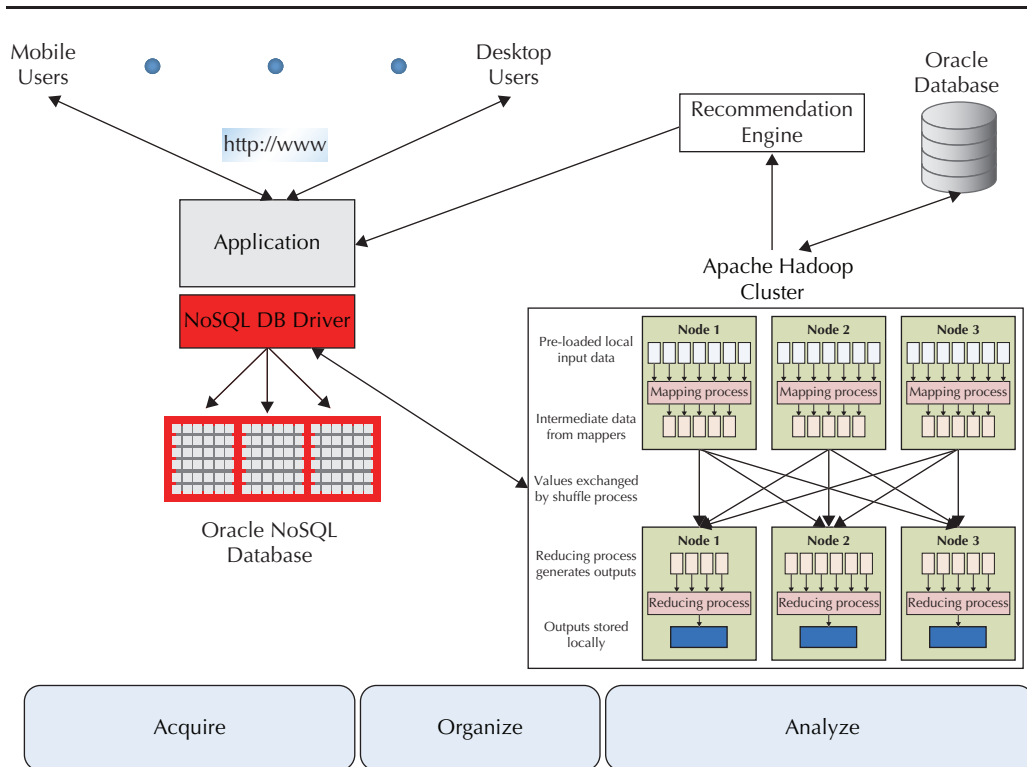


FIGURE 9-1. *Deployment architecture for an online retailer showing the integration between NoSQL and Hadoop*

The analysis provides insight into what products coexist on shopping carts across a large number of customers, and gives the retailer a good idea of what additional products to recommend.

In essence, the Hadoop cluster contributes to the building of a statistical model, which helps ascertain what recommendations to make to the customers; this in turn helps drive additional sales with each customer. A rules engine could help refine these recommendations further before sending them over to the application tier for display on the customer's browser or mobile application.

Oracle NoSQL Database provides an efficient mechanism to integrate with Apache Hadoop systems and has the capacity to move data in a bidirectional fashion. To read data from Oracle NoSQL Database, you use the `oracle.kv.hadoop.KVInputFormat` class and then prepare it for insertion into a Hadoop system. An example is included in the `<kvstore>/examples/hadoop` directory of the Oracle NoSQL Database installation, and it shows how one can read from

Oracle NoSQL Database in a Map/Reduce job and count the number of records for each major key in the store.

```

public class CountMinorKeys extends Configured implements Tool {

    public static class Map
        extends Mapper<Text, Text, Text, IntWritable> {

        private Text word = new Text();
        private final static IntWritable one = new IntWritable(1);

        @Override
        public void map(Text keyArg, Text valueArg, Context context)
            throws IOException, InterruptedException {

            /*
             * keyArg is in the NoSQL Database canonical Key format described in
             * the Key.toString() method's javadoc.
             *
             * The Output is the NoSQL Database record's Major Key as the
             * Map/Reduce key and 1 as the Map/Reduce value.
             */
            Key key = Key.fromString(keyArg.toString());
            /* Convert back to canonical format, but only use the major path. */
            word.set(Key.createKey(key.getMajorPath().toString()));
            context.write(word, one);
        }
    }

    public static class Reduce
        extends IntSumReducer<Text> {
    }

    @Override
    public int run(String[] args)
        throws Exception {

        @SuppressWarnings("deprecation")
        Job job = new Job(getConf());
        job.setJarByClass(CountMinorKeys.class);
        job.setJobName("Count Minor Keys");

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);

        job.setInputFormatClass(KVInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        KVInputFormat.setKVStoreName(args[0]);
        KVInputFormat.setKVHelperHosts(new String[] { args[1] });
        FileOutputFormat.setOutputPath(job, new Path(args[2]));

        boolean success = job.waitForCompletion(true);
        return success ? 0 : 1;
    }
}

```

```

    }

    public static void main(String[] args)
        throws Exception {

        int ret = ToolRunner.run(new CountMinorKeys(), args);
        System.exit(ret);
    }
}

```

In this example, the `map()` function is passed the key and value for each record in the store, and it outputs the major path component as the output key and a value of 1. The `reduce()` step sums the value for each of the records with the same key. More complex map/reduce jobs can be written based on the needs of the application. To move data in the reverse, from Hadoop into Oracle NoSQL Database, you would read data from the Hadoop using the standard mechanisms. The data is then written to Oracle NoSQL Database using the APIs described in this book.

RDF Graph

With the advent of social networks, enterprises are looking at ways to benefit from the social and business relationships that customers are maintaining on social media sites like LinkedIn, Twitter, and Facebook. These relationship graphs and their corresponding arcs when traversed can unlock a treasure trove of information regarding the influence and clout of individual customers. Further, you can combine this information with enterprise content and domain vocabularies to provide a fuller context of your customer base and potential opportunities. This allows for a targeted outreach based on market segmentation and a high degree of propensity to buy. For example, if an enterprise were able to find the top 100 most connected and influential members of its customer base and through them reach out to prospects, it would have greater success with its sales campaign.

To store the information described in the preceding scenario, a database that uses graph structures to store objects as nodes and also captures the relationship between these nodes is required. The RDF (Resource Description Framework) Graph database provides a very flexible and efficient mechanism to store and retrieve associative datasets. RDF has its roots in the semantic web, and it has an abstract syntax that represents a graph-based data model. Oracle NoSQL Database Enterprise Edition supports this semantic technology, the SPARQL query language, and a subset of the Web Ontology Language (OWL), which collectively form the RDF Graph feature of Oracle NoSQL Database.

RDF represents data as triples, with a subject, predicate, and object, and RDF Graph supports named graphs by extending this triple. In the triple shown in Figure 9-2, the lender is the object and “loan products” is the subject. The relationship between the two is the predicate “is seller of.” This basic paradigm of an RDF triple provides for a very flexible and intuitive way to store objects and the relationships between them.

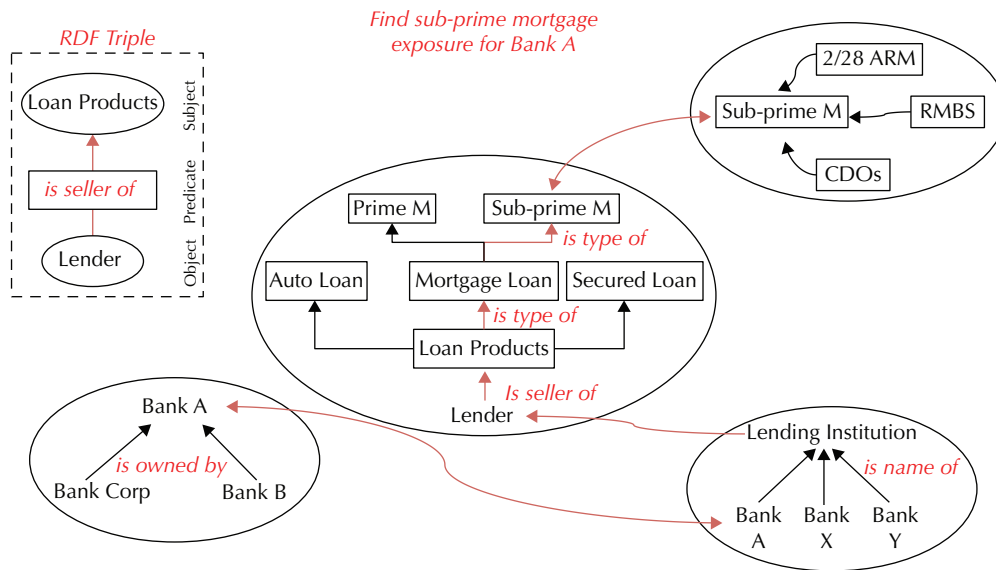


FIGURE 9-2. *RDF Graph concepts*

Further, as shown in the next paragraph, the capability to traverse RDF Graph can provide a very effective mechanism to analyze data.

Figure 9-2 also depicts a larger use case with multiple objects and relationships. This example showcases how RDF Graph can help detect risk liabilities for financial institutions. In this example scenario, a hypothetical Bank A has gone through mergers and, in the process, has acquired multiple other banks and lending businesses. Lenders in turn provide a variety of loans like Auto, Home Mortgage and Secured Loans. If you further look at the classification of the home mortgage loans, you see both Prime and Sub-prime loans. These final class of loans could have exposure to the sub-prime lending crisis that plagued the industry a few years ago. RDF has the capability to store relationships between entities, and it also provides the mechanism to traverse different relationships. This capability of RDF gives us a very elegant mechanism to find the exposure for Bank A to the sub-prime lending risk, something that otherwise could have been lost in the large amount of unstructured and disjointed data sets that exist in any large financial institution.

RDF Graph databases lack primary keys, making the relationship between tables completely arbitrary. The flexible schema evolves easily by adding new relationships and supports querying and discovery by graph patterns and traversal. Within Oracle NoSQL Database, to easily separate RDF-related data, the keys are prefixed

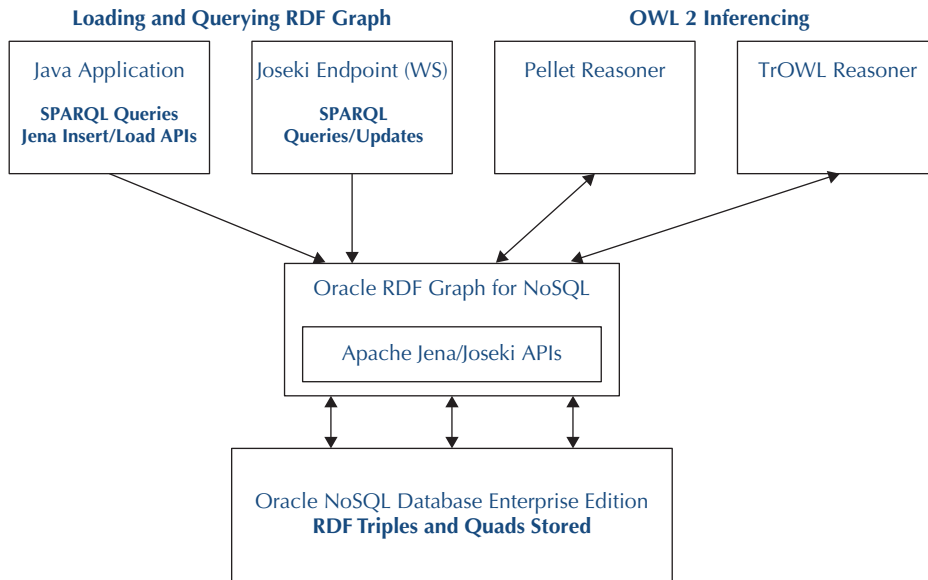


FIGURE 9-3. *The RDF Graph feature of Oracle NoSQL Database*

specifically for RDF Graph data. In addition to these basic features, the RDF Graph implementation in Oracle NoSQL Database features various enhancements to support incremental inserts and for loading large datasets through concurrent bulk loads. The massively parallel scalability of Oracle NoSQL Database makes it possible to process petabytes of triples, and run queries aggregating over the entirety of a large graph. For querying, the RDF Graph feature provides a Java-based interface to store and query semantic data. Also, web services endpoints such as Jena and Joseki SPARQL are supported. With the support of Apache Jena, it is possible to use tools for query, visualization, and ontology engineering. For a detailed diagram of the various RDF tools and technologies supported, refer to Figure 9-3.

Integration with Complex Event Processing

One way of viewing the evolution of business intelligence and analytics is by observing the kinds of BI users that exist today. There are two classes of users: the casual users who consume dashboards and reports generated by BI architects and

developers, which are often based on a small set of predefined queries that are run ahead of time for rapid response; and power users, including business analysts and data analysts, who need analytic sandboxes in order to run novel, ad-hoc queries. The queries of power users are very iterative in nature and may need to be rapidly modified so that they perform the right analysis. Increasingly, power users work on data generated both inside and outside of the enterprise, much of which is a mix of structured metadata and unstructured main data.

With the evolution of Big Data, enterprises are making an investment in real-time analytics, and the speed at which an enterprise can make sense of its database will rapidly become a major competitive advantage. For example, risk analysis algorithms are often run by competing financial institutions seeking real-time information on their investments in order to maintain advantages over competitors. In this case, data must be analyzed rapidly as it streams in; inefficiencies in these analyses, even those that only delay a transaction by a few milliseconds, may result in massive price swings and losses of millions of dollars.

As we have explained in earlier chapters, Oracle NoSQL Database has a strong capability to quickly store large volumes of data and retrieve records of interest with speed. For the high-speed, real-time intelligence scenario, customers have been using NoSQL Database in conjunction with a Complex Event Processing (CEP) engine like Oracle Event Processing (OEP). Complex Event Processing engines provide the capability to analyze streams of data as they come in. They are the first line of defense when it comes to connecting the velocity of Big Data to value. CEPs can quickly detect patterns, and filter and correlate data. For example, in the case of detecting fraud in financial systems, the streaming transactions are passed through a Complex Event Processing engine, and to help detect fraud, the process requires a low latency lookup of recent transactions and user profiles and combining them with the output of CEP algorithms. In this scenario, Oracle NoSQL Database may be used to find recent transactions, authorization requests, account changes, and other indicators, which can then be analyzed in order to inform investigators and prosecutors as to whether the events merit further investigation.

Similar applications can be found in healthcare monitoring, where OEP captures incoming patient monitoring data and looks into Oracle NoSQL Database to find medical data, test results, monitoring trends, monitoring thresholds, or patient profile information—all of which contribute to determining whether an alert or notification should be sent, who should be alerted, and what the most likely treatment, if any, should be. Utilities sensor monitoring (telecommunications, water, power, and so on) is very similar. OEP captures incoming sensor data and looks into Oracle NoSQL Database to find recent sensor history, and alert thresholds, which helps engineers determine whether action should be taken and, if so, decide the course of this action.

Database External Tables

In Chapter 1, we discussed the use case for online advertisement and highlighted the stringent requirement of being able to decide which ad to display in less than 75 milliseconds. There are multiple parties involved in the successful rendering of an advertisement on a mobile device or desktop. To be effective, it is important for the publisher to track the behavior of its consumers, and with the help of the ad server make the right decision of which ad to display. A campaign management system is required to streamline the entire workflow, which includes launching an ad campaign, serving the advertisements, and reporting for billing purposes. This helps merchants, agencies, and marketers see the true value of their advertisement campaigns. The online advertisement scenario requires both Oracle NoSQL Database and Oracle Database because each fulfills critical functionality, and to be effective you also need to integrate the two databases.

In addition to the low latency requirements, online display advertising has to support extremely high data throughput of multimillion requests per second. The platform has to be highly available, and to maximize revenue it must deliver the most relevant ads. Oracle NoSQL Database is used in this scenario to store user cookies and associated behavioral patterns. The behavioral data includes timestamp and frequency. Also, to optimize ad delivery the recentness and frequency of ad display is stored.

A relational database such as Oracle Database 12c can be used to store campaign booking information and real-time business metrics for publishers and advertisers. Oracle Database also works well in order to store longer-term financials such as year-to-date revenues, quarter-over-quarter revenue changes, and the like.

Another use case where the two databases need to work together is a multiplayer online gaming application. Such games have very low latency requirements; player movements must happen in real time, while being tracked on the server. Popular games have tens of millions of active users, and have high availability requirements and heavy workloads. Many of these games also provide the capability for in-game micro-transactions, such as the purchase of power-ups or more advanced weapons with an in-game currency. With millions of players performing such transactions, they represent a major source of revenue for the customer.

Oracle NoSQL Database can be used to track the player movements with low latency, and can be used to store player usage statistics. For games that allow player communication via chat, Oracle NoSQL Database is used as a persistent message store for auditing and COPA compliance. There are various levels of consistency that the database needs to support in the online gaming scenario; for example, loose consistency is fine for some interactions such as player proximity sensing, while ACID transaction support is required for the in-game micro-transactions. Oracle NoSQL Database, with its flexible and configurable consistency model, is ideal for this purpose.

For business financials such as tracking credit card transactions, subscription billing, and payment in the gaming platform, relational databases are used. Oracle Database can be used as the master data store for all player information and payment processing. To better analyze usage trends, the combination of this master data with the micro-transactions stored in Oracle NoSQL Database is required. This could provide critical business information on which geographies and which in-game promotions are performing the best, and which product lines are bringing in the most revenues. The mechanics to do this elegantly is provided in version 2 of the product, through the use of NoSQL and Oracle Database external tables. This functionality provides an easy mechanism to access Oracle NoSQL Database as an external table to Oracle Database. No changes can be made to NoSQL Database content using this interface.

The next section will provide details on the architecture for NoSQL and Oracle external tables, as well as an example walk-through of this feature.

The Oracle Database external table feature allows a user to access data that resides outside of the database as if it were in a table in the database. This flexibility allows you to run SQL queries against the external dataset, and it provides you the mechanism to join data across internal and external tables for data analysis purposes. Oracle has developed functionality to work with Oracle NoSQL Database as an external source. The NoSQL database is read with the aid of a preprocessor utility.

Multiple steps need to be followed to configure the two databases to work together.


The first step is to build a `PREPROCESSOR` for the external table. You then define the external table with one or more Location Files and the name of the `PREPROCESSOR`. Let us assume you name the “publish utility” `nosql_stream`:

1. The next step is to invoke the `PREPROCESSOR` and have it save the configuration details in the Location Files specified. The `PREPROCESSOR` will need the connection information for Oracle Database and Oracle NoSQL Database instances, the name of the external table, details on which NoSQL Database records to process, and the name of any class that needs to be used to convert the key-value pairs from the NoSQL format to the external table format.
2. For the first time, you manually run the publish utility. After that, you only need to run the publish utility again if you want to change the way NoSQL Database is accessed (for example, using a different key prefix, or if you change the port or the rep nodes where you access the database).
3. After the publish utility has been run, you can query the external table in the same way as you would query any other Oracle database table.

Define an External Table

As a first step for the creation of an external table, you will need to specify where the Location Files reside and where the “publish utility” can be found.

```


 sqlplus / as sysdba
SQL> CREATE DIRECTORY ext_tab AS '<exttab_pathname>';
SQL> CREATE DIRECTORY nosql_bin_dir AS '<bin_pathname>';

```

In the preceding SQL statements, `exttab_pathname` is the directory containing the Location File(s), and `bin_pathname` refers to the `exttab/bin/` directory of the NoSQL Database installation where the `nosql_stream` utility is located.

You now need to grant permission to the Oracle user who needs access to the external table. Let's name this user `nosqluser`:


```

 sqlplus / as sysdba
SQL> CREATE USER nosqluser IDENTIFIED BY password;
SQL> GRANT CREATE SESSION TO nosqluser;
SQL> GRANT EXECUTE ON SYS.UTL_FILE TO nosqluser;
SQL> GRANT READ, WRITE ON DIRECTORY ext_tab TO nosqluser;
SQL> GRANT READ, EXECUTE ON DIRECTORY nosql_bin_dir TO nosqluser;
SQL> GRANT CREATE TABLE TO nosqluser;

```

The next step is to define the external table:

```

 SQL> CONNECT nosqluser/password
SQL> CREATE TABLE nosql_data (email VARCHAR2(30),
2                               gender CHAR(1),
3                               address VARCHAR2(40),
4                               phone VARCHAR2(20))
5     ORGANIZATION EXTERNAL
6     (type oracle_loader
7     default directory ext_tab
8     access parameters (records delimited by newline
9     preprocessor nosql_bin_dir:'nosql_stream'
10    fields terminated by '|')
11    LOCATION ('nosql.dat'))
12    PARALLEL;

```

Table created.

SQL>

Let's assume that your NoSQL Database is loaded with the necessary data. For sample datasets, please refer to the Oracle NoSQL Database manual, or the `LoadCookbookData` program in the `<KVHOME>/examples/externaltables` directory.

Edit the Configuration File

Make a copy of the configuration file in `<KVHOME>/examples/externaltables/config.xml` and edit your site-specific values for the `oracle.kv.exttab.connection.url`, `oracle.kv.exttab.connection.user`, `oracle.kv.exttab.connection.wallet_location` (optional), `oracle.kv.kvstore`, and `oracle.kv.hosts` properties based on your Oracle Database and Oracle NoSQL Database installations.

Publish the Configuration

Run the `oracle.kv.exttab.Publish` utility to publish the configuration to the external table Location Files:

```
cd <KVHOME>
java -classpath lib/kvstore.jar:$ORACLE_HOME/jdbc/lib/ojdbc6.jar \
    oracle.kv.exttab.Publish \
    -config <pathname-to-edited-copy-of-config.xml> -publish
```

If you are using Oracle Wallet as an external password store, then you should also include `$ORACLE_HOME/jlib/oraclepki.jar` in your classpath. If the process executes successfully, there will be no output. If you have read access to the Location file(s), you can verify the Publish operation by looking inside one to see if the configuration XML is written there. You will see that two additional properties have been added to the XML: `oracle.kv.exttab.totalExternalTableFiles` and `oracle.kv.exttab.externalTableFileNumber`. Optionally, you can specify the `-verbose` argument to the Publish utility to see more verbose (i.e., debugging) output.

Test the `nosql_stream` Script

Edit the `<KVHOME>/exttab/bin/nosql_stream` script to have the correct values for `PATH`, `KVHOME`, and `CLASSPATH` when the script is run in the execution environment of the Oracle Database server. For this example, `CLASSPATH` should include the `KVHOME/examples` directory (in addition to the `kvstore.jar`).

Test the `nosql_stream` script by running it in a shell:

```
$ <KVHOME>/exttab/bin/nosql_stream <exttab_pathname>/nosql.dat
```

where `<exttab_pathname>` is the path of the Location Files specified earlier in the `CREATE DIRECTORY` command. You should see output similar to the following:

```
user6@example.com|F|#6 Example St, Example Town, AZ|666.666.6666
user1@example.com|M|#1 Example St, Example Town, AZ|111.111.1111
user9@example.com|M|#9 Example St, Example Town, AZ|999.999.9999
user0@example.com|F|#0 Example St, Example Town, AZ|000.000.0000
```

```

user7@example.com|M|#7 Example St, Example Town, AZ|777.777.7777
user8@example.com|F|#8 Example St, Example Town, AZ|888.888.8888
user5@example.com|M|#5 Example St, Example Town, AZ|555.555.5555
user2@example.com|F|#2 Example St, Example Town, AZ|222.222.2222
user4@example.com|F|#4 Example St, Example Town, AZ|444.444.4444
user3@example.com|M|#3 Example St, Example Town, AZ|333.333.3333

```

Use the External Table to Read Data from Oracle NoSQL Database

Using sqlplus (as nosqluser or whatever user you created the external table with), perform a SELECT on the nosql_data external table:

```

SQL> select * from nosql_data;

```

EMAIL	G	ADDRESS	PHONE
user6@example.com	F	#6 Example St, Example Town, AZ	666.666.6666
user1@example.com	M	#1 Example St, Example Town, AZ	111.111.1111
user9@example.com	M	#9 Example St, Example Town, AZ	999.999.9999
...			

```

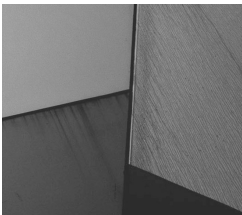
SQL>

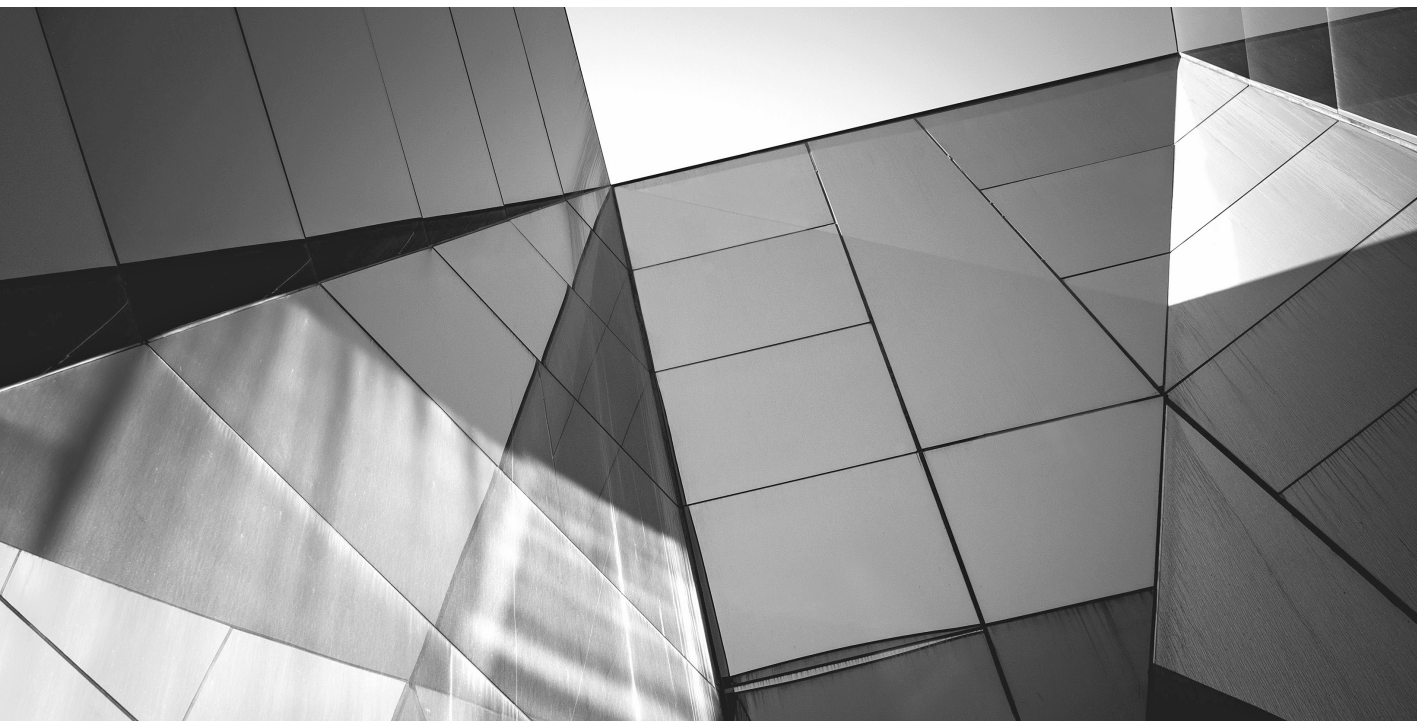
```

To improve the performance of your external table queries, consider using multiple Location Files, as this dictates the degree of parallelism possible when retrieving data.

Summary

Enterprises typically have a variety of technologies deployed in their data center. The success of their business is heavily reliant on the ease with which these heterogeneous technologies work together in a seamless fashion. With emerging technologies like Big Data, it is all the more important to fit into an existing IT environment. As we have seen in this chapter, Oracle NoSQL Database with its tight integration with Oracle Database and a variety of other complementary technologies is an enterprise-grade database offering for this space.





SUMMARY

The advent of new technologies, such as Big Data and Cloud Computing, has led to a drastic shift in the way application developers think about handling and processing data. New applications being built today often require the databases to support schema-less structures, along with the ability to provide high transaction throughput with low latencies. Moreover, deploying an application on the cloud requires the underlying infrastructure to be elastically scalable, and be able to support the agile and unpredictable requirements of cloud-oriented workloads.

Oracle NoSQL Database is a distributed key-value database and supports the requirements of big data by allowing schema-less data to be stored as key-value pairs distributed across a set of storage nodes. It includes a simple yet powerful set of APIs that provide the application developer the ability to perform Create, Read, Update and Delete (CRUD) operations on data. It also provides a flexible transactional model by allowing the application to adjust the transaction durability and consistence guarantees. Avro support is also provided for serialization and de-serialization of key-value records, from both C and Java applications.

Oracle NoSQL Database supports cloud deployments by providing a highly available, fault-tolerant and a horizontally scalable data storage platform, across a set of servers called storage nodes. It offers predictable and bounded transaction latency, and high throughput, by providing the ability to scale on demand and dynamically, by adding additional storage nodes. Data is replicated across the storage nodes to ensure high availability, failover, and read scalability, and to eliminate single points of failure.

Furthermore, Oracle NoSQL Database allows you to seamlessly integrate big data with enterprise applications, Oracle Database and the Hadoop MapReduce framework, via Oracle supplied adapters. The ability to harness and analyze enterprise data together with big data provides significant advantages to the business. This value proposition, along with the full commercial-grade support provided by Oracle, is unique and unmatched, and gives organizations the confidence to deploy Oracle NoSQL Database in production environments.

For an in-depth discussion on more advanced topics, such as the fundamentals of key design for a key-value store, sizing, and configuring Oracle NoSQL Database for an enterprise deployment and advanced programming concepts, refer to *Oracle NoSQL Database: Real-Time Big Data Management for the Enterprise*, ISBN: 0-07-181653-4 (McGraw-Hill Education, 2014).