Oracle® NoSQL Database

Concepts Guide

Release 26.1
E85371-35
April 2026

Oracle NoSQL Database Concepts Guide, Release 26.1

E85371-35

Contents

Preface

This document introduces Oracle NoSQL Database.

This book is aimed at technical users, primarily database administrators and developers who are new to Oracle NoSQL Database.

Conventions Used in This Book

The following typographical conventions are used within this manual:

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

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

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

1

Introduction to Oracle NoSQL Database

Welcome to Oracle NoSQL Database, offering a horizontally scalable distributed storage for key-value pairs, with scalable throughput, and great performance. Oracle NoSQL Database services network requests to store and retrieve data, accessing data as either tables or key-value pairs. Oracle NoSQL Database services data requests with low latency, high throughput, and predictable data consistency, based on how you configure the store.

Oracle NoSQL Database uses Oracle Berkeley DB Java Edition as its underlying storage engine. For more information about Oracle Berkeley DB Java Edition, see Oracle Berkeley DB Java Edition.

Oracle NoSQL Database offers full Create, Read, Update and Delete (CRUD) operations with adjustable durability guarantees. Oracle NoSQL Database is designed with high availability (HA), excellent throughput, and low latency, while requiring minimal administrative interaction.

Oracle NoSQL Database provides performance scalability. To increase performance, you can add hardware. If your performance is sufficient for your needs, you can purchase and manage fewer hardware resources.

Oracle NoSQL Database is designed for applications that require network-accessible data with user-definable read/write performance levels. A typical example is a web application servicing requests across the traditional three-tier architecture: web server, application server, and back-end database. In this configuration, Oracle NoSQL Database should be installed behind the application server, either taking the place of the back-end database, or working alongside it. To make use of Oracle NoSQL Database, you must supply code to run on the application server.

An application makes use of Oracle NoSQL Database by performing network requests against a data store using either Java direct driver or Oracle NoSQL Database language SDKs. The Java direct driver is packaged with the Oracle NoSQL Database software. You can use the Java APIs supported by the Java direct driver to access data from the data store. The Oracle NoSQL Database language SDKs require a proxy server, which translates network activity between the SDK and the Oracle NoSQL Database data store.

- To use Java direct driver, link the Oracle NoSQL Database driver to your application as a Java library (.jar file). Your code can then access any of the Java APIs that the library supplies. See Java Direct Driver Developer's Guide.
- To use a language SDK, link the Oracle NoSQL Database language SDK to your application to access the library and create application requests. Configure the Oracle NoSQL Database Proxy. The Proxy allows the Oracle NoSQL Database language SDK access to the Oracle NoSQL Database data store. To learn more about the Proxy configuration, see Oracle NoSQL Database Proxy.

The language SDKs are compatible with NoSQL on-premises as well as the cloud service. Therefore, it is advisable to write your application to the language SDK APIs for ultra-flexibility. If required, the application can run against the cloud service or on-premises without any application code changes, other than the authentication. For ultra-low latency, it is advisable to consider the Java direct driver, as this driver removes a network hop from each request.

Oracle NoSQL Database language SDKs support Go, Java, .NET, Node.js/TypeScript, Python, and Rust programming languages and Spring framework.

> **Note**
>
> Oracle NoSQL Database supports both direct driver and language SDK for Java.

Table 1-1 API SDK Reference

Language SDK	Documentation	Libraries
Go	GO SDK API Guide	Oracle NoSQL Go SDK
Java	Java SDK API Guide	Oracle NoSQL Java SDK
.NET	.NET SDK API Guide	Oracle NoSQL Dotnet SDK
Node.js/TypeScript	Node.js SDK API Guide	Oracle NoSQL Node SDK
Python	Python SDK API Guide	Oracle NoSQL Python SDK
Rust	Rust SDK API Guide	Oracle NoSQL Rust SDK
Spring	SDK for Spring Data API Guide	Oracle NoSQL Spring SDK

> **Note**
>
> Oracle NoSQL Database requires Java 11 or later version. Because Oracle NoSQL Database is tested using Java 17, it is recommended to use that Java version with Oracle NoSQL Database.

Oracle NoSQL Database also provides SQL for Oracle NoSQL Database, which is an easy to use SQL-like language that supports read-only queries and data definition (DDL) statements. Use this SQL-like language to access table data for read-only queries and DDL statements.

To follow along with the query examples run with the interactive shell and language SDKs, see Developers Guide.

To execute queries using the Java API, see Introduction to SQL for Oracle NoSQL Database.

For a more detailed description of the SQL language (DDL, DML, and queries), see SQL Reference Guide .

NoSQL Database Server Licensing

Oracle NoSQL Database Server is available with two licensing options: Oracle NoSQL Database Community Edition (CE) and Oracle NoSQL Database Enterprise Edition (EE).

For a description on these two licenses, see NoSQL Database Option Differences.

NoSQL Database Client Licensing

Oracle NoSQL Database client APIs are released as open source. Clients ship with source code and are released under the Apache 2.0 License. You use the client APIs to access Oracle NoSQL Database servers using either the Community Edition (CE) or Enterprise Edition (EE) licenses.

Oracle NoSQL Database also supports access to a data store for client applications using Oracle NoSQL Database language SDKs. The SDKs are supported for C, Go, Java, .NET,

Node.js, Python, Rust, and Spring. The SDK drivers are released under the Universal Permissive License (UPL), Version 1.0.

NoSQL Database Option Differences

Oracle NoSQL Database Server is available in two different options: Community Edition (CE), and Enterprise Edition (EE).

Community Edition (CE)

Community Edition is released under the Apache 2.0 License, and ships with source code.

> **Note**
>
> The Community Edition is not always at the same release number as EE. For example, Enterprise Edition can be shipping few versions ahead of Community Edition.

Enterprise Edition (EE)

Enterprise Edition requires a commercial license. It does not ship with source code.

Feature Differences between EE and CE

Oracle NoSQL Database Server Enterprise Edition includes new and updated features with each release. However, Community Edition is neither released as frequently as the Enterprise Edition, nor can it support every EE feature. Currently, CE does not support these features:

- Oracle GeoJSON Data support
- Kerberos Authentication Service integration
- Oracle Database External Table integration
- Oracle Event Processing integration - Streams Processor Engine
- Oracle Wallet integration for external password storage
- Multi Region Tables
- Secure Elastic Search

Architecture

Oracle NoSQL Database applications read and write data by performing network requests against an Oracle NoSQL Database data store. The data store is a collection of Storage Nodes, each of which hosts one or more Replication Nodes. Data is automatically spread across these Replication Nodes by internal data store mechanisms. Given a traditional three-tier web architecture, the data store either takes the place of your back-end database, or runs alongside it.

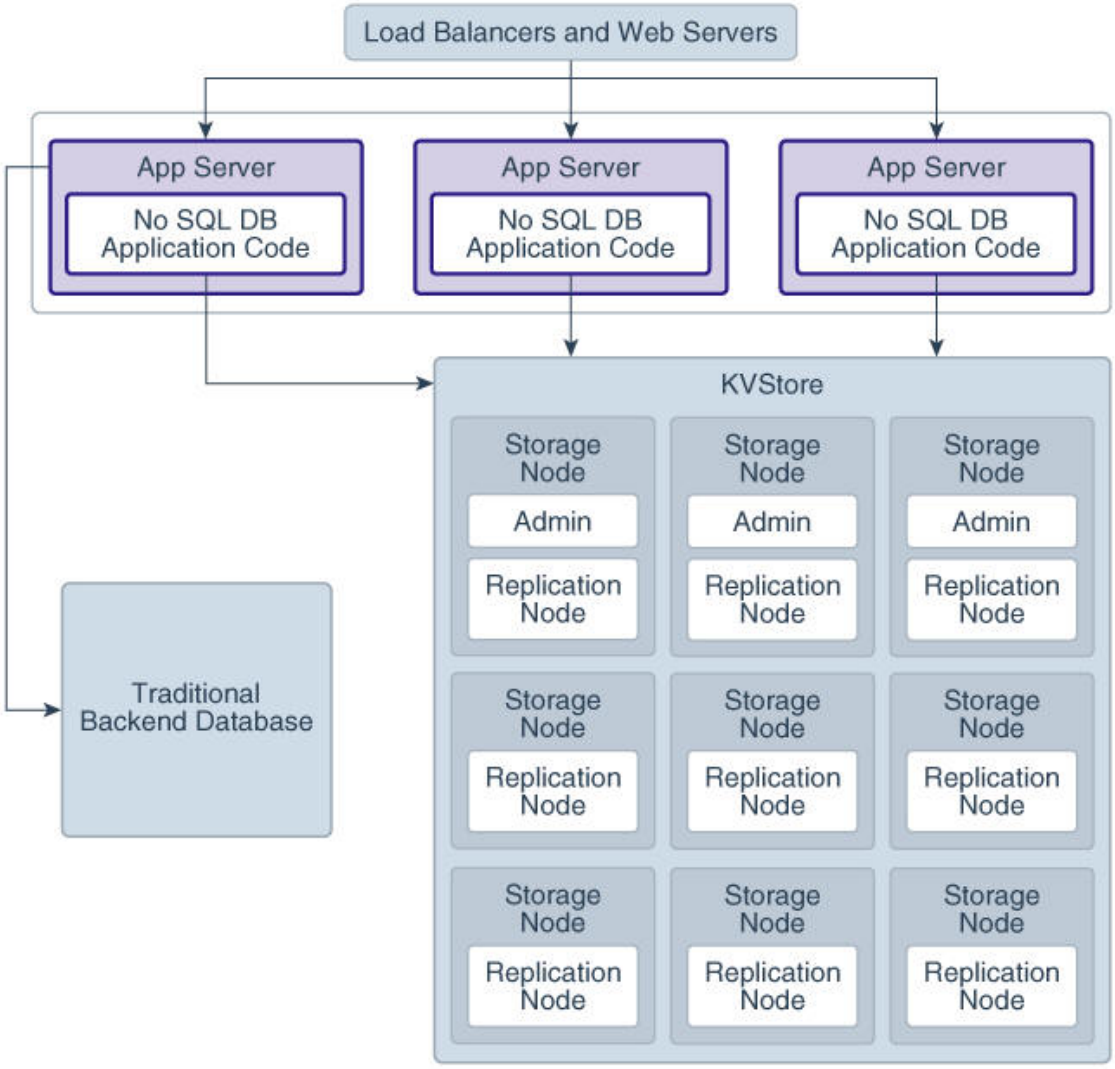
Optionally, a data store installation can be spread across multiple physical locations, each of which is called a *zone*. Zones are described in Zones.

> **Note**
>
> Replication Nodes are implemented using Berkeley DB, Java Edition (JE). JE is an enterprise-class, transaction-protected database, which is fully described in the Oracle Berkeley DB Java Edition.

The store contains multiple Storage Nodes. A *Storage Node* is a physical (or virtual) machine with its own local storage. The machine is intended to be commodity hardware. While not a requirement, each storage node is typically identical to all other Storage Nodes within the store.

The following illustration depicts a typical architecture used by an application that uses an Oracle NoSQL Database. Specifically, three of nine Storage Nodes each host an Admin process and a Replication Node. The remaining Storage Nodes each host a Replication node.

Figure 1-1 Typical Architecture for Oracle NoSQL Database Store



Every Storage Node hosts one or more Replication Nodes as determined by its *capacity*. A Storage Node's capacity serves as a rough measure of the hardware resources associated with it. Stores can contain Storage Nodes with different capacities, and Oracle NoSQL Database ensures that a Storage Node is assigned a proportional load size to its capacity.

A Replication Node, in turn, contains a subset of the store's data. Storage node data is automatically divided evenly into logical collections called *partitions*. Every Replication Node contains at least one, and typically many, partitions. Partitions are described in greater detail in Partitions.

Finally, each Storage Node contains monitoring software that captures information ensuring the Replication Nodes that it hosts are running and healthy.

For more information on how to associate capacity with a Storage Node and know the best way to balance the number of Storage Nodes and Replication Nodes, see Determining Your Store's Configuration in the *Administrator's Guide*.

Replication Nodes and Shards

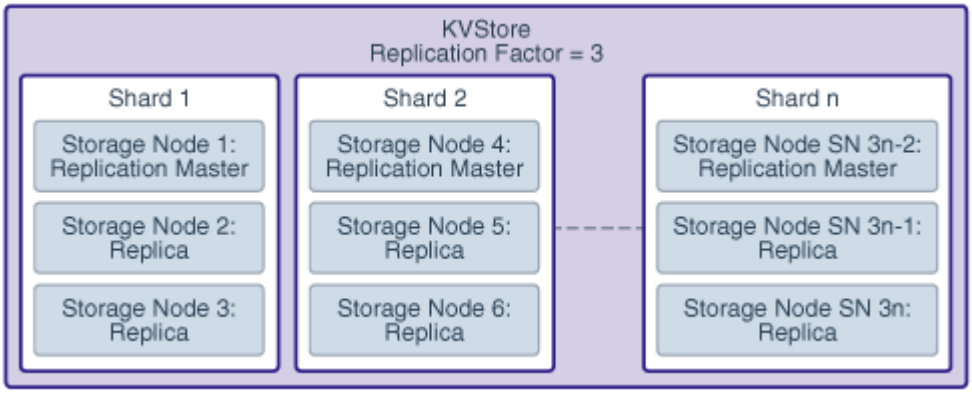
At a high level, you can think of a *Replication Node* as a single database containing tables or key-value pairs. Storage Nodes host one or more Replication Nodes. Because hosting a Replication Node depends on a healthy amount of resources, generally, Storage Nodes host only a single Replication Node. However, for installations with hardware that has abundant resources (memory, CPUs, and disks), Storage Nodes can, and do, host multiple Replication Nodes.

Your store's Replication Nodes are organized into *shards*. A single shard contains multiple Replication Nodes. Each shard has a *master node*. The master node performs all database write activities. Each shard also contains one or more read-only *replicas*. The master node copies all new write activity data to the replicas. The replicas are then used to service read-only operations.

While there can be only one master node per shard at any given time, any of the other shard members can become a master node. An exception to this is for nodes in a secondary zone as described below.

The following illustration shows how the data store is divided up into shards:

Figure 1-2 Data store Shards



If the machine hosting the master node fails in any way, the master automatically fails over to one of the other nodes in the shard. One of the replica nodes is promoted automatically to master.

Production data stores should contain multiple shards. At installation time you provide information that allows Oracle NoSQL Database to automatically decide how many shards the store should contain. The more shards that your store contains, the better your write performance is because the store contains more nodes that are responsible for servicing write requests.

Replication Factor

The number of nodes belonging to a shard is called its *Replication Factor*. The larger a shard's Replication Factor, the faster its read throughput, because there are more machines servicing the read requests. However, a large replication factor reduces write performance, because there are more machines to which writes must be copied.

A store can be installed across multiple physical locations called zones. You set a Replication Factor on a per-zone basis. Once you set the Replication Factor for each zone in the store, Oracle NoSQL Database makes sure the appropriate number of Replication Nodes are created for each shard residing in every zone in your store. Here are the terms used to describe these aspects of Oracle NoSQL Database:

- *Replication Factor*: The number of nodes belonging to a shard.
- *Zone Replication Factor*: The number of copies, or replicas, maintained in a zone.
- *Primary Replication Factor*: The total number of replicas in all primary zones.
- *Secondary Replication factor*: The total number in replicas in all secondary zones
- *Store Replication Factor* The total number of replicas in all zones across the entire store.

For additional information on how to identify the *Primary Replication Factor* and the implications of its value, as well on multiple zones and replication factors, see Replication Factor in the *Administrator's Guide*.

Partitions

All data in the store is accessed by one or more keys. A key might be a column in a table, or it might be the key portion of a key/value pair.

Keys are placed in logical containers called *partitions*, and each shard contains one or more partitions. Once a key is placed in a partition, it cannot be moved to a different partition. Oracle NoSQL Database distributes records evenly across all available partitions by hashing each record's key.

As part of your planning activities, you must decide how many partitions your store should have. You cannot configure the number of partitions after the store has been installed. For information about how to plan your store, see Initial Capacity Planning in the *Administrator's Guide*.

You can expand and change the number of Storage Nodes in use by the store. The store is then reconfigured to take advantage of the new resources by adding new shards. When this happens, existing data is spread across new and old shards by redistributing partitions from one shard to another. For this reason, it is desirable to have a large number of partitions to support fine-grained reconfiguration of your store.

As a general guideline, each shard should have at least 10 to 20 partitions. The number of partitions should be evenly divisible by the number of shards. Since the number of partitions cannot be changed after the initial deployment, plan the number of partitions for the maximum size of your store in the future. For example, while there is overhead in configuring a large number of shards, it is reasonable to specify a partition number that is 100 times the maximum number of shards you expect your store to contain.

Zones

A zone is a physical location that supports high capacity network connectivity between the Storage Nodes deployed within it. Each zone has some level of physical separation from other zones. Typically, each zone includes redundant or backup power supplies, redundant data communications connections, environmental controls (for example: air conditioning, fire suppression), and security devices. A zone can represent a physical data center building, the floor of a building, a room, pod, or rack, depending on the particular deployment.

Oracle recommends installing and configuring your store across multiple zones. Having multiple zones provides fault isolation, and increases data availability in the event of a single zone failure. Multiple zones help mitigate systemic failures that affect an entire physical location, such as a large scale power or network outage.

There are two types of zones — *primary* and *secondary*. Primary zones are the default. They contain nodes that can serve as masters or replicas. Secondary zones contain nodes that can serve only as replicas. You can use secondary zones to make a copy of the data available at a distant location, or to maintain an extra copy of the data to increase redundancy or read capacity.

Only primary zones can have a Replication Factor equal to zero. Zero capacity Storage Nodes are used for Arbiter Nodes, which only primary zones can host.

You can use the command line interface to create and deploy one or more zones. Each zone hosts the deployed storage nodes. For additional information on zones and how to create them, see Create a Zone in the *Administrator's Guide*.

Arbiter Nodes

An `Arbiter Node` is a lightweight process that is capable of supporting write availability in two situations. First, when the primary replication factor is two and a single Replication Node becomes unavailable. Second, when two Replication Nodes are unable to communicate to determine which one of them is the master. The role of an Arbiter Node is to participate in elections and respond to acknowledge requests in these situations.

An Arbiter Node does not host any data. You create a Storage Nodes with zero storage capacity to host an Arbiter Node. While you can allocate Arbiter Nodes on Storage Nodes with a capacity greater than zero, those Arbiter Nodes have a lower priority during allocation than those on zero capacity Storage Nodes.

The Arbiter Node is allocated on a Storage Node outside of the shard. An error occurs if there are not enough Storage Nodes to host an Arbiter Node located on a different Storage Node from other shard members. The Arbiter Node provides write availability in the absence of a single Storage Node. The pool of Storage Nodes in a primary zone configured to host Arbiter Nodes is used for allocating an Arbiter Node.

For more information on Arbiter Nodes, see Deploying an Arbiter Node Enabled Topology in the *Administrator's Guide*.

Topologies

A *topology* is the collection of zones, storage nodes, shards, replication nodes, and administrative services that make up your NoSQL Database store. A deployed store has one topology that describes its state at a given time.

After initial deployment, the topology is laid out so as to minimize the possibility of a single point of failure for any given shard. This means that while a Storage Node might host more than one Replication Node, those Replication Nodes are never from the same shard. This improves the chances that the shard will have continuous availability for reads and writes, even if a hardware failure takes down the host machine.

Arbiter Nodes are automatically configured in a topology if the primary replication factor is two and a zone is configured to host `Arbiter Nodes`.

Topologies can be changed to achieve different performance characteristics, or in reaction to changes in the number or characteristics of the Storage Nodes. Changing and deploying a topology is an iterative process. For information on how to use the command line interface to create, transform, view, validate and preview a topology, see Steps for Changing the Store's Topology in the *Administrator's Guide*.

Using Master Affinity Zones

Master Affinity zones let you specify which Primary Zone handles write requests for your client applications.

Oracle NoSQL Databases use zones. Zones duplicate the entire data store, spreading the data store and load across multiple physical locations. Having zones helps to avoid catastrophic data loss and operational disruptions. A zone consists of a number of Storage Nodes (SNs) and Replication Nodes (RNs). See Architecture in the *Concepts Guide*.

Two kinds of zones exist:

- Primary zones — can host both master nodes and replication nodes, though they are not required to do so. Data read and write requests go to Primary zones configured to handle such requests.
- Secondary zones — have no master node. They handle only read requests from client applications.

Each shard has a single Master Node, which is capable of writing data to all RNs. Regardless of zone type, all zones require high quality network connectivity to maintain optimal performance for writing data to the RNs, and accessing data from RNs for application data requests.

You choose which Primary zones have Master Affinity, which provides a way for you to send write requests to a specific Primary zone. Setting the `-master-affinity` property confirms its designation as such, while keeping the default `–no-master-affinity` property designates that a zone is not a Master Affinity zone. Using the `–master-affinity` property organizes Master nodes from different shards into the Master Affinity zone, providing several advantages:

- Master Affinity zones service high demand write requests across shards.
- When a Master Node fails, a replacement from the Master Affinity zone is available to take over from the failed node, with virtually no lag in service.
- RNs in the Master Affinity zone perform a standard election process to determine the Master Node that assumes the role of the failed Master Node.

Using Master Affinity zones successfully requires knowledge of the zones that are in closest proximity to your client applications with the highest demands. The client application is then predictably serviced by both the Master Node and RNs in the Master Affinity zone.

Benefits of Master Affinity Zones

Master affinity is a zone property. A zone either has the Master Affinity property (`-master-affinity`), or does not (`-no-master-affinity`). Most likely, you will choose a specific Primary Zone to become a Master Affinity zone because that zone is ideally suited to service demanding client write requests. The candidate zone is in close proximity to the application demands, and has high quality communication capabilities to service them.

You can set the Master Affinity property only on Primary Zones. Once you do, only nodes in Master Affinity zones can become masters during a failover. Having a Master Affinity zone with one or more Master nodes supports both low latency write activities and high availability.

Typically, when a Master Node fails, the Replication Nodes (RNs) enter a selection process to elect a new Master node. The election involves an algorithmic approach using, among other factors, a criteria to elect the RN with the most recent data. Once a zone is a Master Affinity zone, and a Master Node fails, a similar process occurs. When a new Master node exists, write requests are automatically directed to the new Master, and absolute consistency requests are serviced by the new Master in the Master Affinity zone.

All storage nodes (SNs) can determine if they are part of a Master Affinity zone. If they are not part of a Master Affinity zone, they help determine which SNs are candidates to host RNs that will transfer to the Master Affinity zone as potential Master Nodes during election. By choosing and assigning RNs to a Master Affinity zone, if the current Master node fails, the next applicable node will assume its responsibilities.

Adding a Master Affinity Zone

Describes the Master Affinity zone parameter, and the effects of setting it.

Using Master Affinity zones is optional. By default, after upgrading to the current release, all zones are set to `-no-master-affinity`. To use Master Affinity, you change the zone property manually. The Master Affinity zone property affects only the Replication Node masters, and has no effect on the database Admin masters. This section describes how to use Master Affinity zones, and what effects they can have on your operations.

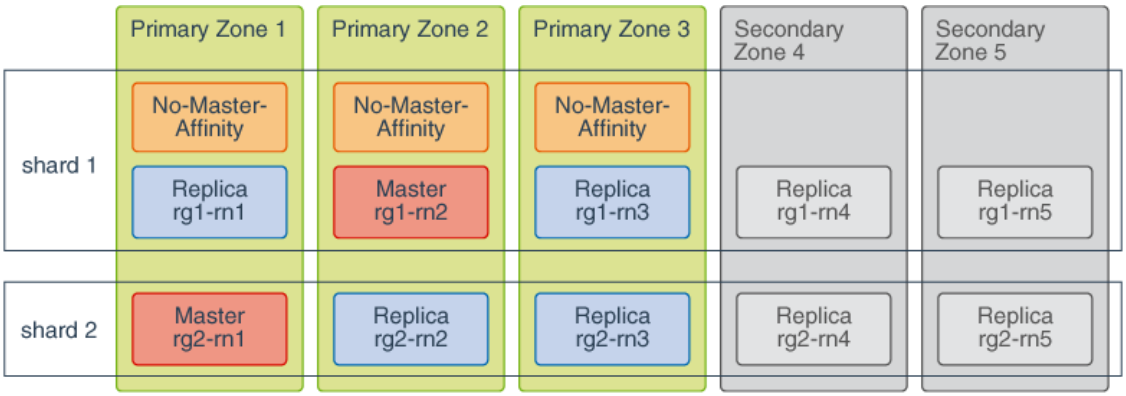
Your first choice is to determine which zones should have Master Affinity. The chosen zones must be in close physical proximity to the applications they serve. In this way, a Master Affinity zone provides the lowest latency write performance.

As an example, the following topology is for two (2) shards (`rg1` and `rg2`) with a replication factor of three (3), described as a 2 * 3 data store, where `rg2-rn1` and `rg1-rn2` are the master nodes in `zn1` and `zn2`, respectively:

```
Storage Node [sn1] on localhost:5100        Zone: [name=1 id=zn1
type=PRIMARY allowArbiters=false            Status: RUNNING
    Admin [admin1]             Status:           RUNNING,MASTER
    Rep Node [rg1-rn1]         Status:           RUNNING,REPLICA
    Rep Node [rg2-rn1]         Status:           RUNNING,MASTER
Storage Node [sn2] on localhost:5200        Zone: [name=2 id=zn2
type=PRIMARY allowArbiters=false             Status: RUNNING
    Admin [admin2]             Status:           RUNNING,REPLICA
    Rep Node [rg1-rn2]         Status:           RUNNING,MASTER
    Rep Node [rg2-rn2]         Status:           RUNNING,REPLICA
Storage Node [sn3] on localhost:5300        Zone: [name=3 id=zn3
type=PRIMARY allowArbiters=false             Status: RUNNING
    Admin [admin3]             Status:           RUNNING,REPLICA
    Rep Node [rg1-rn3]         Status:           RUNNING,REPLICA
    Rep Node [rg2-rn3]         Status:           RUNNING,REPLICA
```

Here are the zones before using Master Affinity. Primary Zones 1 and 2 each have a master node in their respective shards (`rg1` and `rg2`):

Figure 1-3 Zone Distribution Before Master Affinity



After choosing the Primary Zone best suited for having Master Affinity, set the `–master-affinity` property as follows:

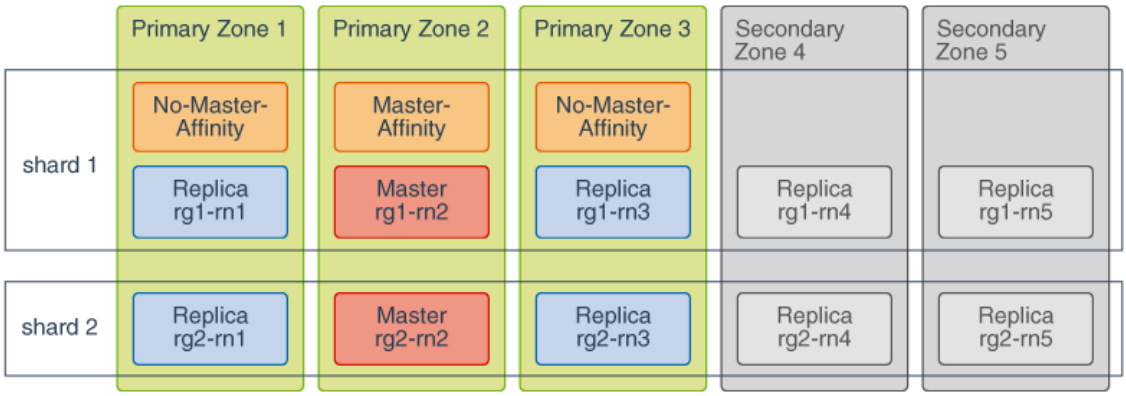
- When deploying a zone for the first time, use the `plan deploy-zone` command.
- After deploying a zone, use the `topology change-zone-master-affinity` command.

For example, here is the `plan deploy-zone` command being used as part of configuring the store `mystore` to change the `master-affinity` zone property. In this example, you set the `master-affinity` property for Zone 2.

```
configure -name mystore
plan deploy-zone -name 1 -rf 1 -no-master-affinity -wait
plan deploy-zone -name 2 -rf 1 -master-affinity -wait
plan deploy-zone -name 3 -rf 1 -wait
```

> **Note**
>
> When Master Affinity is in effect for Zone 2, both master nodes for the two shards are placed in Zone 2.

Figure 1-4 Zone Distribution After Master Affinity



Losing a Master Affinity Zone Node

Describes what occurs when a Master Node fails in a Master Affinity Zone.

After your initial setup, you determine which Primary zone will be a Master Affinity zone. Using Master Affinity zones optimizes write requests to Master Nodes in that zone. The Storage Nodes (SNs) can detect if they are part of a Master Affinity zone. If an SN is not part of a zone itself, it detects which SNs are part of a Master Affinity zone.

If a Master Affinity zone master node fails, the RNs detect if an applicable node exists within the zone. For example, the Master Affinity zone may have another master node. If another master node is not available, RNs elect the best candidate, or have applicable RNs from other zones migrate into the Master Affinity zone for Master Node consideration. Such zone realignment occurs automatically to support the Master Affinity zone.

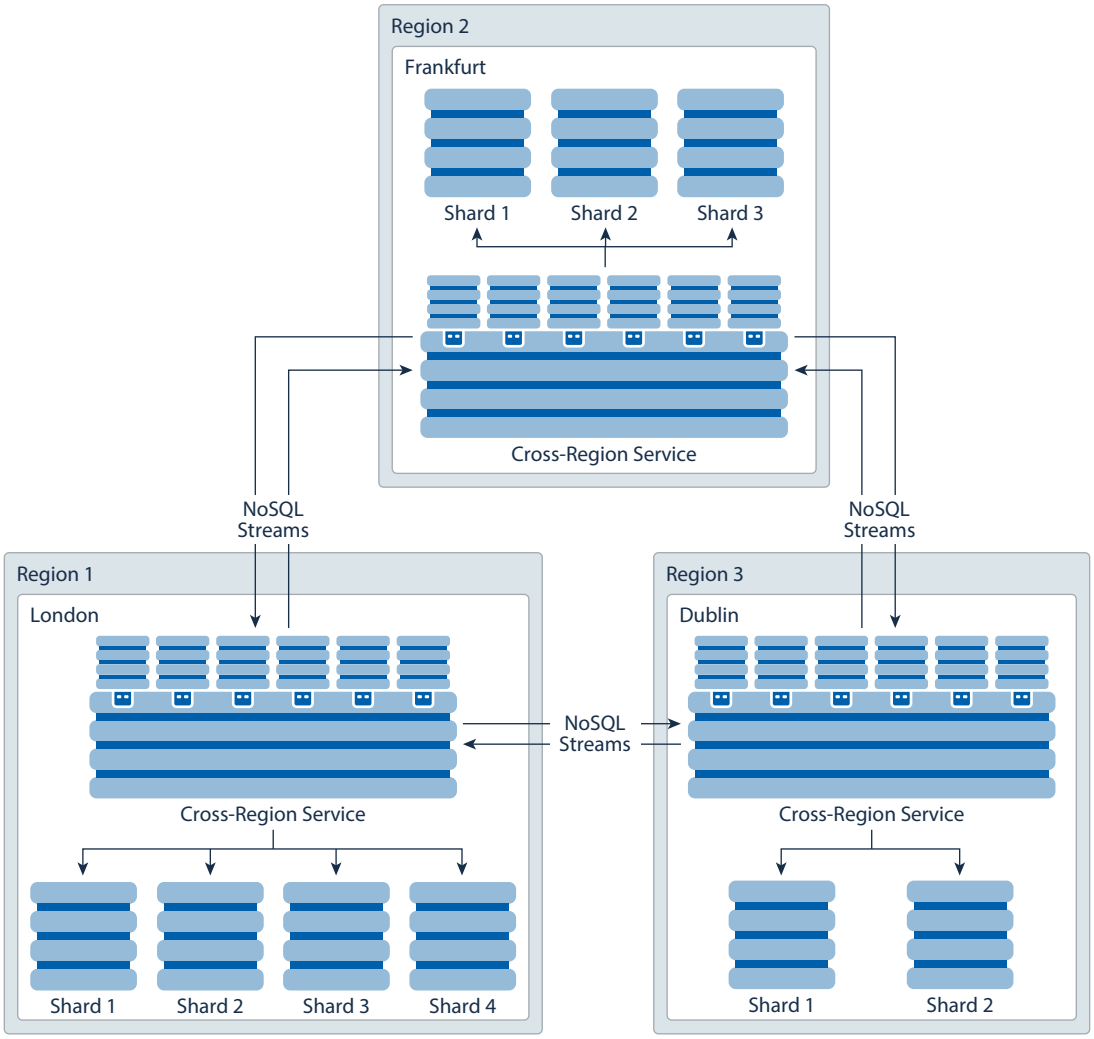
Finally, the RNs vote to determine which node should become the next Master node. For voting and deciding on a new master node, only the highest performance RNs can become master nodes in the Master Affinity zone. Once the next Master node is available, Oracle NoSQL directs all write requests and absolute consistency requirements to that Master.

Multi-Region Architecture

Oracle NoSQL Database applications read and write data by performing network requests against an Oracle NoSQL Database data store. Sometimes, organizations may need to set up multiple data stores to maintain their NoSQL data. In more realistic situations, these data store clusters may even be geographically distributed. Oracle NoSQL Database multi-region architecture enables you to create tables in multiple data store clusters and maintain consistent data across these clusters.

For example, consider a use-case where an organization deploys three on-premises data stores, one each at Frankfurt, London, and Dublin. In such a setup involving multiple data stores, each independent Oracle NoSQL Database installation is referred to as a *Region*. Such an architecture having two or more independent, geographically distributed data store clusters bridged by bi-directional NoSQL Streams is known as *Multi-Region Architecture*.

Figure 1-5 Multi-Region Architecture



Suppose you want to collect and maintain similar data across multiple regions. You need a mechanism to create tables that can span across multiple regions and keep themselves updated with the inputs from all the participating regions. You can achieve these using Multi-Region tables. A *Multi-Region Table* or *MR Table* is a global logical table that is stored and maintained in different regions or installations. It is a *read-anywhere* and *write-anywhere* table that lives in multiple regions.

As you can see in the diagram, all the Multi-Region Tables defined in these regions are synchronized via NoSQL Streams. Essentially, all the distributed data stores form a fully connected graph. For each distributed data store cluster, there is one inbound stream from each remote data store cluster. This inbound stream subscribes the local data store to all the Multi-Region Tables from the remote data store. Each region must be running a *Cross-Region Service (XRegion Service)* to receive the data from the subscribed tables in the remote regions.

In addition, note that in a Multi-Region Architecture are:

- The local and remote data stores:
  - May have different topology
  - May experience elastic operations.

  - Are independently managed, that is, each data store has its own index, security credentials etc.
  - Have sufficient create and read table privileges to each other.
- The inbound and outbound streams are:
  - Completely symmetrical.
  - Independently managed without any coordination between the outbound and the inbound streams.

Replication in a Multi-Region Table:

All writes to the table, including insert, update, and delete would be replicated. All DDL operations (Create Table, Alter Table and Drop Table, Create Index, Alter Index, and Drop Index) and operations that change the table metadata like TTL, will not be replicated. For example, the following actions will not be replicated.

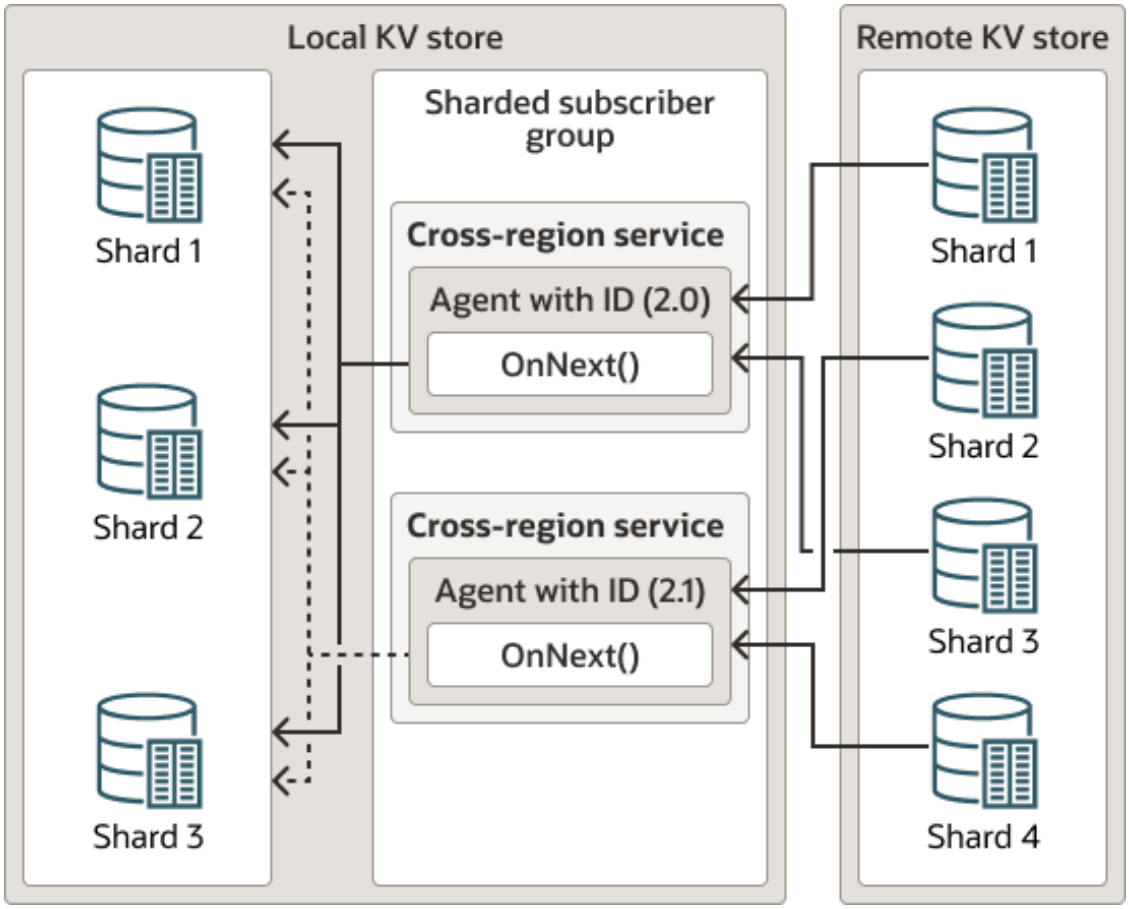
- Index creation in one region
- Altering the definition of an existing index from one region
- Dropping the index from one region
- Changing the schema definition in one region
- Changing the Table's default Table Time to Live (TTL) in one region

You can create child tables in the Multi-Region architecture. That means an existing Multi-Region table can have child tables. If you enable a top-level table in a Multi-Region architecture, the child tables created are automatically enabled in those regions. You need not explicitly specify the regions while creating the child tables. That is, the Multi-Region architecture is enabled for the whole hierarchy.

Cross Region Service

In a Multi-Region Oracle NoSQL Database setup, a *Cross-Region Service* or *XRegion Service* is a standalone service running on a separate node. In simple terms, this is also called an *agent*. The XRegion Service is deployed when you are connecting the local data store with a remote data store to create a Multi-Region Table.

Figure 1-6 Inbound Stream from Remote KVStore



Consider an example where the remote data store has four shards and the local data store has three shards. The local data store deploys two NoSQL agents to stream two shards each from the remote data store, as depicted in the diagram above. These agents subscribe to updates from the remote data store and publish the new and modified rows to the local data store. All the agents connecting a remote data store to a local data store are referred as the *Agent Group* for the local data store. Streams from multiple shards can coordinate by checkpointing to ensure that for any key, its writes on multiple shards in the data store during the migration will be delivered to the subscriber on their original order.

You need multiple XRegion Service agents when a single XRegion Service agent handles concurrent writes in different shards of the data store. You can determine whether additional XRegion agents are needed using the following steps.

- Determine if the source or target data store is overloaded. You can determine this using various application statistics like the latency of requests, any timeout, the CPU and memory usage of the Storage Node.
- If both the source and target data stores are not overloaded, then you can view the statistics of the XRegion agent. The show command can be used to view `mrtable-agent-statistics`. A sample output of the show command is shown below.

```
show mrtable-agent-statistics -agent 0 -json
{ "operation": "show mrtable-agent-statistics",
  "returnCode": 5000,
  description": "Operation ends successfully",
```

```
"returnValue": {
   "XRegionService-1_0":
   {
    "timestamp": 1592901180001,
    "statistics":
    { "agentId": "XRegionService-1_0",
      "beginMs": 1592901120001,
      "dels": 1024,
      "endMs": 1592901180001,
      "incompatibleRows": 100,
      "intervalMs": 60000,
      "localRegion": "slc1",
      "persistStreamBytes": 524288,
      "puts": 2048,
      "regionStat":
       { "lnd":
          { "completeWriteOps": 10,
          "laggingMs": { "avg": 512, "max": 998, "min": 31 },
          "lastMessageMs": 1591594977587,
          "lastModificationMs": 1591594941686,
          "latencyMs": { "avg": 20, "max": 40, "min": 10 }
          }
         ……
       }
    }
  }
}
```

Under the section **regionStat** there is a per-region field called `latencyMs (avg, max, min)`. You should monitor this stat over time. If the target is not overloaded and this stat keeps increasing, it is likely that XRegion agent cannot keep up with the remote writes and is running into a scalability issue.

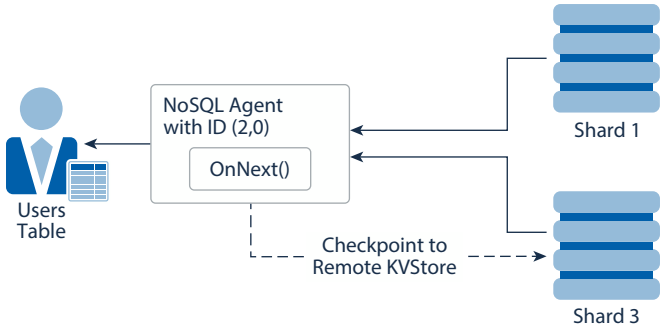
You can achieve horizontal scalability by adding more XRegion Service agents. The mapping of data store shards to XRegion Service agents is determined in a round-robin manner in order to balance the load of the agents.

> **Note**
>
> You should not configure more agents than the number of shards in your data store or else you will prevent XRegion Service agent from starting.

Each XRegion Service group consists of a group of independent XRegion Service agents, and each agent in the group is running on a node and is responsible to handle one or more shards of the data store. The agents in XRegion Service Group are completely independent of each other, that is, each agent does not talk directly to any other agent in the group. Any agent can be shut down and restarted without impacting other agents. It is recommended that you add XRegion Service agents on individual hosts that do not contain any Storage Node configured.

As you can see in the diagram below, an agent enables streaming the data from a remote data store to a set of MR Tables in the local data store.

Figure 1-7 View of a Single Agent



Each inbound stream utilizes the subscriber group feature in the Streams API to create a group of subscribers to stream from a store. Each local agent is responsible for:

- Establishing the inbound subscription stream from a remote data store.
- Maintaining the connection and reconnect during any failures.
- Checkpointing the subscription stream in case of any failures.
- Subscribing writes to the MR Tables from a remote data store.
- Automatically dealing with elastic operations in the remote data store.

For a local data store, the overhead of inbound stream consists of:

- A group of threads in the NoSQL agent that streams the data from the remote data store. Please note that these threads run outside the local store as a standalone agent. Even though the agent serves the local data store, it does not add to the local data store's expense.
- Local PUT or DELETE resulting from the streamed data from the remote data store after conflict resolution.

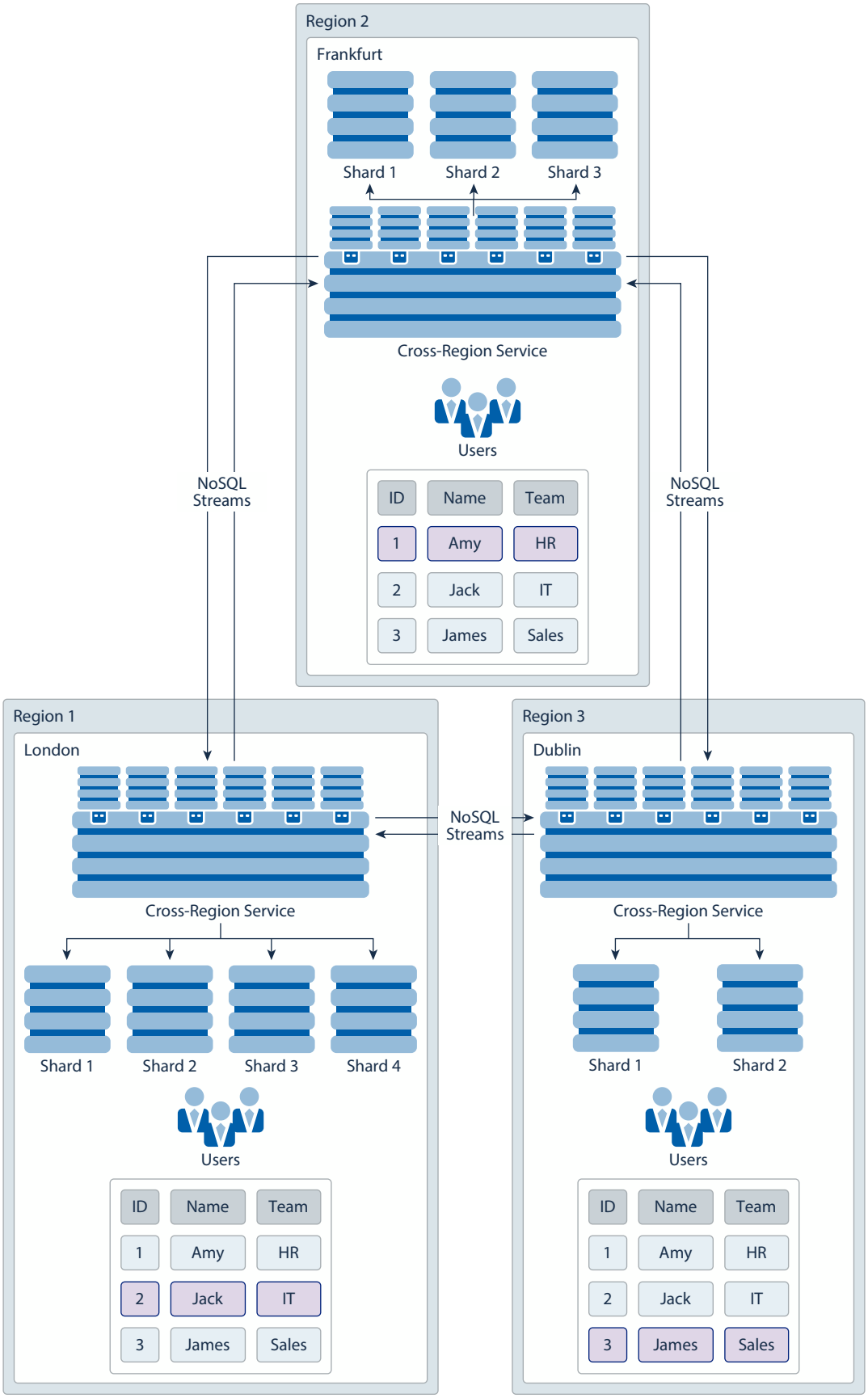
For a local data store, the overhead of outbound stream involves the create, read, and write checkpoints for the remote data store.

Life Cycle of Multi-Region Tables

To create and use Multi-Region Tables (MR Tables) in Oracle NoSQL Database, you must be aware of the sequence of tasks to execute and the related concepts.

For clarity, let us discuss the life cycle of MR Tables with an example. Consider an Oracle NoSQL Database with three regions, Frankfurt, London, and Dublin. Assume that you want to create a table called `Users` to store the user details for all the three regions as depicted in the diagram.

Figure 1-8 Multi-Region Table



The sequence of tasks that you must perform to create and manage the `Users` table as a Multi-Region Table are:

- **Deploy Independent data stores**: You must deploy data store in each region of the Multi-Region NoSQL Database independently. See Configuration Overview in the *Administrator's Guide*.
- **Set Local Region Name**: After deploying the data store and before creating the first MR Table in each participating region, you must set a local region name. You can change the local region name as long as no MR Tables are created in that region. After creating the first MR Table, the local region name becomes immutable. The local region name is completely independent of the data stores created in that region. See Set Local Region Name in the *Administrator's Guide*.
- **Configure XRegion Service**: Before creating any MR Table, you must deploy an XRegion Service with one or more agents. The agent runs independently with the local data store and it is recommended to deploy it close to the local data store.
  While setting up a secure data store to support multi-region tables, the administrator has to grant the following permissions to the XRegion Service agent:
  - `WRITE_SYSTEM_TABLE` (or the equivalent writesystable role) to the local store.
  - Write permission on all the multi-region tables in the local store.
  - Read permission on all the multi-region tables in the remote stores.
  - Write permission for checkpoint table in the remote stores.

  To learn how to deploy an agent, see Configure XRegion Service in the *Administrator's Guide*.
- **Start XRegion Service:**You must start XRegion service in each region using the `XRSTART` command. As this service is a long-running process, it is recommended to invoke it as a background process by appending the & at the end of the command. see Start XRegion Service in the *Administrator's Guide*.

  > **Note**
  >
  > The local data store must be started before starting the XRegion Service. If the data store in the local region has not started or is not reachable, the XRegion Service will not start.

- **Create Remote Regions**: Before creating and operating on the MR table, you must define the remote regions. A remote region signifies a region different from the region where the command is executed. In this example, to create the MR Table called `Users` from the Frankfurt region, you must first define the other two regions, that is, London and Dublin using the `CREATE REGION` DDL command. To learn how to create remote regions, see Create Remote Regions in the *Administrator's Guide*.
- **Create the MR Table**: You must create an MR Table on each data store in the connected graph, and specify the list of regions that the table should span. In this example, to create the `Users` table as an MR Table at the Frankfurt and London regions, you need to execute the `CREATE TABLE` command specifying Frankfurt and London as the regions. The order in which you list the regions in the DDL command does not matter. After you create the `Users` MR Table, it will be included in the incoming stream from each remote data store specified. Symmetrically at the remote data store the `Users` table will be included in its own incoming stream too. To create the MR table successfully, you must:

- Ensure that you acquire the necessary privileges to create the table in the specified regions, in advance. Otherwise, the MR Table creation will fail in all the regions. See Data store Required Privileges in the *Security Guide*.
- Specify at least one region in the `CREATE TABLE` DDL command. If you specify only one region, then the MR Table is created only in the specified region and no writes will be replicated to the other regions.

> **Note**
>
> Even though a single-region MR table works similar to a local table, the difference between them is that the single-region MR Table can be expanded to multiple regions in future.

To learn how to create an MR Table, see Create MR Tables in the *Administrator's Guide*.

- **Perform Read/Write Operations on the MR Table *(Optional)***: After creating the MR Table, you can perform read or write operations on the table using the existing data access APIs or DML statements. There is no change to any existing data access APIs or DML statements to work with the MR Tables. The following aspects applicable to the regular tables apply to the MR Tables also without any deviation:
  - **Durability and consistency configurations and constraints:** For any local writes to an MR table, the semantics of consistency model does not change. It is the same as any writes to a regular (non MR) table.

> **Note**
>
> In case of MR Tables, absolute consistency is not global across the participating regions. It is only local to a single region where you perform the read and write operations.

  - **Table index infrastructure:** Creating primary and secondary indices on the MR Table in each region remains the same as with any regular (non MR) table. However, if you wish to drop an MR Table from any region, you must first drop all the indices defined on this table.

Read Operations:

- Each Read operation on an MR Table is a local read, that is, you read only the local copy of the data. However, this local copy may have rows that might have come from one of the other participating regions, as a result of a table sync-up via Oracle NoSQL Streams.

Write Operations:

- Whenever you execute a write operation (INSERT, UPDATE, or DELETE) on an MR Table, the changes will be replicated across multiple regions asynchronously. It means, when you write a row in the local region, the write operation is executed completely in the local region without waiting for the subscribing regions to update.
- The latency for replicating the changes across multiple regions includes the time taken to:
  - Complete the write operations at the remote region, and
  - Receive the data from the subscribed tables.
- If multiple regions update a row with the same primary key, a built-in conflict resolution rule is applied to decide which region's update is considered as final. In all such cases,

this built-in conflict resolution rule will cause the update with the latest timestamp to win and commit to the database.

- The above mentioned built-in conflict resolution rule applies to the TTL value updates too.
- When you delete a row from an MR Table, the system allows time to ensure that the change is propagated to all the other regions where this table exists.
- You can define a TTL value while inserting or updating a row in the MR Table. This value applies only to the row being added or updated. The row-level TTL overrides table level TTL if any exists.
- An MR Table can have different table level TTL values in different regions.
- When a row is replicated to other regions, its expiration time is replicated as an absolute timestamp to the replicated rows. This can be either the default table level TTL value or a row level override that is set by your application. Therefore, this row will expire in all the regions at the same time, irrespective of when they were replicated.
- If a row expires before it makes it to one of the regions during replication, the rows that are already replicated to other regions will expire immediately after persistence.

See Access and Manipulate MR Tables in the *Administrator's Guide* for examples.

- **Add New Regions to the MR Table *(Optional)***: Oracle NoSQL Database lets you expand an MR Table to new regions. It effectively means adding new regions to an existing MR Table. In the example being discussed, suppose you created the `Users` table only in two regions, Frankfurt and London. Later, if you want to expand this `Users` table to the Dublin region, you must:
  - Create the `Users` MR Table in the new region, that is, Dublin. Note that you must specify all the three regions while creating the MR Table in the new region. See Create MR Table in New Region in the *Administrator's Guide*.
  - Add the new region (Dublin) to the `Users` MR Table in existing regions, that is, Frankfurt and London. This is achieved with the help of the `ALTER TABLE` DDL command. See Add New Region to Existing Regions in the *Administrator's Guide*.

  > **Note**
  >
  > Depending on the volume of the data in the existing regions, it might take some time to initialize the MR Table in the new region with the data from the other regions. However, the MR Table in the new region is available for read/write operations immediately after its creation.

  To learn how to expand an MR Table with detailed code demonstrations, see Use Case 2: Expand a Multi-Region Table in the *Administrator's Guide*.
- **Remove an Existing Region from the MR Table *(Optional)*:** Not only can you add new regions to an existing MR Table but also remove any regions linked to it. It effectively means that you disconnect the MR Table from a particular region so that MR Table is not synchronized with any writes from the removed region. This is called contracting an MR Table. To learn how to contract an MR Table with detailed code demonstrations, see Use Case 3: Contract a Multi-Region Table in the *Administrator's Guide*.
- **Drop Remote Regions *(Optional)***: You can drop one or more participating regions from a Multi-Region Oracle NoSQL Database setup as per your business requirement. However, before removing a region from a Multi-Region NoSQL Database, it is recommended to:
  - Stop writing to all the MR Tables linked to that region.

– Ensure that all writes to the MR Tables in that region have synchronized with other regions. This helps in maintaining consistent data across the different regions.

> **Note**
>
> Even though NoSQL Database lets you drop a region directly, it is a recommended practice to isolate that region from all the other regions before dropping it. This ensures that the existing regions are no longer linked with the region being dropped.

To learn how to drop a region from a Multi-Region NoSQL Database, see Use Case 4: Drop a Region in the *Administrator's Guide*.

- **Shut Down XRegion Service and data stores**: In a case where you want to relocate your XRegion Service to another host machine, you must shut it down in the current machine and then restart it in the new host machine. See Stop XRegion Service in the *Administrator's Guide*.

Using CRDT datatype in a multi-region table

Overview of the MR_COUNTER data type

MR_Counter data type is a counter CRDT. CRDT stands for Conflict-free Replicated Data Type. In a multi-region setup of an Oracle NoSQL Database, a CRDT is a data type that can be replicated across servers where regions can be updated independently and it converges on a correct common state. Changes in the regions are concurrent and not synchronized with one another. In short, CRDTs provide a way for concurrent modifications to be merged across regions without user intervention. Oracle NoSQL Database currently supports the counter CRDT which is called MR_Counter. The MR_COUNTER datatype is a subtype of the INTEGER or LONG or NUMBER data type. You can also use the MR_COUNTER data type in a schema-less JSON field, which means one or more fields in a JSON document can be of MR_COUNTER data type.

Why do you need MR_Counter in a multi-region table?

In a multi-region database configuration, copies of the same data need to be stored in multiple regions. This configuration needs to deal with the fact that the data may be concurrently modified in different regions.

Take an example of a multi-region table in three different regions (where data is stored in three different Oracle NoSQL Database stores). Concurrent updates of the same data in multiple regions, without coordination between the machines hosting the regions, can result in inconsistencies between the regions, which in the general case may not be resolvable. Restoring consistency and data integrity when there are conflicts between updates may require some or all of the updates to be entirely or partially dropped. For example, in the current configuration of a multi-region table in the Oracle NoSQL Database, if the same column (a counter) of a multi-region table is updated across two regions at the same time with different values, a conflict arises.

Currently, the conflict resolution is that the latest write overwrites the value across regions. For example, Region 1 updates column1 with a value R1, and region2 updates column1 with a value R2, and if the region2 update happens after region1, the value of the column (counter) in both the regions becomes R2. This is not what is actually desired. Rather every region should update the column (a counter) at their end and also the system internally needs to determine the sum of the column across regions.

One way to handle this conflict is making serializable/linearizable transactions (one transaction is completed and changes are synchronized in all regions and only then the next transaction happens). A significant problem of having serializable transactions is performance. This is where MR_COUNTER datatype comes in handy. With MR_COUNTER datatype, we don't need serializable transactions and the conflict resolution is taken care of. That is, MR_COUNTER datatype ensures that though data modifications can happen simultaneously on different regions, the data can always be merged into a consistent state. This merge is performed automatically by MR_COUNTER datatype, without requiring any special conflict resolution code or user intervention.

Use-case for MR_COUNTER datatype

Consider a Telecom provider providing different services and packages to its customers. One such service is a "Family Plan" option where a customer and their family share the Data Usage plan. The customer is allocated a free data usage limit for a month which your the customer's entire family collectively uses. When the total usage of customer's family reaches 90 percent of the data limit, the telecom provider sends the customer an alert. Say there are four members in customer's family plan who are spread across different physical regions. The customer needs to get an alert from the telecom provider once the total consumption of their family reaches 90 percent of the free usage. The data is replicated in different regions to cater to latency, throughput, and better performance. That means there are four regions and each has a data store containing the details of the customer's data usage. The usage of their family members needs to be updated in different regions and at any point in time, the total usage should be monitored and an alert should be sent if the data usage reaches the limit.

An MR_COUNTER data type is ideal in such a situation to do conflict-free tracking of the data usage across different regions. In the above example, an increment counter in every data region's data store will track the data usage in that region. The consolidated data usage for all regions can be determined by the system at any point without any user intervention. That is the total data usage at any point in time can be easily determined by the system using an MR_COUNTER datatype.

Types of MR_COUNTER Datatype

Currently, Oracle NoSQL Database supports only one type of MR_COUNTER data type. which is Positive-Negative (PN) counter.

Positive-Negative (PN) Counter

A PN counter can be incremented or decremented. Therefore, these can serve as a general-purpose counter. For example, you can use these counters to count the number of users active on a social media website at any point. When the users go offline you need to decrement the counter.

To create a multi-region table with an MR_COUNTER column, See Create multi-region table with an MR_COUNTER column section in the Administrator's Guide.

A MR_COUNTER (JSON and a non-JSON) can only be defined when the field in a schema is defined. You can do this in the following places:

- During schema definition in table creation.
- During schema definition when adding a field to the schema.

Data Models

You can model your data in Oracle NoSQL Database by using Tables or a key-value interface.

Tables are the easiest way to model data. They provide the highest level of abstraction, they are simple to model and should be familiar to any developer. This model also supports secondary indices and table evolution.

If you are using tables, then you can use JSON to model data. If strongly typed data is not a priority, then this is a good choice.

Oracle NoSQL Database also supports multi-region tables. A multi-region table is a global logical table that is stored and maintained in different regions or installations. In short, they are called as MR Tables.

Finally, if you want to serialize data, manage the key structure, manage secondary indices through index views, manage evolution and security through your client code, or work with large objects, then you can use the key-value interface.

Transactions in NoSQL

In an Oracle NoSQL Database, a transaction is a logical, atomic unit of work which entails one database access operation. In Oracle NoSQL Database every data operation takes place in a single transaction, managed by the system. Users do not have the ability to group multiple operations into a single transaction, although some operations allow multiple rows to participate in a single operation.

Transactional semantics are often described in terms of ACID properties.

ACID properties:

- **Atomicity** means a transaction either completes or fails in entirety. There is no state in between. You don't see a partial completion of a transaction.
- **Consistency** means the transaction leaves the database in a valid state.
- **Isolation** means no two transactions mingle or interfere with each other. You get the same result when the two transactions are executed in sequence or executed in parallel.
- **Durability** means the changes of a transaction are saved and the changes survive any type of failure (network, disk, CPU or a power failure).

Oracle NoSQL Database transactions maintain all these properties. Oracle NoSQL Database offers the user some control over the properties of a transaction. If your transaction involves a number of write operations on rows that share the same shard key, all of the write operations can be executed as a single atomic unit. So all of the operations will execute successfully, or none of them will.

The sequence of the write operations in the transaction is performed in isolation. This means that if you have a thread running a sequence of write operations, then another thread cannot intrude on the data in use by the sequence. The second thread will not be able to see any of the modifications made by the first running sequence until the sequence is complete.

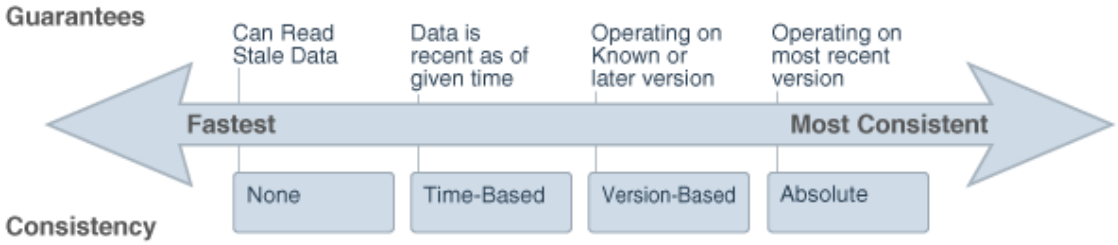
Atomicity and Isolation are not configurable but Oracle NoSQL Database allows users to control Consistency and Durability policies in order to trade off performance for applications that have differing needs for these properties.

Consistency

Oracle NoSQL Database provides several different consistency policies. At one end of the spectrum, applications can specify absolute consistency, which guarantees that all reads return the most recently written value for a designated key. At the other end of the spectrum, applications capable of tolerating inconsistent data can specify weak consistency, allowing the

database to return a value efficiently even if it is not entirely up to date. In between these two extremes, applications can specify time-based consistency to constrain how old a record might be or version-based consistency to support both atomicity for read-modify-write operations and reads that are at least as recent as the specified version.

The following illustration depicts the range of consistency policies that can be used by an application that makes use of Oracle NoSQL Database:



Flexible consistency policies enables developers to easily create business solutions providing data guarantees while meeting application latency and scalability requirements.

Durability

Oracle NoSQL Database provides a range of durability policies that specify what guarantees the system makes after a crash. At one extreme, applications can request that write requests block until the record has been written to stable storage on all copies. This has obvious performance and availability implications, but ensures that if the application successfully writes data, that data will persist and can be recovered even if all the copies become temporarily unavailable due to multiple simultaneous failures. At the other extreme, applications can request that write operations return as soon as the system has recorded the existence of the write, even if the data is not persistent anywhere. Such a policy provides the best write performance, but provides no durability guarantees.

The following illustration depicts the range of durability policies that can be used by an application that makes use of Oracle NoSQL Database:

By specifying when the database writes records to disk and what fraction of the copies of the record must be persistent (none, all, or a simple majority), applications can enforce a wide range of durability policies.

Quorum

Operations that modify data in Oracle NoSQL Database require that at least a simple majority of primary nodes be available to form a *quorum* in the shard that stores the specified key.

Quorum is the minimum number of primary nodes required in a shard, or in the set of admin nodes, to permit electing a master to support write operations. To form a quorum requires that a minimum number of primary nodes represent a majority in the group.

> **Note**
>
> Secondary nodes are not counted when computing the quorum.

Consider the following example using a store with four zones. Zones 1, 2, and 3 are primary zones with replication factor 1, and zone 4 is a secondary zone with replication factor 1. The number of primary nodes in each shard is 3, which is the sum of the replication factors for the primary zones. In a group of 3 nodes, 2 is the smallest number of nodes that represent a majority, so the quorum is 2. The secondary nodes in zone 4 have no impact on the quorum.

In general, to compute the quorum, first determine the primary replication factor, which is the sum of the replication factors of all primary zones. The quorum value must be one greater than half of the primary replication factor, rounding down when computing the half.

For example, for primary replication factor of 1, the quorum is 1. For primary replication factor of 5 the quorum is 3. For primary replication factor of 6, the quorum is 4.

Administration

The Administration command line interface (CLI) is the primary tool you use to manage your store. You use it to configure, deploy, and change store components. Use the CLI to verify the system, check the service status, check for critical events and browse the store-wide log file.

You can use the CLI to get, put, and delete store records or tables, retrieve schema, and display general information about the store. It can also be used to diagnose problems or potential problems in the system, fix actual problems by adding, removing, or modifying store data, and/or verify that the store has data. The CLI is particularly well-suited for a developer who is creating an Oracle NoSQL Database application. Developers can use the CLI to populate a store with a small number of records to use for development purposes or to examine the store's state as part of debugging activities.

Access the command line interface using this command:

```
java -Xmx64m -Xms64m \
  -jar KVHOME/lib/kvstore.jar runadmin \
  -host <hostname> -port <portname>
```

> **Note**
>
> To avoid using too much heap space, specify `-Xmx` and `-Xms` flags for Java when running administrative and utility commands.

For a complete list of all CLI commands and their usage, see Admin CLI Reference in the *Administrator's Guide*.

KVLite

KVLite is a simplified version of Oracle NoSQL Database. It provides a single-node store that is not replicated. It runs in a single process without requiring any administrative interface. You configure, start, and stop KVLite using a command line interface.

KVLite is intended for use by application developers who need to unit test their Oracle NoSQL Database application. It is not intended for production deployment, or for performance measurements.

KVLite is installed when you install the data store. It is available in the `kvstore.jar` file in the `lib` directory of your Oracle NoSQL Database distribution.

For more information on KVLite, see *Quick Start to KVLite*.

Saving Admin CLI History

By default, Oracle NoSQL Database uses the Java Jline library to support CLI history that you can save. To disable this feature, set the following Java property while starting the runadmin program:

```
java -Xmx64m -Xms64m \
-Doracle.kv.shell.jline.disable=true -jar KVHOME/kvstore.jar \
runadmin -host <hostname> -port <portname>
```

Unless you disable the feature, CLI history is saved to a file so that it is available after restart. By default, Oracle NoSQL Database attempts to save 500 lines of history in the following file, which is created and opened automatically:

```
KVHOME/.jlineoracle.kv.impl.admin.client.CommandShell.history
```

> **Note**
>
> If the Admin CLI cannot open the history file, it fails silently. The CLI runs without saving any history.

To change the default history file path, set the location of the `oracle.kv.shell.history.file="path"` Java property.

To change the default number of lines, set the value of the `oracle.kv.shell.history.size=<int_value>` Java property.

Monitoring

Information about the performance and availability of your store is available. You can monitor the information through an API class, log files, and Java Management Extensions (JMX).

These agents provide interfaces on each Storage Node that allow management clients to poll them for information about the status, performance metrics, and operational parameters of the Storage Node and its managed services, including replication nodes, and admin instances. Also, JMX can be used to monitor Arbiter Nodes.

For more information, see Java Management Extensions (JMX) in the *Administrator's Guide*.

Troubleshooting

Errors can occur in your store deployment. Tools, commands, logs and procedures can be used in order to solve problems.

To catch configuration errors early, you can use the `Diagnostics Utility`. You can also use this tool to package information and files to send them to Oracle Support, for example.

For more information on troubleshooting your store, see Troubleshooting in the *Administrator's Guide*.

Access and Security

There are two ways to access the data store and its data.

For routine access to the data, use Java APIs that application developers use to allow applications to interact with the Oracle NoSQL Database Driver. The drive communicates with the store's Storage Nodes to perform whatever data access the developer application requires.

For administrative access to the store, use the command line interface (CLI). System administrators use this interface to perform any actions that are required by Oracle NoSQL Database. You can also monitor the store using the CLI interface.

For most production stores, authentication over SSL is normally required by both the command line interface and the Java APIs. While you can install a store that does not require authentication, this is not recommended. For details on Oracle NoSQL Database's security features, see the Security Guide .

> **Note**
>
> Oracle NoSQL Database is intended to be installed in a secure location where physical and network access to the store is restricted to trusted users. For this reason, at this time Oracle NoSQL Database's security model is designed to prevent accidental access to the data. It is *not* designed to prevent denial-of-service attacks.

Integration

Oracle NoSQL Database can be integrated with Apache Hadoop and products in the Oracle stack. The following sections describe more about integration.

Hadoop Integration

Oracle NoSQL Database can be integrated with Apache Hadoop systems using the `oracle.kv.hadoop.KVInputFormat` class. This class allows you to read data from Oracle NoSQL Database and then prepare it for insertion into a Hadoop system. To move data back to your Oracle NoSQL Database, you can read data from the Hadoop system using the standard mechanisms, and then write the records to Oracle NoSQL Database using the APIs. An example of using `KVInputFormat` to read data from Oracle NoSQL Database in a Map/Reduce job can be found in the `<KVHOME>/examples/hadoop` directory. See Integration with Apache Hadoop MapReduce in the *Integrations Guide*.

Integration with Elastic Search for Full Text Search

Full Text Search provides the capability to identify natural-language documents that satisfy a query, and optionally to sort them by relevance to the query. Oracle NoSQL Database integrates with the Elasticsearch third-party open-source search engine to enable Full Text Search capability against data stored in an Oracle NoSQL Database table. See Integration with Elastic Search for Full Text Search in the *Integrations Guide*.

Oracle External Tables Integration

Oracle NoSQL Database data can be accessed using Oracle Database's External Tables feature. This capability allows NoSQL Database data to be read into Oracle Database. Oracle NoSQL Database data cannot be modified using the External Tables feature.

Note that this is a feature which is only available to users of the Oracle NoSQL Database Enterprise Edition.

To use the Oracle Database External Table feature to read Oracle NoSQL Database data, you must use the `<KVHOME>/exttab/bin/nosql_stream` preprocessor to populate our Oracle tables with the data. You must then configure your Oracle Database to use the External Tables feature.

For information on how to use the `nosql_stream` preprocessor, and how to configure Oracle Database to use External Tables, see oracle.kv.exttab package summary in the *Java Direct Driver API Reference*.

Oracle GoldenGate Integration

The transactional data from Oracle GoldenGate can be replicated to a target Oracle NoSQL Database using Oracle NoSQL Handler. The Oracle NoSQL Handler streams change data capture into Oracle NoSQL Database using the Oracle NoSQL Java SDK. The Oracle NoSQL Java SDK supports both on-premise and OCI cloud instances of Oracle NoSQL database.

The Oracle NoSQL Handler moves operations to Oracle NoSQL using synchronous API. The insert, update, and delete operations are processed differently in Oracle NoSQL databases rather than in a traditional RDBMS.

The following explains how insert, update, and delete operations are interpreted by the handler depending on the mode of operation:

- **insert:** If the row does not exist in your database, then an insert operation is processed as an insert. If the row exists, then an insert operation is processed as an update.
- **update:** If a row does not exist in your database, then an update operation is processed as an insert. If the row exists, then an update operation is processed as update.
- **delete:** If the row does not exist in your database, then a delete operation has no effect. If the row exists, then a delete operation is processed as a delete.

The state of the data in Oracle NoSQL databases is idempotent. You can replay the source trail files or replay sections of the trail files. Ultimately, the state of an Oracle NoSQL database is the same regardless of the number of times the trail data was written into Oracle NoSQL.

You configure the Oracle NoSQL Handler operation using the properties file. These properties are located in the Java Adapter properties file.

For more information, see Using the Oracle NoSQL Handler in the *Using Oracle GoldenGate for Big Data*.