Oracle® NoSQL Database

Streams Developer's Guide

Release 25.3
E91416-25
October 2025

Oracle NoSQL Database Streams Developer's Guide, Release 25.3

E91416-25

Contents

Preface

This document is intended to provide a rapid introduction to the Oracle NoSQL Database Streams API. This API is an implementation of the Reactive Streams standard, which provides for asynchronous stream processing with non-blocking back pressure. See Reactive Streams.

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 the Oracle NoSQL Database Streams API

The Oracle NoSQL Database Streams API lets you subscribe to all logical changes (`puts` and `deletes`) made to Oracle NoSQL Database tables. A subscription stream includes the changes made by insert, update, and delete operations on tables. In addition, the Streams API allows you to configure your subscription to optionally include features such as streaming before-images of the rows along with the after-images and transaction event streaming. You can stream either a discrete list of committed write operations, or a single committed transaction event containing a list of write operations performed in the transaction event.

The Streams API also allows you to configure both before-images and transaction events. That is, you can stream transaction events, and each operation in the transaction event also carries before-images, provided, the before-images are enabled for that table and are within their expiration dates.

These changes are streamed to your application as a series of discrete `StreamOperation` class objects. This API is based on the Reactive Streams standard. See Reactive Streams.

Oracle NoSQL Database Streams APIs are prefixed with `NoSQL` to differentiate them from the APIs described by the Reactive Streams standard. For example, Reactive Streams describes a Publisher class. The Oracle NoSQL Database implementation of that class is called `NoSQLPublisher`.

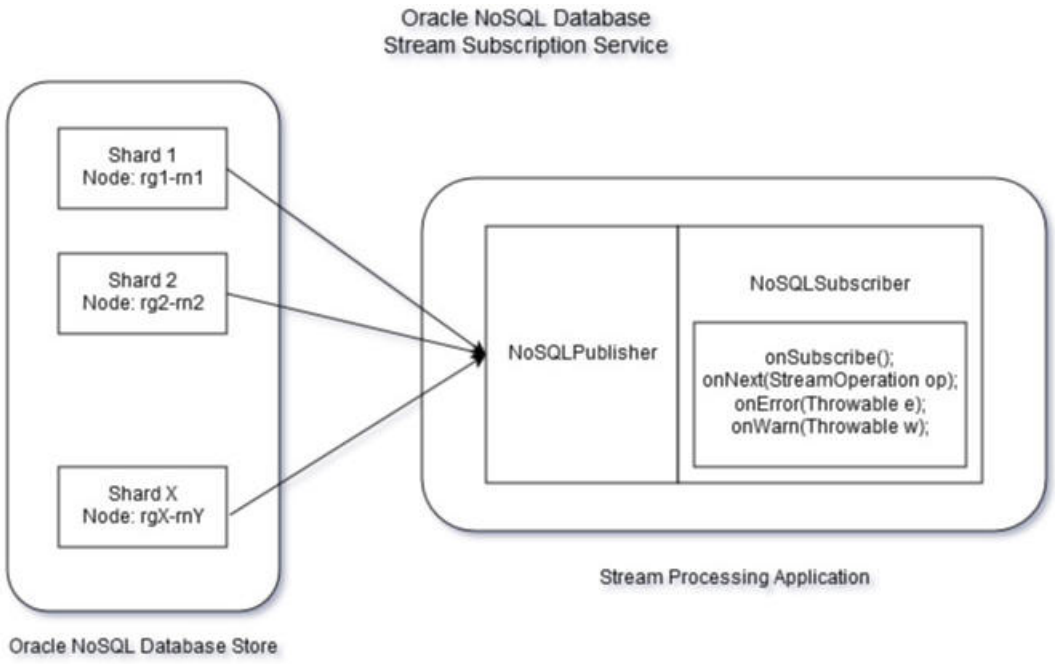
> **Note**
>
> The Oracle NoSQL Database Streams API supports table namespaces. If you reference a table that is in a namespace, prefix the table, such as `Users`, with its namespace name (such as `ns1`) followed by a colon (`:`). For example, `ns1:Users`.

The remainder of this chapter provides an introduction to the Oracle NoSQL Database Streams API.

Architecture

The Oracle NoSQL Database streams API offers access to a store-wide stream of all the insert, update, and delete operations made to the tables that an application has subscribed to.

The following illustration depicts the overall architecture of the Oracle NoSQL Database streaming service:



Each shard in the Oracle NoSQL Database supplies a stream of updates from the shard. The `NoSQLPublisher` aggregates all shard streams to implement a unified publisher interface for the entire Oracle NoSQL Database store that it presents to users.

A `NoSQLPublisher` is responsible for creating, managing, and canceling all subscriptions that the user creates.

> **Note**
>
> The current Streams API continues to stream data while store elasticity operations are in progress. When Storage Nodes, Replication Nodes, or Admins are being added to the store, streaming continues without effect on performance.

API Components

The Reactive Streams standard, available at (http://www.reactive-streams.org) describes various API components. These are all implemented for the Oracle NoSQL Database Streams API. Briefly, these are the major API components:

API	Description	More Informatin
`NoSQLPublisher`	Aggregates table operations into a single stream, and publishes them according to the configuration received from its subscriber(s). For any Java JVM, create only one `NoSQLPublisher` class instance for a publisher configuration. You can create multiple publishers per JVM, as long as their configurations differ. For example, you can create a single JVM to support two or more publishers, each connecting to different stores with their different configurations. You can scale the streaming service to use multiple subscribers to stream events from Oracle NoSQL Database store.	`NoSQLPublisher` is introduced in the chapter titled Using a Streams Publisher. For more information about scaling the streaming service to use multiple substribers, see Scaling a Streams Application. For more information on `NoSQLPublisher`, see NoSQLPublisher.
`NoSQLSubscriber`	Interface that you must implement to define how to process each store operation, warning, and error. `NoSQLPublisher` manages each `NoSQLSubscriber` instance that you provide. The `NoSQLPublisher` can use multiple `NoSQLSubscriber` instances.	For an introduction to subscriber implementation, see Implementing Subscribers. For information about how to use multiple subscribers streaming events from Oracle NoSQL Database store, see Scaling a Streams Application. For more information on `NoSQLSubscriber`, see NoSQLSubscriber.
`NoSQLSubscription`	Represents a single subscription that you created. This class receives `write` and `delete` events present in the operations stream, in the form of `StreamOperation` objects, and provides them to your `NoSQLSubscriber` implementation for processing.	For an introduction to `NoSQLSubscription`, see Working with Subscriptions. The `StreamOperation` class is introduced in Using the StreamOperation Class. For more information on `NoSQLSubscription`, see NoSQLSubscription.

Checkpoints

When a subscriber opens a subscription stream, the subscriber can start consuming events from the earliest available point in the stream, or from some other location in the stream. To begin consuming from another location, the application must have run and saved at least one checkpoint (perhaps more). Checkpoints represent different locations in the stream. For example, your application could save a stream checkpoint after the publisher has streamed every 1024 records.

Your application can take a checkpoint at any time; however, only one checkpoint may be in progress at any given time. The most recent checkpoint is saved in a checkpoint table within the Oracle NoSQL Database store. If you want to save more than the most recent checkpoint, you must manually save it to disk or to a database of your choosing.

For an introduction to checkpoints, see Using Checkpoints. For information about NoSQLStreamMode, see Implementing Checkpoints in GSGStreamExample.

Dynamic Streaming

The Oracle NoSQL Database supports Dynamic Streaming. Dynamic streaming enables a user to add or remove a table in a live stream at run time without shutting down and recreating a new stream. An empty stream which does not have any tables can be created, and later the

tables can be added at runtime. Also, all the existing tables in a stream can be deleted such that the stream is made empty. A subscription is called empty when it does not have at least one table in it. The duration for which an empty stream should be kept alive can be determined by the user.

Row metadata Streaming

> **Note**
>
> This feature has been made available to you on a "preview" basis so that you can get early access and provide feedback. It is intended for demonstration and preliminary use only. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to this feature and will not be responsible for any loss, costs, or damages incurred due to the use of this feature.

Oracle NoSQL Database supports annotating row data changes with additional row metadata. This information is delivered to subscribers of a change stream. The Streams API enables you to subscribe to changes made to the table, including row metadata, providing a reliable way to monitor changes, track, and audit data modifications. For more information about Streams API, see Using the StreamOperation Class.

- When a new row is inserted, both the row data and the associated metadata are included in the Streams API.
- When an existing row is updated, the updated row data and its metadata are included in the stream. This allows you to track how both the data and metadata are modified over time.
- When a row is deleted, the Streams API includes the metadata that was associated with the deleted row.
- When multiple rows are deleted in a batch, metadata for each deleted row is included in the change stream ensuring consistent and reliable tracking across both single and multi-row delete operations.

Additionally, see Using user-defined row metadata, in *Developers Guide*, for in-depth information on using row metadata.

Before-images Streaming

Learn about streaming before-images to your applications using the Oracle NoSQL Database Streams API.

Oracle NoSQL Database supports streaming before-images. Before-images enable applications to compute the change or delta made by write operations to tables in a data store.

The before-image of any write is the table row before it gets updated or deleted by a DML operation. By default, Oracle NoSQL Database does not generate before-images. After before-images are enabled using either the CREATE TABLE or ALTER TABLE DDL statement, you can configure your subscription to include before-images of subscribed tables in addition to after-images.

To explicitly generate before-images during any writes to a table and stream it to subscribers, you must:

1. Enable before-images for the table using the ENABLE BEFORE IMAGE syntax. For more details, see Enabling Before-images During Table Creation and ALTER TABLE statement topics.
2. Configure the subscription to stream before-images to the subscriber for the table of interest. For method details, see Using NoSQLSubscriptionConfig.
3. Implement the subscriber to receive a stream with before-images events. For method details, see Using the StreamOperation Class.

Following table summarizes the content of a before-image for a DML operation on any user table, either local or Multi-Region:

Table 1-1 Before-images for DML Operations

DML operation	Before-image content
INSERT	NULL
UPDATE	row before update
DELETE	deleted row

For either an update or a delete operation, the before-image is the row before it is updated or deleted. For insert operations, before-images are empty.

Note

1. Enabling/disabling before-images is on a per table basis and can be done anytime during the lifetime of a table.
2. A TTL definition sets the expiry for before-images. You can define before-images TTL either during the creation of a table or later using an ALTER TABLE statement. For more details, For more details, see Enabling Before-images During Table Creation and ALTER TABLE statement topics.

   After their expiration, the before-images will not appear in the stream.
3. It is possible to change the TTL value of before-images during the lifetime of a table using the ALTER TABLE statement. The before-images that are generated after the TTL update reflect the new expiry. The existing before-images expiry remain unaffected.

   For example, if the before-images are already generated with a TTL of five days and you modify the before-images TTL to one day, then the before-images that are newly generated after the TTL update expire in one day. There is no impact to the expiry of existing before-images.
4. If you have set a small TTL value for before-images and the streaming is lagging, the before-images can expire before the streaming becomes operative, resulting in a loss of the before-images from the stream.
5. It is possible for a table to have both a table TTL and a before-images TTL. These two TTL values work independently of each other. As a result, it is possible for a data row to have expired, while the row's before-image may still be active. In such cases, the Streams API can still stream the before-image until its expiry. The reason being, even though the row has expired, it will still be available for streaming until the row is removed from the transaction log.
6. Queries and DDL operations do not generate before-images.
7. The streams can start or resume from any arbitrary location if the location is available. Therefore, streams may include events without before-images if the events were created prior to enabling before-images or after disabling them.
8. A streamed event is self-describable. From any stream event, a subscriber can distinguish between the following cases by implementing the corresponding methods in the subscriber. For more information, oracle.kv.pubsub.StreamOperation class summary in the — *Java Direct Driver API Reference*
   - Before-images are not enabled for an event.
   - Before-images are enabled but the before-image has expired.
   - Before-images are enabled and has not expired, however, the event does not carry the before-image. For example, an INSERT operation.

Transaction Events Streaming

Learn about streaming transaction events to your applications using the Oracle NoSQL Database Streams API.

Oracle NoSQL Database supports an optional feature that allows you to configure transaction events streaming in your subscription. A transaction is a logical sequence of database operations treated as a single unit of work.

All write operations in a single committed transaction are encapsulated in a single stream operation and delivered to subscribers. The default behavior is each committed write, either a PUT or a DELETE, is a single stream operation.

For example, for a transaction with 10 updates that is committed on a data store, the Streams API will include a discrete list of 10 committed write events. However, if you have configured to stream transaction events, the Streams API will include a single committed transaction event containing a list of 10 write operations performed in the transaction event.

To explicitly deliver transaction events to subscribers, you must:

1. Configure the subscription to stream transaction events for the table of interest. For method details, see Using NoSQLSubscriptionConfig.
2. Configure the subscriber to receive a stream with transaction events. For method details, see Using the StreamOperation Class.

> **Note**
>
> The Streams API allows you to configure a subscription to stream either a list of individual committed write operations, or a list of committed transaction events containing one or more writes for a table. If you already have a default subscription and want to stream transaction events, you must unsubscribe the table and resubscribe it to deliver transaction events.

System Requirements

The Oracle NoSQL Database Streams API requires Java 11 or later version. It is recommended to use Java 17.

Limitations

The stream will **not** include a separate notice of DDL operations (table creations, alterations, and table deletions). After the table is dropped, your stream will no longer receive any streamed operation from the dropped table. It is your responsibility to check with the server if the table has been dropped. Then, decide if the stream shall be canceled after the table is dropped.

Compiling and Running a Streams Application

The Oracle NoSQL Database Streams APIs can be found in the `kvstore.jar` file, which is located in your distribution's `lib` directory.

To run the Streams-related examples in the distribution, first get the `Examples` download from Oracle Technology Network (OTN) and run the following commands:

- `cd $KVHOME/examples`
- `javac -cp $KVHOME/lib/kvstore.jar:. pubsub/NoSQLStreamExample.java`
- `java -cp $KVHOME/lib/kvstore.jar:. pubsub.NoSQLStreamExample`

Example Usage:

```
+ NoSQLStreamExample
    [create-table | load-table | subscribe | cleanup]
    -store <instance name>
    -table <table name>
    -host <host name>
    -port <port number>
    -num <number of rows>
    -checkpoint <checkpoint interval in number of rows>
    -from [now | checkpoint | exact_checkpoint]
```

2

Using the StreamOperation Class

A streams application works by implementing Subscribers. Subscribers receive a stream of events that consist of write operations to a table of interest.

For more information, see Implementing Subscribers.

Every event your application receives in a subscription stream is represented as an `oracle.kv.pubsub.StreamOperation`. Each of these events represents a `put`, `delete`, or `transaction` operation on the table that your application subscribes to.

> **Note**
>
> A `put` represents either a `put` API call or a SQL DML statement that inserts or updates one of more records.

For more information, see oracle.kv.pubsub.StreamOperation class summary in the — *Java Direct Driver API Reference*

The `StreamOperation` interface provides the following methods:

- `StreamOperation.getType()`

  Returns a `StreamOperation.Type` object. This is an enum constant that is either `delete` or `put`. For example:

  ```
  // so is a StreamOperation object. It is obtained using
  // NoSQLSubscriber.onNext().
  switch (so.getType()) {
      case PUT:
          {
              // Process the put operation here.
          }
          break;
      case DELETE:
          {
              // Process the delete operation here.
          }
          break;
      case TRANSACTION:
          {
              // Process the transaction operation here.
          }
          break;
      default:
          // Received an unknown and therefore illegal operation type.
          throw new IllegalStateException("... exception message ...");
  }
  ```

- `StreamOperation.asDelete()`

Returns the operation as a `StreamOperation.DeleteEvent` object. The object contains only the Primary Key associated with the `delete` operation:

```
// so is a StreamOperation object. It is obtained using
// NoSQLSubscriber.onNext().
StreamOperation.DeleteEvent de = so.asDelete();
PrimaryKey pk = de.getPrimaryKey();
```

- `StreamOperation.asPut()`

  Returns the operation as a `StreamOperation.PutEvent` object. This object allows you to obtain the row that was changed by the `put` operation.

  > **Note**
  >
  > – If before-images are not enabled, the events returned represent the state of the rows after the `put` operations are performed.
  >
  > – If before-images are enabled and the stream is configured to include before-images, the events returned represent the state of the rows after and before the `put` operations are performed.

```
// so is a StreamOperation object. It is obtained using
// NoSQLSubscriber.onNext().
StreamOperation.PutEvent pe = so.asPut();
Row row = pe.getRow();
```

- `StreamOperation.asTransaction()`
  Returns the operation as a `StreamOperation.TransactionEvent` object.

  The operations in the transaction event stream will be in the same order they were performed on a shard. The `StreamOperation.TransactionEvent` object allows you to obtain the following internals for the transaction event:

  – `getTransactionId()`: Returns the transaction ID that helps uniquely identify a transaction in the data store. The transaction ID is a store-wide unique value and consists of the shard ID, internal transaction ID, and timestamp when the transaction is committed. This is useful in various scenarios like monitoring, debugging, and so on.

  > **Note**
  >
  > The timestamp is in the time zone of the data store where the transaction event is generated and in the UNIX epoch format.

  – `getTransactionType()`: Returns the transaction event as COMMIT type.

  – `getNumOperations()`: Returns the number of write operations in the transaction.

  – `getOperations()`: Returns an ordered list of `put` and `delete` operations in the transaction event.

```
// 'so' is a StreamOperation object. It is obtained using
// NoSQLSubscriber.onNext().
```

```
StreamOperation.TransactionEvent te = so.asTransaction();
trace("Transaction event=" + te.toJsonString());
```

- `StreamOperation.getRepGroupId()`

  Returns the Shard ID (as an `int`) where this `write` operation was performed.

- `StreamOperation.getSequenceId()`

  Returns the unique sequence ID associated with this operation. This ID uniquely identifies a stream operation associated with a given Publisher.

  These IDs can be used to sequence operations seen for a given key. The Subscription API guarantees that the order of events for a particular key is the same as the order in which these operations were applied in Oracle NoSQL Database. The subscription API provides no guarantees about the order of operations beyond the single key.

You can use the following methods from the `StreamOperation` interface to verify and process events with before-images for a table.

- `StreamOperation.isBeforeImageEnabled()`: Returns true if you have enabled before-images for the subscribed table when the current `Streamoperation` event is made to the server. Otherwise, returns false.
- `StreamOperation.includeBeforeImage()`: Returns true if you have configured the subscription to include before-images. Otherwise, returns false.
- `StreamOperation.isBeforeImageExpired()`: Returns true if the before-image has expired from the persistent storage. Otherwise, returns false.

  > **Note**
  >
  > This method also relies on the table being enabled to generate before-images prior to the delivery of this `StreamOperation` event.

  For example, consider a scenario when you enable before-images with a 24 hours TTL. Streaming starts 36 hours after before-images were generated. The expiry check returns true, and before-images will not appear in the stream.

- `StreamOperation.getBeforeImage()`: Returns the before-image associated with the stream event represented by the current `StreamOperation` event for the row. This method returns null in the following cases:
  - You have not configured the stream to include before-images.
  - You have not enabled before-images for the table.
  - You have enabled before-images, however, it is not generated for write operations such as INSERT.
  - You have enabled before-images and configured stream to include it, however, the before-image has expired.

3

Working with Subscriptions

`NoSQLSubscription` is used to manage an active subscription to the Oracle NoSQL Database store. A Subscription is configured using `NoSQLSubscriptionConfig`. `NoSQLSubscriptionConfig` is used to identify important information such as what table(s) you want to subscribe to.

In this chapter, we first show how to use `NoSQLSubscriptionConfig`, and then show how to use `NoSQLSubscription`.

Using NoSQLSubscriptionConfig

You configure your subscription by building an `oracle.kv.pubsub.NoSQLSubscriptionConfig` object. You then provide this object to your `NoSQLSubscriber` implementation, and also implement `NoSQLSubscriber.getSubscriptionConfig()` to return this object when it is called. When you construct the publisher, you will provide it with your `NoSQLSubscriber` implementation, and the publisher will then call `NoSQLSubscriber.getSubscriptionConfig()`. In order to understand how to create the subscription, see Implementing Subscribers and Using a Streams Publisher.

Configuring a single subscriber:

To build your `NoSQLSubscriptionConfig` object, you use `NoSQLSubscriptionConfig.Builder` as follows:

```
final NoSQLSubscriptionConfig subConfig =
    new NoSQLSubscriptionConfig.Builder("ChkptTable")
    .setSubscribedTables("UserTable")
    .setStreamMode(NoSQLStreamMode.FROM_NOW)
    .build();
```

Configuring multiple sharded subscribers as a part of a subscriber group:

To build a `NoSQLSubscriptionConfig` object for a sharded subscriber of a subscriber group, you use a constructor of the `NoSQLSubscriptionConfig.Builder` class that has a single parameter of a mapper function. This function returns a distinct checkpoint table name computed from each subscriber id in the group. A sample constructor of the Builder class for configuring sharded subscribers is shown below.

```
public void Builder(Function<NoSQLSubscriberId, String> ckptTableMapper);
```

You can build the `NoSQLSubscriptionConfig` object for a sharded subscriber as follows:

```
/**
* Configures a sharded subscriber
* @param subscriberId subscriber id
* @param tables subscribed tables
*/
```

```
void configureShardedSubscribers(NoSQLSubscriberId subscriberId, Set<String>
tables) {
final NoSQLSubscriptionConfig conf =
    new NoSQLSubscriptionConfig.Builder( id -> buildCkptTableName(id)
    .setSubscribedTables(tables)
    .setSubscriberId(subscriberId)
    .build();
}
```

An example code snippet is shown below where the `buildCkptTableName` builds the checkpoint name from the subscriber id by concatenating the subscriber id to `StreamCheckpointTable`. This is one way of using the mapper function. Your application can create its own mapper function, as long as it returns unique checkpoint table name for each subscriber.

The mapping has to be static throughout the lifetime of the XRegion Agent group, not for a specific data store instance. That is, for a data store, you can create a group of 3 agents with a mapping function. Later on, you can shut the agents down and create another group of agents with another mapping function. These two groups use different set of checkpoint tables and thus there will be no conflict.

```
/**
 * Builds stream checkpoint table from given subscriber id
 * @param subscriberId subscriber id
 * @return stream checkpoint table name for that subscriber
 */
private String buildCkptTableName(NoSQLSubscriberId subscriberId) {
    return "StreamCheckpointTable" + subscriberId.toString();
}
```

This configuration causes the subscription to:

- Use the checkpoint table called `ChkptTable`. This table is automatically created. For more information about checkpoint table, see Using Checkpoints. Be aware that the table name used here is chosen by you, and must be unique to your subscription. If you are using multiple subscriptions, then each subscription should use a unique name for the checkpoint table.

  If you are using a secure store, you need read/write access to the checkpoint table. If a checkpoint table does not exist, you also need the `CREATE TABLE` privilege. For information about:

  - Connecting to a secure store, see Authenticating to a Secure Store.
  - Configuring Authorization for a secure store, see Privileges in the *Security Guide*.
- Subscribe to all `write` activity performed on the user table called `UserTable`. Subscriptions can be created for user-defined tables; updates to system tables are not streamed. You can use this to subscribe to multiple tables:

  ```
  new NoSQLSubscriptionConfig.Builder("ChkptTable")
   .setSubscribedTables("UserTable", "PriceTable", "InventoryTable") ....
  ```

  If you do not call `setSubscribedTables()`, then the subscription will subscribe to all tables in the store. If a subscription is for every table in the store, and a new a table is created after subscription is established (using the `DDL CREATE TABLE` operation), the stream will include all `put` events for every row created in the new table.

- Set the stream mode to `NoSQLStreamMode.FROM_NOW`. The stream mode indicates from where in the stream the Publisher will start retrieving events. For more information, see NoSQLStreamMode.

After you have created your subscription configuration, you provide it to your `NoSQLSubscriber` implementation, which then must make it available through the `NoSQLSubscriber.getSubscriptionConfig()` method:

```
class mySubscriber implements NoSQLSubscriber {
    ...
    private final NoSQLSubscriptionConfig config;
    ...
    // Generally the constructor will require more than just
    // the subscription configuration. The point here is that you
    // must somehow provide the configuration object to
    // your subscriber implemention because that is how
    // your publisher will get it.
    mySubscriber(NoSQLSubscriptionConfig config, ....) {
        ...
        this.config = config;
        ...
    }
    @Override
    public NoSQLSubscriptionConfig getSubscriptionConfig() {
        return config;
    }
    ...
```

When you implement your streams application, you will use your subscriber implementation. The `getSubscriptionConfig()` method on the subscriber is how your publisher finds out what tables to follow, and so forth. See Using a Streams Publisher.

You can specify the expiration time for an empty stream using the `NoSQLSubscriptionConfig.setEmptyStreamDuration()` method. The expiration time will begin only when a stream becomes empty after which the publisher will shut down the empty stream. The default empty stream expiration time is 60 seconds. You can override the default empty stream expiration time by using the `setEmptyStreamDuration()` method.

This *Using NoSQLSubscriptionConfig* section covers only a few options that you can set using `NoSQLSubscriptionConfig`. You can optionally configure features such as before-images streaming and transaction events streaming. For details, see Configuring Before-images Streaming and Configuring Transaction Event Streaming sections.

> **Note**
>
> The Streams API also allows you to configure streaming before-images and transaction events at the same time. That is, you can stream transaction events, and each operation in the transaction event also carries before-images, provided before-images are enabled for that table and are within their expiration dates.

For a complete list of configuration options, see NoSQLSubscriptionConfig and NoSQLSubscriptionConfig.Builder in the *Java Direct Driver API Reference*.

Configuring Before-images Streaming

Learn to configure streaming before-images using the Oracle NoSQL Database Streams API.

You can optionally configure the subscription to enable before-images in the streamed events using the `setIncludeBeforeImage()` method as follows:

```
final NoSQLSubscriptionConfig subConfig =
    new NoSQLSubscriptionConfig.Builder("ChkptTable")
    .setSubscribedTables("UserTable")
    .setStreamMode(NoSQLStreamMode.FROM_NOW)
    .setIncludeBeforeImage(true)
    .build();
```

As a pre-requisite, you must have enabled before-images generation for the subscribed table. For details, see Enabling Before-images During Table Creation

Configuring Transaction Event Streaming

Learn to configure streaming transaction events using the Oracle NoSQL Database Streams API.

You can optionally configure subscriptions to stream transaction events using the `setStreamDeliveryMode()` method as follows:

```
/* create subscription to stream transaction events for UserTableTxn tables */

import static
oracle.kv.pubsub.NoSQLSubscriptionConfig.StreamDeliveryMode.GROUP_ALL_IN_TRANS
ACTION;

...
final NoSQLSubscriptionConfig subConfig =
    new NoSQLSubscriptionConfig.Builder("ChkptTable")
    .setSubscribedTables("UserTableTxn")
    .setStreamMode(NoSQLStreamMode.FROM_NOW)
    .setStreamDeliveryMode(GROUP_ALL_IN_TRANSACTION,"UserTableTxn")
    .build();
```

The `setStreamDeliveryMode()` method expects `StreamDeliveryMode` and table name(s) as arguments.

- The `StreamDeliveryMode` argument takes either GROUP_ALL_IN_TRANSACTION or SINGLE_WRITE_EVENT Enum value. You must supply GROUP_ALL_IN_TRANSACTION to stream transaction events. You provide SINGLE_WRITE_EVENT if you want to configure the subscription stream to split apart a transaction event and deliver discrete write operations for every write that was committed in the transaction. The SINGLE_WRITE_EVENT is considered as the default mode if you don't invoke the `setStreamDeliveryMode()` method.
- The table name(s) argument must include tables that exist in the data store. You must first subscribe the tables using the `setSubscribedTables()` method before invoking the `setStreamDeliveryMode()` method. Otherwise, the subscription will fail with the `SubscriptionFailureException` exception.

> **Note**
>
> In case of hierarchical tables, you must add the root table to the subscription to stream transaction events. All the child tables in the same hierarchy also stream transaction events. For example, if there are updates to a parent table and a child table in a transaction, the Streams API encapsulates all the committed writes in this transaction as a single transaction event and streams it to the subscriber. If you don't configure transaction event streaming, each committed write to either a parent or a child table is streamed as a single stream event.

You can also configure the subscription to include the following while streaming transaction events.

- Mixed mode streaming where the streams deliver transaction events for one subscribed table and a discrete list of committed write events for another subscribed table. In the following code sample, you configure the subscription to the `UserTable` table to stream a discrete list of committed write events and the `UserTableTxn` table to deliver transaction events.

```
/**
 * create subscription to stream UserTable and UserTableTxn tables
 * UserTable table streams discrete list of committed writes
 * UserTableTxn table streams transaction events
 */

import static
oracle.kv.pubsub.NoSQLSubscriptionConfig.StreamDeliveryMode.GROUP_ALL_IN_TR
ANSACTION;
...
final NoSQLSubscriptionConfig subConfig =
    new NoSQLSubscriptionConfig.Builder("ChkptTable")
    .setSubscribedTables("UserTable", "UserTableTxn")
    .setStreamMode(NoSQLStreamMode.FROM_NOW)
    .setStreamDeliveryMode(GROUP_ALL_IN_TRANSACTION, "UserTableTxn")
    .build();
```

- Add a new subscribed table to a running subscription and stream its transaction events. Here, you can subscribe to another table and stream its transaction events while you have an existing subscription streaming a list of committed write events for the original table. In the following code sample, you first configure a subscription to stream individual write events for the `UserTable` table and then subscribe table `UserTableTxn` to the same subscription to stream transaction events for that table.

```
/* create subscription to stream UserTable */
final NoSQLSubscriptionConfig subConfig =
    new NoSQLSubscriptionConfig.Builder("ChkptTable")
    .setSubscribedTables("UserTable")
    .setStreamMode(NoSQLStreamMode.FROM_NOW)
    .build();

/* stream all writes */
    ...

/* subscribe UserTableTxn table to stream transaction events */
```

```
subscriber.clearChangeResultFlag();
stream.subscribeTable("UserTableTxn", true);

...
```

> **Note**
>
> Adding a table to the subscription at run time to stream its write operations does not impact existing subscriptions and streaming.

NoSQLStreamMode

`NoSQLStreamMode` is the subscription stream mode used to configure the starting point for a NoSQL subscription.

Once you have taken a checkpoint, you indicate where you want event consumption to begin by specifying a `NoSQLStreamMode` to `NoSQLSubscriptionConfig.Builder.setStreamMode()`. For example, if you specify `NoSQLStreamMode.FROM_EXACT_CHECKPOINT`, then events will begin at the stream position identified by the checkpoint saved in the checkpoint table.

The stream positions available to you are:

- `FROM_CHECKPOINT`

  Starts the stream from the last checkpoint saved in the checkpoint table, using the next available position for shards where the checkpoint position is not available.

- `FROM_EXACT_CHECKPOINT`

  Starts the stream from the last checkpoint saved in the checkpoint table, signaling an exception if the checkpoint position is not available.

- `FROM_EXACT_STREAM_POSITION`

  Starts the stream from the specified start stream position, signaling an exception if the requested position is not available.

- `FROM_NOW`

  Starts the stream from the latest available stream position.

- `FROM_STREAM_POSITION`

  Starts the stream from the specified start stream position, using the next available position for shards where the requested position is not available.

See NoSQLStreamMode in the *Java Direct Driver API Reference*.

Using NoSQLSubscription

`oracle.kv.pubsub.NoSQLSubscription` is used to control your subscription. It is used to request operations from the subscribed tables, to perform checkpoints, terminate the stream, and so forth. It is used as a part of your `NoSQLSubscriber` implementation.

The `NoSQLSubscription` interface extends `org.reactivestreams.Subscription`, so it is sufficient for your `NoSQLSubscription` implementation class to extend `NoSQLSubscription`. When your implementation class implements `NoSQLSubscriber.onSubscribe()`, you will

usually call `NoSQLSubscription.request()`, which asks for an initial set number of events to be delivered to the subscriber (these are consumed using `NoSQLSubscriber.onNext()`):

```
...
private NoSQLSubscription subscription;
...

@Override
public void onSubscribe(Subscription s) {
    subscription = (NoSQLSubscription) s;
    // request 100 store operations be streamed to this
    // subscriber.
    s.request(100);
    }
```

The important actions that you can take with your `NoSQLSubscription` object are:

- Cancel the stream using `NoSQLSubscription.cancel()`.
- Request more operations from the subscribed table.

  If you want to stream infinite number of operations, you can use `Long.MAX_VALUE`, which allows to stream for 584 years, assuming that the subscriber can process 1 billion operations per second.

  If the request is made at the beginning of the application's runtime before any operations have been consumed, then the operations will begin from the location identified by `NoSQLSubscriptionConfig.setStreamMode()`. If the request is made after operations have been consumed, then the operations will begin at the point in the stream immediately after the last consumed operation. For more information, see NoSQLStreamMode.
- Take a checkpoint. See Using Checkpoints for information about checkpoints.
- Get a list of currently subscribed tables anytime during the lifetime of the stream using `NoSQLSubscription.getSubscribedTables()` method. `SubscriptionFailureException` would be raised if the subscription is canceled or has shut down.
- Add a table to the running subscription stream asynchronously using `NoSQLSubscription.subscribeTable()` method. The change result will be signaled via the callback `NoSQLSubscriber.onChangeResult()` method.
- Remove a table from the running subscription stream asynchronously using `NoSQLSubscription.unsubscribeTable()` method. The change result will be signaled via the callback `NoSQLSubscriber.onChangeResult()` method.

For a complete list of operations supported by `NoSQLSubscription`, see NoSQLSubscription in the *Java Direct Driver API Reference*.

4

Implementing Subscribers

For every publisher, you must implement one or more Subscriber. The Subscriber is used to process stream events, which arrive in the form of `StreamOperation` class objects. See Using the StreamOperation Class.

Using the NoSQLSubscriber Interface

You implement subscribers using the `oracle.kv.pubsub.NoSQLSubscriber` interface, an extension of `org.reactivestreams.Subscriber`. `NoSQLSubscriber` provides the following methods, which you must implement:

- `onSubscribe()`

  This is the method invoked after the publisher has successfully established contact with the Oracle NoSQL Database store. The argument you pass to this method is an `org.reactivestreams.Subscription` instance, which you can then cast to `oracle.kv.pubsub.NoSQLSubscription`. See Working with Subscriptions.

  ```
  ...
  private NoSQLSubscription subscription;
  ...

  @Override
  public void onSubscribe(Subscription s) {
      subscription = (NoSQLSubscription) s;
      // request 100 store operations be streamed to this
      // subscriber.
      s.request(100);
  }
  ```

  > **Note**
  >
  > You do not have to call the `s.request(100)`method inside `onSubscribe()`. Once an instance of `NoSQLSubscription` is available, you can call the method outside `onSubscribe()`. The main point of the `onSubscribe()` method here is to pass the user the subscription instance that the publisher generates.

- `onNext()`

  Signals the next Oracle NoSQL Database operation. You pass this method a `StreamOperation` class instance. See Using the StreamOperation Class. This method is where you perform whatever processing you want to perform on the stream events.

  ```
  @Override
  public void onNext(StreamOperation t) {
      // perform processing on the StreamOperation
      // here. Typically you will do different
  ```

```
        // things depending on whether this is
        // a put,delete, or transaction event.
        switch (t.getType()) {
            case PUT:
                {
                    // Process the put operation here.
                }
                break;
            case DELETE:
                {
                    // Process the delete operation here.
                }
                break;
            case TRANSACTION:
                {
                    // Process the transaction operation here.
                }
                break;
            default:
                // Received an unknown and therefore illegal operation
                // type.
                throw new
                IllegalStateException("... exception message ...");
        }
    }
    }
```

If row operations in the change stream include user-defined row metadata, this metadata is made available in the corresponding `StreamOperation` object received by `onNext()`. Subscribers can access this user-defined information attached to each row operation alongside the core data. This applies to all row changes performed through different APIs. For more information on user-defined row metadata, see Using user-defined row metadata in *Developers Guide*.

- `onComplete()`

  Signals the completion of a subscription. Use this method to perform whatever cleanup your application requires once a subscription has ended.

  ```
  @Override
  public void onComplete() {
      /* nothing to do, so make this a no-op */
  }
  ```

  > **Note**
  >
  > You must implement this method in your stream processing application, because streaming from a KVStore table is unbounded by nature, so `onComplete()` will never be called. Any `no-op` implementation of this method will be ignored.

- `onError()`

Signals that the subscription encountered an irrecoverable error and has to be terminated. The argument passed to this method is a `java.lang.Throwable` class instance. Use this method to perform whatever actions you want to take in response to the error.

```
@Override
public void onError(Throwable t) {
    logger.severe("Error: " + t.getMessage());
}
```

- `onWarn()`

  Signals that the subscription encountered a potential issue in the service. The argument passed to this method is a `java.lang.Throwable` class instance. Use this method to perform whatever actions you want to take in response to the warning.

  ```
  @Override
  public void onWarn(Throwable t) {
      logger.warning("Warning: " + t.getMessage());
  }
  ```

  A warning does not end the subscription. Warnings in the form of `ShardTimeoutException` are provided as a way to inform the application of non-disruptive service issues such as a particular shard not responding for an extended period of time.

- `onCheckPointComplete()`

  Signals when a previously requested checkpoint has been completed. Checkpoints are performed by calling `NoSQLSubscription.doCheckpoint()`. Note that if an error occurred, the subscription will lose the checkpoint but the subscription itself will not terminate, and will continue streaming. See Using Checkpoints.

  Call this method with two arguments:

  – `oracle.kv.pubsub.StreamPosition`

    Identifies the location in the stream where the checkpoint was performed.

  – `java.lang.Throwable`

    `Null`, unless an error occurred while taking the checkpoint.

  ```
  @Override
  public void onCheckpointComplete(StreamPosition pos,
                                   Throwable cause) {
      if (cause == null) {
          logger.info("Finish checkpoint at position " + pos);
      } else {
          logger.warning("Fail to checkpoint at position " + pos +
                         ", cause: " + cause.getMessage());
      }
  }
  ```

- `onChangeResult()`

  Adding and removing tables from running subscription streams are made using asynchronous calls. The asynchronous calls will return immediately without any return value. The result of the operation can be fetched using the `onChangeResult` callback method after the change is effective. If the change was successful, this method will be called with a non-null stream position that represents the first stream position for which the

change has taken effect. If the change was unsuccessful, but the subscription is still active, this method will be called with a non-null exception that describes the cause of the failure. If the change caused the subscription to be canceled, this method will not be called, and the `onError` method will be called instead.

- `getSubscriptionConfig()`

  Use this method to return the `oracle.kv.pubsub.NoSQLSubscriptionConfig` object used by this subscription. This method is invoked by the publisher when it is creating a subscription.

NoSQLSubscriber Example

This section provides a complete, but simple, `NoSQLSubscriber` example called `GSGSubscriberExample`. This implementation is used by the publisher example shown in Streams Example.

`GSGSubscriberExample` subscribes to a single table called `Users`. To see the application that defines this table and writes table rows to it, see GSGStreamsWriteTable.

To begin, we provide our imports. Notice that `org.reactivestreams.Subscription` is a required import. Your Java environment must have the `reactive-streams.jar` file in its classpath in order to both compile and run this example code.

```
package pubsub;

import java.util.List;

import oracle.kv.pubsub.NoSQLSubscriber;
import oracle.kv.pubsub.NoSQLSubscription;
import oracle.kv.pubsub.NoSQLSubscriptionConfig;
import oracle.kv.pubsub.StreamOperation;
import oracle.kv.pubsub.StreamPosition;

import oracle.kv.table.MapValue;
import oracle.kv.table.Row;

import org.reactivestreams.Subscription;
```

Next we declare our class, and initialize our data members. As described previously, this is an implementation of `NoSQLSubscriber`.

```
class GSGSubscriberExample implements NoSQLSubscriber {

    /* subscription configuration */
    private final NoSQLSubscriptionConfig config;

    /* number of operations to stream */
    private final int numOps;

    /* number of operations seen in the stream */
    private long streamOps;

    private NoSQLSubscription subscription;

    private boolean isSubscribeSucc;
```

```
    private Throwable causeOfFailure;

    GSGSubscriberExample(NoSQLSubscriptionConfig config,
                                 int numOps) {
        this.config = config;
        this.numOps = numOps;

        causeOfFailure = null;
        isSubscribeSucc = false;
        streamOps = 0;
        subscription = null;
    }
```

The first thing we do is implement `NoSQLSubscriber.getSubscriptionConfig()`. This simply returns our `NoSQLSubscriptionConfig` object, which is provided to the class when it is constructed by the implementing streams application. This method is how the publisher will learn how to configure the stream for this subscriber.

```
    @Override
    public NoSQLSubscriptionConfig getSubscriptionConfig() {
        return config;
    }
```

The implementation we provide for `onSubscribe()` does several things. First, it makes the `NoSQLSubscription` class instance available to this subscriber implementation. Notice that the instance is passed to this class as an object of type `org.reactivestreams.Subscription`, and that object must be cast to `NoSQLSubscription`.

This method is also where this subscriber begins requesting operations from the subscription. Without that call to `NoSQLSubscription.request()`, this subscriber will never receive any operations to process. For this simple implementation, this is the only place operations are requested. In a more elaborate implementation, operations are initially asked for here, and once that number of operations have been received by the subscriber, more can be asked for in another part of the class — usually in `onNext()`.

Finally, we signal that the subscription attempt is a success. This information is used by our streams application when we are creating the publisher and subscriber.

```
    @Override
    public void onSubscribe(Subscription s) {
        subscription = (NoSQLSubscription) s;
        subscription.request(numOps);
        isSubscribeSucc = true;
    }
```

Next we set up our Error and Warning handlers. Note that, when `onError` is called, the subscription has already been canceled. Here, we do the simple thing and simply write to the console. However, a more robust implementation would write to the application log file, and potentially take other notifications and/or corrective actions (such as quit processing the stream entirely), depending on the nature of the error.

```
    @Override
    public void onError(Throwable t) {
```

```
        causeOfFailure = t;
        System.out.println("Error: " + t.getMessage());
    }

    @Override
    public void onWarn(Throwable t) {
        System.out.println("Warning: " + t.getMessage());
    }
```

The application has to provide an `onComplete()` method, although the implementation is not required to do anything, since this method is not called.

```
    @Override
    public void onComplete() {
        /* no-op */
    }
```

Because this example does not implement checkpoints (see Using Checkpoints for more information), there is nothing to do in this method.

```
    /* called when publisher finishes a checkpoint */
    @Override
    public void onCheckpointComplete(StreamPosition pos,
                                     Throwable cause) {
        /* no-op. This example doesn't implement checkpoints */
    }
```

The `onNext()` method is where the subscriber receives and processes individual stream operations in the form of `StreamOperation` objects.

In the following method, you will understand how to determine what type of operation the subscription has received (either `put`, `delete`, or `transaction`). What you would do with an individual operation is up to your application's requirements. In this case, for `put` operations we retrieve field information from the enclosed `Row` object, and write it to the console. Be aware that this code is not very robust. In particular, we expect JSON data with a specific schema. Because any valid JSON can be written to a JSON table column, some defensive code is required here for a production application to ensure that the JSON column contains the expected schema.

If you have optionally configured before-images, you can retrieve before-images and related field information by using the methods provided by the `StreamOperation` interface. For a code sample, see NoSQLSubscriber Example with Before-images Streaming.

For `delete` operations, the `StreamOperation` object is written to stdout.

If you have optionally configured transaction event, you must handle the transaction operation in the `onNext()` method by checking the type of the operation. For a code sample, see NoSQLSubscriber Example with Transaction Event Streaming.

```
    @Override
    public void onNext(StreamOperation t) {

        switch (t.getType()) {
            case PUT:
```

```
                streamOps++;
                System.out.println("\nFound a put. Row is:");

                StreamOperation.PutEvent pe = t.asPut();
                Row row = pe.getRow();

                Integer uid = row.get("uid").asInteger().get();
                System.out.println("UID: " + uid);

                MapValue myjson = row.get("myJSON").asMap();
                int quantity = myjson.get("quantity")
                                     .asInteger().get();
                String array =
                    myjson.get("myArray").asArray().toString();
                System.out.println("\tQuantity: " + quantity);
                System.out.println("\tmyArray: " + array);
                break;
            case DELETE:
                streamOps++;
                System.out.println("\nFound a delete. Row is:");
                System.out.println(t);
                break;

            default:
                throw new
                    IllegalStateException("Receive unsupported " +
                        "stream operation from shard " +
                                t.getRepGroupId() +
                                ", seq: " + t.getSequenceId());
        }
        if (streamOps == numOps) {
            getSubscription().cancel();
            System.out.println("Subscription cancelled after " +
                    "receiving " + numOps + " operations.");
        }
    }
```

Finally, we provide a series of getter methods, which are used by our stream application to retrieve information of interest from this subscriber. Using a Streams Publisher shows how these are used.

```
    String getCauseOfFailure() {
        if (causeOfFailure == null) {
            return "success";
        }
        return causeOfFailure.getMessage();
    }

    boolean isSubscriptionSucc() {
        return isSubscribeSucc;
    }

    long getStreamOps() {
        return streamOps;
    }
```

```
    NoSQLSubscription getSubscription() {
        return subscription;
    }

}
```

NoSQLSubscriber Example with Before-images Streaming

Learn to use the Oracle NoSQL Database Streams API to retrieve before-images and related field information.

If before-images are enabled, you can retrieve before-images and related information and print it to stdout as shown in the following code sample. To successfully implement your operation to display before-images, you must check if before-images are enabled on the table of interest, your subscription is configured to stream before-images, and if before-images are within their expiry dates.

The row objects include the state of the row after and before the operation.

```
    @Override
    public void onNext(StreamOperation t) {

        switch (t.getType()) {
            case PUT:
                streamOps++;
                System.out.println("\nFound a put. Row is:");
                StreamOperation.PutEvent pe = t.asPut();
                Row rowAfter = pe.getRow();
                // Print after-image
                System.out.println("\nAfter-images streaming \n" +
rowAfter.toJsonString(true));

                System.out.println("\nBefore-images enabled = " +
t.isBeforeImageEnabled());
                System.out.println("\nBefore-images subscribed = " +
t.includeBeforeImage());
               // Check if before-images is enabled, subscribed, and within
the expiry date
                if (t.isBeforeImageEnabled() && t.includeBeforeImage() && (!
t.isBeforeImageExpired())) {
                    Row rowBefore = t.getBeforeImage();
                    System.out.println("\nBefore-images streaming \n" +
rowBefore.toJsonString(true));
                }
                else
                    System.out.println("Before-images is not streamed");
                break;

            case DELETE:
                streamOps++;
                System.out.println("\nFound a delete. Row is:");
                System.out.println(t);
                break;

            default:
```

```
                    throw new
                        IllegalStateException("Receive unsupported " +
                            "stream operation from shard " +
                                    t.getRepGroupId() +
                                    ", seq: " + t.getSequenceId());
                }
                if (streamOps == numOps) {
                    getSubscription().cancel();
                    System.out.println("Subscription cancelled after " +
                            "receiving " + numOps + " operations.");
                }
            }
```

NoSQLSubscriber Example with Transaction Event Streaming

Learn to use the Oracle NoSQL Database Streams API to retrieve transaction event related information.

If streaming transaction events is enabled, you can retrieve all the committed writes and related information for a transaction event as shown in the following code sample. To successfully implement your code to display the transaction events, you must ensure your subscription is configured to stream transaction events. For details, see Configuring Transaction Event Streaming.

The `TransactionEvent` object returns the transaction id, transaction type, number of writes in the transaction, and the stream of committed writes for the transaction operation.

```
    @Override
    public void onNext(StreamOperation t) {
        switch (t.getType()) {
            case PUT:
                streamOps++;
                System.out.println("\nFound a put. Row is:");
                StreamOperation.PutEvent pe = t.asPut();
                Row row = pe.getRow();
                break;

            case DELETE:
                streamOps++;
                System.out.println("\nFound a delete. Row is:");
                System.out.println(t);
                break;

            case TRANSACTION:
                streamOps++;
                System.out.println("\nFound a transaction. Details are:");
                StreamOperation.TransactionEvent te = t.asTransaction();
                TransactionIdImpl TxnID = te.getTransactionId();
                List<StreamOperation> TxnOps = te.getOperations();
                System.out.println("\ntransactionID="  + TxnID.toString() +
"\ntransactionType" + te.getTransactionType() + "\nTotalWrites" +
te.getNumOperations());
                System.out.println("\nPrinting writes in the transaction:" );
                int opsNum = 1;
                for (ListIterator<StreamOperation> iterator =
TxnOps.listIterator(); iterator.hasNext(); ) {
```

```
                    System.out.println("\nWrite operation ["+ opsNum +"]: " +
iterator.next());

                    opsNum++;
                    }
                    break;



                default:
                    throw new
                        IllegalStateException("Receive unsupported " +
                            "stream operation from shard " +
                                    t.getRepGroupId() +
                                    ", seq: " + t.getSequenceId());
            }
```

5

Using a Streams Publisher

Each shard in the store publishes changes made to table data in the shard. Each of these publishing streams is combined into a single stream of table write operations, which the `oracle.kv.pubsub.NoSQLPublisher` class can access. This class constructs one or more `NoSQLSubscription` class objects, each of which can be used to manage a single subscription stream (where each subscription stream can include changes made to one or more tables in the store).

You configure `NoSQLPublisher` using `oracle.kv.pubsub.NoSQLPublisherConfig`, described next in this chapter. See Streams Example to see how the configuration is used within a streams application.

For any JVM, only one `NoSQLPublisher` instance can be created given identical `NoSQLPublisherConfig` objects. After creating an instance of `NoSQLPublisher`, the factory constructor returns the same instance of `NoSQLPublisher` for all subsequent construction requests, if the `NoSQLPublisherConfig` for those requests is identical.

Two `NoSQLPublisherConfig` objects are identical if all of the following information the same for both objects:

- Store name
- Shard timeout
- Maximum concurrent allowed subscriptions
- Security properties use identical username and credentials

Using NoSQLPublisherConfig

Use `oracle.kv.pubsub.NoSQLPublisherConfig` to specify connection and authentication information to the store. You can also use this class to configure performance parameters.

Configuring a Connection to the Store

When you construct a `NoSQLPublisherConfig` object, you provide it with an `oracle.kv.KVStoreConfig` object. This object is used to provide store connection information:

- The name of the store that your publisher is monitoring
- A list of one or more helper host port pairs. These helper hosts are Storage Nodes in the store. They must be resolvable using either DNS or the local `/etc/hosts` file.

For example:

```
package pubsub;

import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

...
```

```
String[] hhosts = {"n1.example.org:5088", "n2.example.org:4129"};
KVStoreConfig kconfig = new KVStoreConfig("exampleStore", hhosts);
```

This simple example is sufficient to connect to a store that is not configured for authentication. For information about connecting to a secure store, see Authenticating to a Secure Store.

Creating a Basic NoSQLPublisherConfig Object

You use `NoSQLPublisherConfig.Builder` to construct a `NoSQLPublisherConfig` object. The constructor for this class requires you to provide a `KVStoreConfig` object, as well as a path to the publisher's root directory (this directory is used to contain files necessary for the publisher's proper operation).

```
...

// Create a minimal KVStoreConfig
String[] hhosts = {"n1.example.org:5088", "n2.example.org:4129"};
KVStoreConfig kconfig = new KVStoreConfig("exampleStore", hhosts);

final NoSQLPublisherConfig publisherConfig =
    new NoSQLPublisherConfig.Builder(kconfig, "/export/publisher")
    .build();
```

Once you have created the `NoSQLPublisherConfig` object, you can use it in a call to the `NoSQLPublisher.get()` method to obtain a `NoSQLPublisher` instance and connect to the store. See, Authenticating to a Secure Store for an example of this.

Tuning Your Publisher

When you construct a `NoSQLPublisherConfig` object, you can specify several tuning controls:

- Maximum concurrent subscriptions

  Specifies the maximum number of subscribers that this publisher can run. This must be set to at least 1.

  Use `NoSQLPublisherConfig.setMaxConcurrentSubs()` to configure this value. Default is 32.

- A shard timeout value. If the publisher does not hear from a source shard in the amount of time specified here, the publisher will throw `ShardTimeoutException` via a call to `NoSQLSubscriber.onWarn`. If a `ShardTimeoutException` is thrown, the stream and the connection to the shard still remain alive, just that there is no operation received from that shard within the timeout period.

  Use `NoSQLPublisher.setShardTimeoutMs()` to configure this value. This method takes a `long` that represents the timeout value in milliseconds. Default is 600000 ms (10 minutes).

For example:

```
...

// Create a minimal KVStoreConfig
String[] hhosts = {"n1.example.org:5088", "n2.example.org:4129"};
KVStoreConfig kconfig = new KVStoreConfig("exampleStore", hhosts);
```

```
final NoSQLPublisherConfig publisherConfig =
    new NoSQLPublisherConfig.Builder(kconfig, "/export/publisher")
    .setMaxConcurrentSubs(2)
    .setShardTimeoutMs(10000)
    .build();
```

Authenticating to a Secure Store

To authenticate to a secure store, you must provide login credentials. The simplest way to connect your stream processing application to secure store is by specifying a value for the `oracle.kv.security` system property, which incudes the pathname of a file containing the security property settings generated while setting up a user login for a secure store. For more information about setting up a secure store to generate the security property file, see Performing a Secure Oracle NoSQL Database Installation in the *Security Guide*.

Note that if you choose to follow the method above, you do not need to modify your application code. To run the example to connect to secure store, use the command below:

```
java -Doracle.kv.security=mylogin
-cp $KVHOME/lib/kvstore.jar:. pubsub.NoSQLStreamExample
```

Reauthentication

Once the publisher has created an initial authenticated connection to the store, the authentication credentials are lost; they are not kept in memory or in any way cached.

After the initial connection, every subscription also has to be authenticated. This authentication process ensures that the subscriber has the appropriate read access to the table(s) for which a subscription is being obtained. If the user is attempting to subscribe to a single table or a small set of tables, she needs `READ_TABLE` access for each table. If the user wants to subscribe to any table in the store, then for convenience that user account can be configured with `READ_ANY_TABLE` access.

To allow subscriptions to authenticate, you implement a `ReauthenticationHandler` class and then provide it to your `NoSQLPublisherConfig` object using the `NoSQLPublisherConfig.setReauthHandler()` method.

The following example extends the authentication example shown in the previous section to add a reauthentication handler.

First, you must implement `ReauthenticationHandler`. The following is an example of a very simple implementation:

```
package pubsub;

import oracle.kv.ReauthenticationHandler;
import oracle.kv.PasswordCredentials;

public class MyReauthHandler implements ReauthenticationHandler {
    public void reauthenticate(KVStore reauthStore) {

    // The code you use to obtain the username and password strings
    // should be consistent with the code you use to perform
    // simple authentication for your publisher. Here we do
```

```
    // the simplest -- and least secure -- thing possible.

    // This is really not what you should do for production
    // code.

    final String username = "beth";
    final String password = "my_clever_password00A";

    PasswordCredentials cred = new PasswordCredentials(username,
    password.toCharArray());

    reauthStore.login(cred);
```

We then extend the previous authentication example to use our implemented `ReauthenticationHandler`. We do this with a single line of code, which is in **bold** in the example.

```
package pubsub;

...

// Create a KVStoreConfig object that is configured
// for a secure store.
String[] hhosts = {"n1.example.org:5088", "n2.example.org:4129"};
KVStoreConfig kconfig = new KVStoreConfig("exampleStore", hhosts);

// Need to set some required security properties.
Properties secProps = new Properties();
secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                     KVSecurityConstants.SSL_TRANSPORT_NAME);

// The client.trust file is created when you install your
// store. It must be moved locally to every machine where
// client code will run.
secProps.setProperty
    (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
    "/home/kv/client.trust");
kconfig.setSecurityProperties(secProps);

// Create a PasswordCredentials instance. We hard-code
// the credentials here, but in a production environment
// this information should be provided in a significantly
// more secure way.

// username and password must have been configured for the store
// by its administrator.

final String username = "beth";
final String password = "my_clever_password00A";

PasswordCredentials pc =
    new PasswordCredentials(username,
                            password.toCharArray());

// Create the publisher's configuration object.
```

```
// Keeping it simple.
final NoSQLPublisherConfig publisherConfig =
    new NoSQLPublisherConfig.Builder(kconfig, "/export/publisher")
 .setReauthHandler(new MyReauthHandler())
    .build();

// Now connect to the store
try {
    NoSQLPublisher publisher =
        NoSQLPublisher.get(publisherConfig, pc);
} catch (PublisherFailureException pfe) {
        System.out.println("Connection or authentication failed.");
        System.out.println(pfe.toString());
}
```

Streams Example

This section presents an example of a streams application. While this example is simplified as much as possible, its broad outline is typical for applications of this nature.

This example application makes use of the example Subscriber that we described in NoSQLSubscriber Example.

This application begins by defining information required by the application. It indicates what table the application will watch — a single subscriber can receive operations from multiple tables, but for this example we will only subscribe to the table `Users`. Also, `num` is the number of operations the subscriber will request for the `Users` table.

> **Note**
>
> Stream operations support namespaces. If you want to subscribe to a table in a namespace, prefix the table name with a namespace and a colon (:), as follows: *ns1*`:Users`.

We then provide Oracle NoSQL Database connection information. Because this is a simple example that exists purely for illustration purposes, we avoid authentication issues by using a non-secure store. However, in a production environment you will probably be required to provide authentication credentials as described in Authenticating to a Secure Store and Reauthentication.

Finally, we provide some information that is specific to a streaming application. `MAX_SUBSCRIPTION_TIME_MS` is used to identify how long the application can wait before it times out. `CKPT_TABLE_NAME` is the name of the checkpoint table. This information is required when constructing a `NoSQLPublisher`, but is not otherwise used by this particular application. For information about checkpoints, see Using Checkpoints.

```
package pubsub;

import oracle.kv.KVStoreConfig;
import oracle.kv.pubsub.NoSQLPublisher;
import oracle.kv.pubsub.NoSQLPublisherConfig;
import oracle.kv.pubsub.NoSQLSubscriptionConfig;

public class GSGStreamExample {
```

```
/* table to subscribe */
private static final String TABLE_NAME = "Users";
/* Number of operations to stream */
private static final int num = 100;

private static final String storeName = "kvstore";
private static final String[] hhosts = {"localhost:5000"};

/* max subscription allowed time before forced termination */
private static final long MAX_SUBSCRIPTION_TIME_MS =
    Long.MAX_VALUE;

private static final String rootPath = ".";
private static final String CKPT_TABLE_NAME = "CheckpointTable";

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

    final GSGStreamExample gte = new GSGStreamExample(args);
    gte.run();
}

private GSGStreamExample(final String[] argv) {
}
```

First we construct a `NoSQLPublisher` object. `NoSQLPublisherConfig` is used to specify the Oracle NoSQL Database connection information.

```
/*
 * Subscribes a table. The work flow is ReactiveStream
 * compatible
 */
private void run() throws Exception {

    NoSQLPublisher publisher = null;
    try {
        /* step 1 : create a publisher configuration */
        final NoSQLPublisherConfig publisherConfig =
            new NoSQLPublisherConfig.Builder(
                new KVStoreConfig(storeName, hhosts), rootPath)
                .build();

        /* step 2 : create a publisher */
        publisher = NoSQLPublisher.get(publisherConfig);
```

Next we construct a `NoSQLSubscriptionConfig`. This is where we identify the table(s) to which we are subscribing.

```
        /* step 3: create a subscription configuration */
        final NoSQLSubscriptionConfig subscriptionConfig =
            /* stream with specified mode */
            new NoSQLSubscriptionConfig.Builder(CKPT_TABLE_NAME)
            .setSubscribedTables(TABLE_NAME)
            .build();
```

Now we construct our subscriber. Here, we use the `NoSQLSubscriber` implementation that we describe in NoSQLSubscriber Example.

```
/* step 4: create a subscriber */
final GSGSubscriberExample subscriber =
    new GSGSubscriberExample(subscriptionConfig, num);
System.out.println("Subscriber created to stream " +
        num + " operations.");
```

The above example specifies the number of events to be streamed as 100. However, if you want to do continuous streaming, you must use new `GSGSubscriberExample(subscriptionConfig, Long.MAX_VALUE)`.

Next you create a subscription. If `GSGSubscriberExample` reports an error on creating the subscription (using the `getSubscriptionSucc()` method), we throw an Exception and quit the application with a message that identifies the nature of the error (`getCauseOfFailure()`) and the subscriber's unique ID (`getSubscriberId()`).

```
/* step 5: create a subscription and start stream */
publisher.subscribe(subscriber);
if (!subscriber.isSubscriptionSucc()) {
    System.out.println("Subscription failed for " +
                  subscriber.getSubscriptionConfig()
                            .getSubscriberId() +
                  ", reason " +
                  subscriber.getCauseOfFailure());

    throw new RuntimeException("fail to subscribe");
}
System.out.println("Start stream " + num +
        " operations from table " + TABLE_NAME);
```

At this point we put the application thread to sleep, which allows the subscriber to run unimpeded by the parent application. Occasionally we allow this thread to wake up, check how many stream operations have been consumed by the subscriber, and make sure we have not exceeded our maximum amount of run time. If we have exceeded our timeout threshold, we throw an exception and quit the application. Otherwise, we continue to run until all of the required operations have been consumed.

```
/*
 * Wait for the stream to finish. Throw exception if it
 * cannot finish within max allowed elapsed time
 */
final long s = System.currentTimeMillis();
while (subscriber.getStreamOps() < num) {
    final long elapsed = System.currentTimeMillis() - s;
    if (elapsed >= MAX_SUBSCRIPTION_TIME_MS) {
        throw new
            RuntimeException("Not done within max " +
                    "allowed elapsed time");
    }
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        throw new RuntimeException("Interrupted!");
```

```
            }
        }
```

Finally, we clean up and close the application. Note that we could cancel the subscription at this point using `subscriber.getSubscription.cancel()`, but our `GSGSubscriberExample` class is already calling that in its `onNext()` method. For a more robust application, you could call `cancel()` from the stream application itself, particularly as a part of responding to error situations.

```
            /* step 6: clean up */
            subscriber.getSubscription().cancel();
            publisher.close(true);
            System.out.println("Publisher closed normally.");

        } catch (Exception exp) {
            String msg = "Error: " + exp.getMessage();
            System.out.println(msg);
            if (publisher != null) {
                publisher.close(exp, false);
                System.out.println("Publisher closed with error.");
            }
            throw exp;
        } finally {
           System.out.println("All done.");
        }
    }
}
```

Sample Streams Output

Once the `Users` table is loaded with sample data (see GSGStreamsWriteTable), the output from this example program is as follows (the output is truncated at the 12th operation for brevity):

```
> java pubsub.GSGStreamExample
Subscriber created to stream 100 operations.
Start stream 100 operations from table Users

Found a put. Row is:
UID: 0
    Quantity: 10
    myArray: [1,14,3,9,12,12,13,13,4,6]

Found a put. Row is:
UID: 1
    Quantity: 4
    myArray: [3,14,1,13]

Found a put. Row is:
UID: 2
    Quantity: 5
    myArray: [5,7,15,1,5]

Found a put. Row is:
```

```
UID: 3
    Quantity: 2
    myArray: [10,7]

Found a put. Row is:
UID: 4
    Quantity: 7
    myArray: [2,17,5,9,1,10,5]

Found a put. Row is:
UID: 5
    Quantity: 5
    myArray: [13,1,2,3,11]

Found a delete. Deleted row is:
Del OP [seq: 6304, shard id: 1, primary key: {
  "uid" : 2
}]

Found a put. Row is:
UID: 6
    Quantity: 9
    myArray: [16,7,11,13,13,10,11,15,5]

Found a put. Row is:
UID: 7
    Quantity: 2
    myArray: [11,3]

Found a put. Row is:
UID: 8
    Quantity: 6
    myArray: [12,12,5,11,11,3]

Found a put. Row is:
UID: 9
    Quantity: 4
    myArray: [10,7,6,4]

Found a put. Row is:
UID: 10
    Quantity: 8
    myArray: [3,9,18,11,16,12,6,2]

    ...
```

Every time this example is run, it always starts streaming from the first table operation seen for the `Users` table; that is, for the write operation that created the first row in the table (UID 0). Instead of streaming from the beginning every time, if you want to stream from the Nth operation, you need to implement checkpoints. These are described in the next chapter, Using Checkpoints.

6
Using Checkpoints

When a subscriber opens a subscription stream, it starts consuming events from the earliest available point in the stream, unless you specify a different start point. To begin consuming from another location, your application must have run and saved a checkpoint that represents a stream location. Use `NoSQLSubscription.getCurrentPosition()` to obtain the current stream position. This method returns as `StreamPosition` class. Use `NoSQLSubscription.doCheckpoint()` to run the actual checkpoint.

Running a checkpoint causes the current stream position to be saved in the store using the checkpoint table you identified when you configured your `NOSQLSubscription` instance. Only the latest checkpoint is saved to this table. If you want to save other checkpoints, you can serialize the `StreamPosition` class representing a checkpoint, and save it to disk or a database of your choice.

> **Note**
>
> - You are responsible for choosing a name for the checkpoint table. Be sure that the name is unique to your subscription. If you are using multiple subscriptions, make sure that each subscription has a unique name for its checkpoint table.
> - Checkpoint tables are used to store checkpoint-related information. Do not delete or change the table structure without consideration. If you delete the table, you lose the checkpoint for this subscription. If the subscription continues after its checkpoint table is deleted, at the next checkpoint, the subscriber will be unable to locate the expected checkpoint and will skip a checkpoint. The method `onCheckpointComplete()` captures the `CheckpointFailureException` error message.
>
>   If you cancel the current subscription and re-create a new one, the new subscription will create the checkpoint for you at the beginning, as long as it has the privilege to do so.

The method `NoSQLSubscription.doCheckpoint()` runs asynchronously, so the call returns after the checkpoint is requested, and `NoSQLSubscriber.onCheckpointComplete` is called when the checkpoint is complete. The`CheckpointFailureException` is raised if you call this method while there is another outstanding request for a checkpoint running for the same subscription.

Implementing Checkpoints in GSGStreamExample

This section shows how to implement checkpoints by adding the functionality to the examples provided in Streams Example. You must also add functionality to the Subscriber implementation shown in NoSQLSubscriber Example. For those updates, see the next section, Implementing Checkpoints in GSGSubscriberExample.

New additions to the original example code are indicated by **bold** text.

The changes to `GSGStreamsExample.java` are fairly minor. To begin, we need to import `NoSQLStreamMode`:

```
package pubsub;

import oracle.kv.KVStoreConfig;
import oracle.kv.pubsub.NoSQLPublisher;
import oracle.kv.pubsub.NoSQLPublisherConfig;
import oracle.kv.pubsub.NoSQLStreamMode;
import oracle.kv.pubsub.NoSQLSubscriptionConfig;
```

Next, we add several new private data members.

The first of these is `chkptIntv`, indicating how many operations this application will see before it runs a checkpoint. In this case, for illustration purposes, we are running a checkpoint for every ten operations. If this were production code, this would probably prove to be too frequent. Also, you are not required to take a checkpoint on a number of operations interval. You can perform them for any reason whatsoever. You could, for example, take checkpoints on a clock interval. Or you could take them whenever you see a `delete` operation, or whenever you see a table row written that conforms to some meaningful criteria.

Beyond the checkpoint interval, we indicate our stream mode will be `FROM_CHECKPOINT`.

```
public class GSGStreamExample {

    /* table to subscribe */
    private static final String TABLE_NAME = "Users";
    /* Number of operations to stream */
    private static final int num = 100;

    private static final String storeName = "kvstore";
    private static final String[] hhosts = {"localhost:5000"};

    /* max subscription allowed time before forced termination */
    private static final long MAX_SUBSCRIPTION_TIME_MS =
        Long.MAX_VALUE;

    private static final String rootPath = ".";
    private static final String CKPT_TABLE_NAME = "CheckpointTable";
    /* number of ops before a checkpoint is performed */
    private long ckptIntv = 10;
    private NoSQLStreamMode streamMode =
                            NoSQLStreamMode.FROM_CHECKPOINT;
```

Next we add the desired stream mode when we configure the subscription.

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

        final GSGStreamExample gte = new GSGStreamExample(args);
        gte.run();
    }

    private GSGStreamExample(final String[] argv) {
    }
```

```
/*
 * Subscribes a table. The work flow is ReactiveStream
 * compatible
 */
private void run() throws Exception {

    NoSQLPublisher publisher = null;
    try {
        /* step 1 : create a publisher configuration */
        final NoSQLPublisherConfig publisherConfig =
            new NoSQLPublisherConfig.Builder(
                new KVStoreConfig(storeName, hhosts), rootPath)
                .build();

        /* step 2 : create a publisher */
        publisher = NoSQLPublisher.get(publisherConfig);

        /* step 3: create a subscription configuration */
        final NoSQLSubscriptionConfig subscriptionConfig =
            /* stream with specified mode */
            new NoSQLSubscriptionConfig.Builder(CKPT_TABLE_NAME)
            .setSubscribedTables(TABLE_NAME)
            .setStreamMode(streamMode)
            .build();
```

The only other change to this application is to provide our checkpoint interval to our `NoSQLSubscriber` implementation. Again, this change is driven purely by how we choose to know when to take a checkpoint in this example. Your production code can, and probably will, do something entirely different.

```
        /* step 4: create a subscriber */
        final GSGSubscriberExample subscriber =
            new GSGSubscriberExample(subscriptionConfig, num,
                    ckptIntv);
        System.out.println("Subscriber created to stream " +
                num + " operations.");

        /* step 5: create a subscription and start stream */
        publisher.subscribe(subscriber);
        if (!subscriber.isSubscriptionSucc()) {
            System.out.println("Subscription failed for " +
                        subscriber.getSubscriptionConfig()
                                  .getSubscriberId() +
                        ", reason " +
                        subscriber.getCauseOfFailure());

            throw new RuntimeException("fail to subscribe");
        }
        System.out.println("Start stream " + num +
                " operations from table " + TABLE_NAME);


        /*
         * Wait for the stream to finish. Throw exception if it
```

```
                 * cannot finish within max allowed elapsed time
                 */
                final long s = System.currentTimeMillis();
                while (subscriber.getStreamOps() < num) {
                    final long elapsed = System.currentTimeMillis() - s;
                    if (elapsed >= MAX_SUBSCRIPTION_TIME_MS) {
                        throw new
                            RuntimeException("Not done within max " +
                                    "allowed elapsed time");
                    }
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        throw new RuntimeException("Interrupted!");
                    }
                }

                /* step 6: clean up */
                publisher.close(true);
                System.out.println("Publisher closed normally.");

            } catch (Exception exp) {
                String msg = "Error: " + exp.getMessage();
                System.out.println(msg);
                if (publisher != null) {
                    publisher.close(exp, false);
                    System.out.println("Publisher closed with error.");
                }
                throw exp;
            } finally {
               System.out.println("All done.");
            }
        }
    }
```

Implementing Checkpoints in GSGSubscriberExample

In this section, we illustrate how to implement checkpoints by adding functionality to the examples provided in NoSQLSubscriber Example.

Be aware that you must also add functionality to the example streams application shown in Streams Example. For those updates, see the previous section, Implementing Checkpoints in GSGStreamExample.

New additions to the original example code are indicated by **bold** text.

The changes to `GSGSubscriberExample.java` are moderately extensive. To begin, we add some private data members necessary for our checkpoint implementation.

- `chkptInv` is the checkpoint interval that we defined when we updated `GSGStreamsExample` for checkpoints. This variable indicates the number of operations that this subscriber sees before running a checkpoint.
- `ckptSucc` is a flag to indicate whether a checkpoint is successful.

- `CHECKPOINT_TIMEOUT_MS` is the time in milliseconds a checkpoint can run before it is declared a failure.

```
package pubsub;

import java.util.List;

import oracle.kv.pubsub.NoSQLSubscriber;
import oracle.kv.pubsub.NoSQLSubscription;
import oracle.kv.pubsub.NoSQLSubscriptionConfig;
import oracle.kv.pubsub.StreamOperation;
import oracle.kv.pubsub.StreamPosition;

import oracle.kv.table.MapValue;
import oracle.kv.table.Row;

import org.reactivestreams.Subscription;

class GSGSubscriberExample implements NoSQLSubscriber {

    /* subscription configuration */
    private final NoSQLSubscriptionConfig config;

    /* number of operations to stream */
    private final int numOps;

    /* number of operations seen in the stream */
    private long streamOps;

    private NoSQLSubscription subscription;

    private boolean isSubscribeSucc;

    private Throwable causeOfFailure;

/* checkpoint interval in number of ops */
    private final long ckptInv;

    /*
     * true if checkpoint is successful.
     * because this value can technically be changed by
     * different threads, we declare it as volatile
     */
    private volatile boolean ckptSucc;
    /*
     * amount of time in milliseconds that the checkpoint
     * has to run before the operation times out.
     */
    private final static long CHECKPOINT_TIMEOUT_MS = 60 * 1000;
```

Next we change our class signature to allow specification of the checkpoint interval when this class is constructed. We also initialize our `ckptInv` private data member.

```
    GSGSubscriberExample(NoSQLSubscriptionConfig config,
                                int numOps, long ckptIntv) {
        this.config = config;
        this.numOps = numOps;

        causeOfFailure = null;
        isSubscribeSucc = false;
        streamOps = 0;
        subscription = null;

        this.ckptInv = ckptIntv;
    }

    @Override
    public NoSQLSubscriptionConfig getSubscriptionConfig() {
        return config;
    }

    @Override
    public void onSubscribe(Subscription s) {
        subscription = (NoSQLSubscription) s;
        subscription.request(numOps);
        isSubscribeSucc = true;
    }

    @Override
    public void onError(Throwable t) {
        causeOfFailure = t;
        System.out.println("Error: " + t.getMessage());
    }

    @Override
    public void onComplete() {
        /* shall be no-op */
    }

    @Override
    public void onWarn(Throwable t) {
        System.out.println("Warning: " + t.getMessage());
    }
```

Next, we implement `onCheckpointComplete()`, which was not implemented earlier. In this simple example, we use it only to indicate the checkpoint's success status. You can tell if the checkpoint is successful if the `cause` method parameter is `null`.

Notice that we cannot examine the return status of `NoSQLSubscription.doCheckpoint()` because that method runs asynchronously, in a separate thread. The reason is so that `doCheckpoint()` is free to return immediately without waiting for the checkpoint to complete.

```
    /* called when publisher finishes a checkpoint */
    @Override
    public void onCheckpointComplete(StreamPosition pos,
```

```
                                    Throwable cause) {
    if (cause == null) {
            ckptSucc = true;
            System.out.println("Finish checkpoint at position " +
                    pos);
        } else {
            ckptSucc = false;
            System.out.println("Fail to checkpoint at position " +
                    pos + ", cause: " + cause.getMessage());
        }

    }
```

Next, we update the `onNext()` method to always call a new internal method, `performCheckpoint()` (described next).

We could have added logic here to determine if it is time to run a checkpoint. Instead, we push that functionality into the new `doCheckpoint()` method.

```
    @Override
    public void onNext(StreamOperation t) {

        switch (t.getType()) {
            case PUT:
                streamOps++;
                System.out.println("\nFound a put. Row is:");

                StreamOperation.PutEvent pe = t.asPut();
                Row row = pe.getRow();

                Integer uid = row.get("uid").asInteger().get();
                System.out.println("UID: " + uid);

                MapValue myjson = row.get("myJSON").asMap();
                int quantity = myjson.get("quantity")
                                     .asInteger().get();
                String array =
                    myjson.get("myArray").asArray().toString();
                System.out.println("\tQuantity: " + quantity);
                System.out.println("\tmyArray: " + array);
                break;
            case DELETE:
                streamOps++;
                System.out.println("\nFound a delete. Row is:");
                System.out.println(t);
                break;

            default:
                throw new
                    IllegalStateException("Receive unsupported " +
                        "stream operation from shard " +
                                t.getRepGroupId() +
                                ", seq: " + t.getSequenceId());
        }

        performCheckpoint();
```

```
        if (streamOps == numOps) {
            getSubscription().cancel();
            System.out.println("Subscription cancelled after " +
                    "receiving " + numOps + " operations.");
        }
    }
```

Finally, we implement a new private method, `performCheckpoint()`. This method implements the bulk of the checkpoint functionality.

In this method, we first check if `chkptInv` is `0`. If it is, we return:

```
    private void performCheckpoint() {

        /* If 0, turn off checkpointing */
        if (ckptInv == 0) {
            return;
        }
```

A checkpoint is run if the number of `streamOps` is greater than zero, and if the number of `streamOps` is evenly divisible by `ckptInv`. If these conditions are met, `NoSQLSubscription.getCurrentPosition()` is used to get the current `StreamPosition`, and then `NoSQLSubscription.doCheckpoint()` is used to actually perform the checkpoint.

Finally, once the checkpoint concludes, we check its success status. Regardless of the success status, we report it to the console, and then we are done. For production code, we recommend that you consider taking more elaborate actions here, especially if the checkpoint was not successful.

```
            if (ckptSucc) {
                System.out.println("\nCheckpoint succeeded after "
                        + streamOps +
                        " operations at position " + ckptPos +
                        ", elapsed time in ms " +
                        (System.currentTimeMillis() - start));
                /* reset for next checkpoint */
                ckptSucc = false;
            } else {
                System.out.println("\nCheckpoint timeout " +
                                   "at position " + ckptPos +
                                   ", elapsed time in ms " +
                                   (System.currentTimeMillis() - start));
            }
        }
    }

    private boolean isCkptTimeout(long start) {
        return (System.currentTimeMillis() - start) >
            CHECKPOINT_TIMEOUT_MS;
    }

    String getCauseOfFailure() {
        if (causeOfFailure == null) {
            return "success";
        }
        return causeOfFailure.getMessage();
```

```
    }

    boolean isSubscriptionSucc() {
        return isSubscribeSucc;
    }

    long getStreamOps() {
        return streamOps;
    }

    NoSQLSubscription getSubscription() {
        return subscription;
    }

}
```

Example Checkpoint Behavior

As shown in Sample Streams Output, the reason why we want to implement checkpoints is so that our streams application will not consume operations from the very beginning of the stream every time it is run. Now that we have implemented checkpoints, our application will begin streaming from the last saved checkpoint.

On the initial run, of 100 operations, the application's behavior is no different from the original application, with the exception of the checkpoints. (Output is truncated for brevity.)

```
> java pubsub.GSGStreamExample
Subscriber created to stream 100 operations.
Start stream 100 operations from table Users

Found a put. Row is:
UID: 0
    Quantity: 10
    myArray: [19,10,3,6,14,17,20,8,7,20]

Found a put. Row is:
UID: 1
    Quantity: 5
    myArray: [2,3,10,12,5]

Found a put. Row is:
UID: 2
    Quantity: 9
    myArray: [16,6,19,17,6,11,19,1,6]

... skipped ops for brevity ...

Found a put. Row is:
UID: 9
    Quantity: 1
    myArray: [2]
Finish checkpoint at position {kvstore(id=1500857641631):
[rg1(vlsn=69)]}

Checkpoint succeeded after 10 operations at position
```

```
{kvstore(id=1500857641631): [rg1(vlsn=69)]}, elapsed time in ms 36

Found a put. Row is:
UID: 10
    Quantity: 3
    myArray: [4,7,9]

Found a put. Row is:
UID: 11
    Quantity: 5
    myArray: [14,9,14,14,12]

... skipped ops for brevity ...

Found a delete. Row is:
Del OP [seq: 233, shard id: 1, primary key: {
  "uid" : 54
}]

Found a put. Row is:
UID: 88
    Quantity: 6
    myArray: [4,12,2,2,11,9]

Found a put. Row is:
UID: 89
    Quantity: 1
    myArray: [4]
Fail to checkpoint at position {kvstore(id=1500857641631):
[rg1(vlsn=249)]}, cause: Cannot do checkpoint because there
is a concurrently running checkpoint for subscriber 1_0
Finish checkpoint at position {kvstore(id=1500857641631):
[rg1(vlsn=249)]}

Checkpoint succeeded after 100 operations at position
{kvstore(id=1500857641631): [rg1(vlsn=249)]}, elapsed time in ms 42
Publisher closed normally.
All done.
```

Notice in the previous output that at least one checkpoint failed to complete because there was already a concurrently running checkpoint. This happened because we are taking checkpoints far too frequently in this example. As a consequnce, we tried to take a checkpoint before the previous checkpoint finished. Extending the checkpoint interval to something more reasonable would eliminate the error situation.

Having completed one run of the example program, a subsequent run will begin where the previous run left off. In this example run, the previous stream left off on the database write that created the row with UID 89. The next run begins with the write operation that created row UID 90.

```
> java pubsub.GSGStreamExample
Subscriber created to stream 100 operations.
Start stream 100 operations from table Users

Found a put. Row is:
UID: 90
```

```
    Quantity: 3
    myArray: [3,1,8]

Found a put. Row is:
UID: 91
    Quantity: 4
    myArray: [2,9,6,13]

Found a put. Row is:
UID: 92
    Quantity: 6
    myArray: [2,3,9,9,7,3]


... skipped ops for brevity ...
```

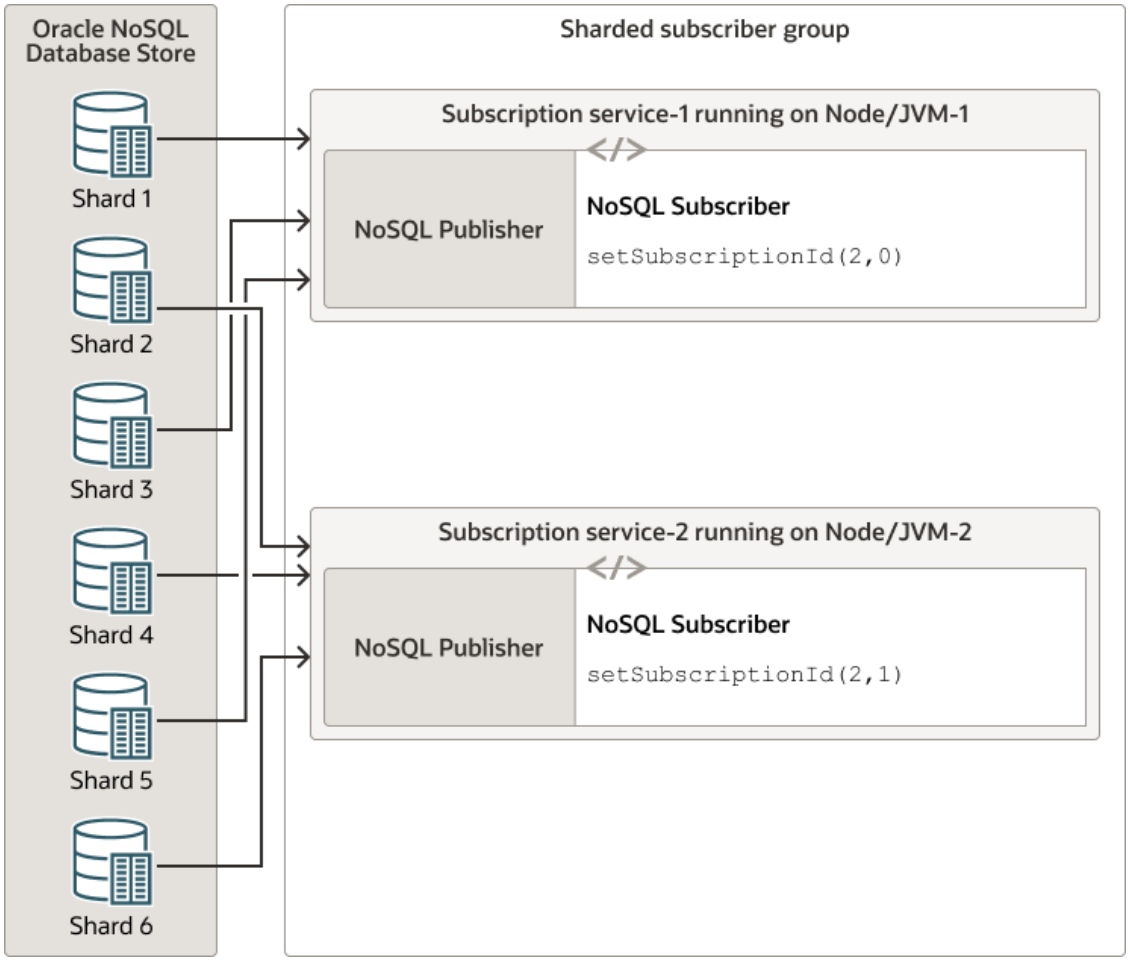
7

Scaling a Streams Application

You can scale a single streaming service to run on multiple nodes to handle a high volume of stream events from large Oracle NoSQL Database stores. The streaming service can use multiple subscribers to stream data from the Oracle NoSQL Database store.

The stream processing application does not need to know the topology of the Oracle NoSQL Database store, but can simply add or remove more independent subscribers as needed. All that the stream processing application needs to specify is the number of subscribers and a subscriber ID.

The following illustration depicts how a stream application can be scaled to use two clients to stream data from the six shards of the Oracle NoSQL Database store:



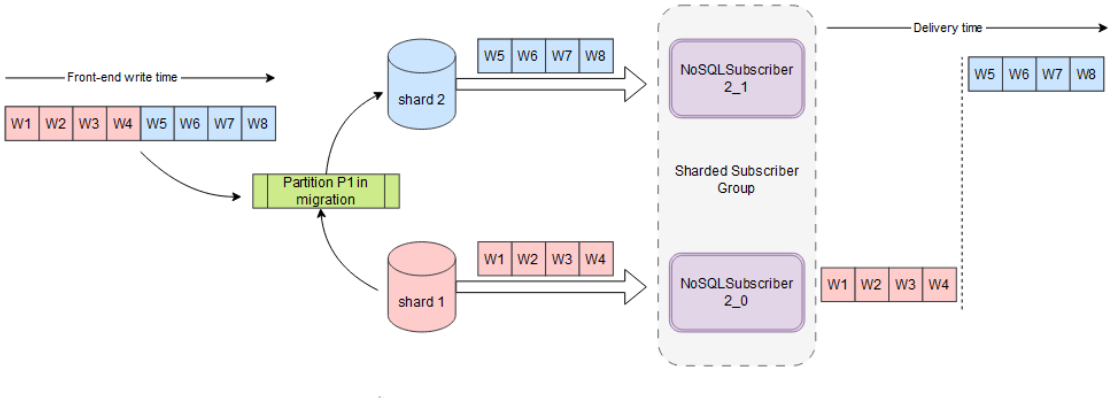
The Oracle NoSQL Database store strives to achieve even distribution of streams among its subscribers. As shown in the illustration, there are six shards and two subscribers. In this example, each subscriber receives streams from three shards. The subscriber does not choose which shard it gets streams from. The system determines this automatically, and the decision is transparent to the subscribers. In cases where there are more shards than of

subscribers (as in this example), some subscribers can receive streams from more than one shard.

> **Note**
>
> - The maximum number of scalable subscribers cannot exceed the number of shards. For example, if the Oracle NoSQL Database has six shards, subscribers cannot be scaled to more than six clients.
> - Oracle NoSQL Databasethat you scale stream processing applications by running them on different nodes to benefit from newly added resources.

Although scalable subscribers can be created and run inside separate JVMs on the same node, such configuration would not have any benefit over running a single subscriber without using scalable subscribers. In our example, running two scalable subscribers inside different JVMs (but within the same node), streaming over three shards each, would not benefit over running a single subscriber on the same node that is subscribed to the entire data store.

Elastic operations in data stores to Streams API are supported both when a single subscriber or multiple, sharded subscribers are deployed. Sharded subscribers are a part of a Subscriber group. When multiple sharded subscribers are deployed, after an elastic operation, the writes to a shard key would be delivered (via your implementation of `NoSQLSubscriber#onNext`), in the same order as they were made in the data store.



The above figure shows an example that two sharded subscribers streaming writes to a single key in a partition P1 during migration. There are two shards, source shard1 and target shard2. Writes to a migrating partition P1 are partly in shard1 and partly on shard2. The partition P1 would be removed from the shard1 after the migration is complete. Similarly, the partition P1 would be re-built at shard2 after the migration is complete. All writes to the partition P1 at shard1 will stream from shard1 to the sharded subscriber (whose id is 2_0). Similarly, all writes to partition P1 at shard 2 will stream from shard2 to the sharded subscriber (whose id is 2_1). From the diagram you can see that writes to shard1 takes place prior to writes in shard 2. At the delivery time, all writes delivered to subscriber 2_0 will be delivered before all writes delivered to subscriber 2_1. This will ensure the correct order of delivery of multiple sharded subscribers.

Scaling Subscribers

To add or remove subscribers running on different nodes, `NoSQLSubscriptionConfig` has to be created with the following additional builder API.

```
/* step 3: create a subscription configuration */
  final NoSQLSubscriptionConfig subscriptionConfig =
    // Scalable subscriber should set Subscriber Id
    // with 2 as total number of subscribers and
    // 0 as its own SubscriberId within the group of 2 subscribers

    new NoSQLSubscriptionConfig.Builder(CKPT_TABLE_NAME)
    .setSubscribedTables("usertable")
    .setSubscriberId(new NoSQLSubscriberId(2,0))
    .setStreamMode(streamMode)
    .build();
```

The API `setSubscriberId()` takes a single argument `NoSQLSubscriberID`. `NoSQLSubscriberId` is an object with both total number of subscribers and subscriber index. Hence, we need the following two arguments to construct a `NoSQLubscriberId` object.

- Number of Subscribers

  The total number of subscribers that would be running on different nodes. For example, in the code example above, `.setSubscriberId(new NoSQLSubscriberId(2,0))`, the `NoSQLSubscriberId` created has two subscribers in total.

- Subscriber Index

  A numerical index of the current subscriber among the total number of subscribers. Note that a numerical index begins with 0. For example, two subscriber clients can be identified as 0 and 1.

Configure multiple sharded subscribers as part of a subscriber group

To build a `NoSQLSubscriptionConfig` object for a sharded subscriber of a subscriber group, you use a constructor of the `NoSQLSubscriptionConfig.Builder` class that has a single parameter of a mapper function. This function returns a distinct checkpoint table name computed from each subscriber id in the group. A sample constructor of the Builder class for configuring sharded subscribers is shown below.

```
public void Builder(Function<NoSQLSubscriberId, String> ckptTableMapper);
```

You can build the `NoSQLSubscriptionConfig` object for a sharded subscriber as shown below:

```
/** * Configures a sharded subscriber
* @param subscriberId subscriber id
* @param tables subscribed tables
*/
void configureShardedSubscribers(NoSQLSubscriberId subscriberId, Set<String>
tables) {
   finalNoSQLSubscriptionConfig conf =
     newNoSQLSubscriptionConfig.Builder( id -> buildCkptTableName(id))
                                          .setSubscribedTables(tables)
                                          .setSubscriberId(subscriberId)
```

```
                                                .build();
}
```

An example code snippet is shown below where the `buildCkptTableName` builds the checkpoint name from the subscriber id by concatenating the subscriber id to *StreamCheckpointTable*. This is one way of using the mapper function. Your application can create its own mapper function, as long as it returns unique checkpoint table name for each subscriber.

```
/**
 * Builds stream checkpoint table from given subscriber id
 * @param subscriberId subscriber id
 * @return stream checkpoint table name for that subscriber
 */
private String buildCkptTableName(NoSQLSubscriberId subscriberId) {
    return "StreamCheckpointTable" + subscriberId.toString();
}
```

A
GSGStreamsWriteTable

The examples in this document rely on a `Users` table that is populated with data. The application we used to create and populate this table is provided in this appendix.

> **Note**
>
> While this example does not use namespaces, the streaming API supports them. To assess a table in a namespace, such as ns1, prefix the table name with the namespace, followed by a colon. For example: `ns1:Users`.

We provide this class without comment and solely for completeness. The actions taken by this class should be familiar to anyone who has used the Oracle NoSQL Database Java API. See *Java Direct Driver Developer's Guide*.

```
package pubsub;

import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import oracle.kv.FaultException;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.StatementResult;

import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;


public class GSGStreamsWriteTable {

    private static final String[] hhosts = {"localhost:5000"};
    private static final int MAX_ROWS = 200;

    public static void main(String args[]) {
        GSGStreamsWriteTable gswt = new GSGStreamsWriteTable();

        gswt.run(args);

        System.out.println("All done.");
    }

    private void run(String args[]) {
        KVStoreConfig kconfig = new KVStoreConfig("kvstore", hhosts);
```

```
        KVStore kvstore = KVStoreFactory.getStore(kconfig);

        defineTable(kvstore);
        writeTable(kvstore);
    }

    private void defineTable(KVStore kvstore) {
        System.out.println("Creating table schema....");
        TableAPI tableAPI = kvstore.getTableAPI();
        StatementResult result = null;
        String statement = null;

        try {
            statement = "DROP TABLE IF EXISTS Users";
            result = kvstore.executeSync(statement);
            displayResult(result, statement);

            statement = "CREATE TABLE Users ( " +
                "    uid INTEGER, " +
                "    myJSON JSON, " +
                "    PRIMARY KEY(uid))";
            result = kvstore.executeSync(statement);
            displayResult(result, statement);

        } catch (IllegalArgumentException e) {
            System.out.println("Invalid statement:\n" +
                    e.getMessage());
        } catch (FaultException e) {
            System.out.println
              ("Statement couldn't be executed, please retry: " + e);
        }
    }

    private void writeTable(KVStore kvstore) {
        System.out.println("In writeTable....");

        TableAPI tableH = kvstore.getTableAPI();

        Table myTable = tableH.getTable("Users");
        int count = 0;
        Random rand = new Random();

        /*
         * Write rows to the table, using random information
         * for the JSON data.
         */
        while (count < MAX_ROWS) {
            Row row = myTable.createRow();
            row.put("uid", count);

            int q = rand.nextInt(10) + 1;
            List<Integer> integersList = new ArrayList<Integer>();
            int a_count = 0;
            while (a_count < q) {
                int val = rand.nextInt(q + 10) + 1;
                integersList.add(val);
```

```
                    a_count++;
                }

                String json = "{";
                json += "\"quantity\" : " + q + ", ";
                json += "\"myArray\" : " + integersList.toString();
                json += "}";

                /* Write the row to the store */
                row.putJson("myJSON",  json);
                tableH.put(row, null, null);

                /* Randomly delete table rows */
                int shouldDelete = rand.nextInt(10);
                if (shouldDelete == 1) {
                    /* Randomly select a row to delete */
                    int toDelete = rand.nextInt(count);
                    PrimaryKey pk = myTable.createPrimaryKey();
                    pk.put("uid", toDelete);
                    tableH.delete(pk, null, null);
                }

                count++;
            }
            System.out.println("Wrote " + count + " rows");
        }

        private void displayResult(StatementResult result,
                                   String statement) {
            System.out.println("============================");
            if (result.isSuccessful()) {
                System.out.println("Statement was successful:\n\t" +
                        statement);
                System.out.println("Results:\n\t" + result.getInfo());
            } else if (result.isCancelled()) {
                System.out.println("Statement was cancelled:\n\t" +
                        statement);
            } else {
                /*
                 * statement wasn't successful: may be in error, or may
                 * still be in progress.
                 */
                if (result.isDone()) {
                    System.out.println("Statement failed:\n\t" +
                            statement);
                    System.out.println("Problem:\n\t" +
                            result.getErrorMessage());
                } else {
                    System.out.println("Statement in progress:\n\t" +
                            statement);
                    System.out.println("Status:\n\t" +
                            result.getInfo());
                }
            }
        }
    }
```