# 25 Support for Java library for Reactive Streams Ingestion

Starting with Oracle Database Release 21c, you have a Java library that provides support for Reactive Streams Ingestion (RSI), which enables efficient streaming of data into Oracle Database. The new Java library enables Java applications to continuously receive and ingest data from a large group of clients.

Using the direct path load method of Oracle Database for inserting data, the new Java library makes the ingestion process nonblocking and extremely fast. It uses an extension of the existing UCP APIs, which enables the ingestion process to furnish several high-availability and scalability features of the database, like support for table partitions, Oracle RAC connection affinity, and sharding.

## 25.1 Overview of the Java Library for Reactive Streams Ingestion

The Java library for Reactive Streams Ingestion (RSI) enables efficient data streaming into Oracle Database in a nonblocking way.

This library is particularly useful when a large number of clients use the database to persist information in the form of table rows, and do not want to be blocked waiting for a synchronous response from the database. So, in case of use cases like the following, when the application must ingest streaming data into the database at a very high speed, and persist it in the form of rows in an Oracle Database table, you can use this library:

- Internet of Things (IoT) sensors
- Time series data for stock trade
- Call detail records (CDRs)
- Geospatial activities
- Video web sites
- Social media posts

For using this library, you must add the following `JAR` files to your `CLASSPATH`:

- `rsi.jar`
- `ojdbc11.jar` or `ojdbc17.jar`
- `ucp.jar`
- `ons.jar`

## 25.2 Features of the Java Library for Reactive Streams Ingestion

The Java library for Reactive Streams Ingestion (RSI) uses the direct path load method of Oracle Database for inserting data. It also uses Oracle Universal Connection Pool (UCP) to furnish several high-availability and scalability features of the Database, such as support for table partitions, Oracle RAC connection affinity, and sharding. So, this library provides the benefits of the following features:

- Reactive Streams Ingestion
- Direct Path Load
- Universal Connection Pool

## 25.2.1 Reactive Streams Ingestion

This is the core feature of the Java library for Reactive Streams Ingestion, which provides the APIs to handle the logic.

The library implements the `Java.util.concurrent.Flow` Subscriber interface. You must implement the `Java.util.concurrent.Flow` Publisher interface for invoking the methods of the Subscriber interface. The Subscriber interface has the following methods:

- `onSubscribe`
  Invoke this method only once for establishing the initial relationship between the Subscriber interface and the Publisher interface.
- `onNext`
  Invoke this method for creating each new row in the implementation of the Subscriber interface that the library provides.
- `onError`
  Invoke this method in case of any error that might occur during the ingestion process.
- `onComplete`
  Invoke this method when the ingestion job completes.

The Subscriber interface calls the `request(n)` method or the `cancel` method of the `Java.util.concurrent.Flow` Subscriber interface to indicate whether it can accept more data or it should stop ingesting data.

**See Also:**

The Java.util.concurrent.Flow Subscriber Interface

## 25.2.2 Direct Path Load

The Java library for Reactive Streams Ingestion uses the direct path load method of Oracle Database for nonblocking data ingestion. This method eliminates the SQL layer overhead significantly as it formats Oracle data blocks and writes the data blocks directly to the database files.

**See Also:**

The Direct Path Load Method

During the direct path load method call, the database appends the inserted data after the existing data in the table. This method writes data directly into data files, bypassing the buffer cache. It does not perform reuse of free space in the table, and ignores the referential integrity constraints. So, a direct path load method can perform significantly better than conventional insert operations.

### 25.2.3 Universal Connection Pool

The Java library for Reactive Streams Ingestion (RSI) uses Universal Connection Pool (UCP) for various connection pooling and management activities like sharding topology knowledge, Fast Application Notification (FAN) awareness for an Oracle Real Application Cluster (Oracle RAC) database, and so on.

The RSI library uses the UCP sharding APIs to establish a proper connection for the specified sharding key. You can map each record in RSI to a unique chunk ID and then use the unique chunk ID to group these records together. When the RSI library has enough records to send to the database, then it borrows a chunk specific connection from UCP and uses that connection to insert the record into the sharded database.

## 25.3 About Reactive Streams Ingestion (RSI) Modes

The current release introduces the new DataLoad mode, which is useful when you use the Java library for Reactive Streams Ingestion (RSI) to execute a large `INSERT` batch to the database.

Starting from Oracle Database 23ai Release, you can use the Java library for Reactive Streams Ingestion (RSI) in the following modes, depending on your business use case:

- **The Streaming mode:** Use this default mode, when you want to use RSI in a server where the number of rows to be inserted is not finite, but you do not need to insert a large number of rows at one go.
- **The DataLoad Mode:** Use the DataLoad mode, when there is a known large list of records to be inserted to the database at one go.

The key differences between the DataLoad mode and the Streaming mode are:

- In the DataLoad mode, the changes are not committed until the RSI instance is closed. In the default Streaming mode, the changes are committed regularly. This can negatively impact the throughput, when you execute a large `INSERT` batch to the database.
- In the DataLoad mode, each worker thread has its own JDBC connection. So, there is no effort to reduce the number of JDBC connections needed to execute the insertion task. This behavior is different from the default Streaming mode, where the worker threads share a pool of JDBC connections.

### 25.3.1 Enabling the DataLoad Mode

The Streaming mode is enabled by default with the Java library for Reactive Streams Ingestion. To enable the DataLoad mode, you must use the `useDataLoadMode` method.

Enable the DataLoad mode as shown in the following code snippet:

```
ReactiveStreamsIngestion.Builder rsiBuilder =
ReactiveStreamsIngestion.builder()
     .useDataLoadMode()
     .username("<user_name>")
     .password("<password>")
     .url("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=myhost.com)(PORT=5521))(CONNECT_DATA=(SERVICE_NAME=myservice.com)))")
     .table("customers")
     .columns (new String[] { "id", "name", "region" });
```

```
// Use try-with-resource statement to ensure that RSI instance is closed at
the
// end of the statement.
try (ReactiveStreamsIngestion rsi = rsiBuilder.build()){
   // Publish Records.
}
```

## 25.4 Code Samples: Java Library for Reactive Streams Ingestion

This section contains code samples showing how to use the Reactive Streams Ingestion library.

- PushPublisher
- Flow.Publisher Dynamic Implementations
- Flow.Publisher Third-Party implementations

### 25.4.1 PushPublisher

This is the simplest way to use the Reactive Streams Ingestion (RSI) library, where the RSI library implements the `java.util.concurrent.Flow.Subscriber` interface and the Java code in your application implements the `java.util.concurrent.Flow.Publisher` interface.

The following example demonstrates this implementation, where you create a Publisher and the RSI library subscribes to that Publisher:

**Example 25-1    PushPublisher**

```
package oracle.rsi.demos;
import java.sql.SQLException;
import java.time.Duration;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import oracle.rsi.ReactiveStreamsIngestion;
import oracle.rsi.PushPublisher;

public class SimplePushPublisher {

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

    ExecutorService workerThreadPool = Executors.newFixedThreadPool(2);

    ReactiveStreamsIngestion rsi = ReactiveStreamsIngestion
        .builder()
        .url(
            "jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=example.com)(PORT=5521))(CONNECT_DATA=(SERVICE_NAME=myservice.com)))")
        .username(<user_name>)
        .password(<password>)
        .schema(<schema_name>)
        .executor(workerThreadPool)
        .bufferRows(10)
        .bufferInterval(Duration.ofSeconds(20))
        .table("customers")
        .columns(new String[] { "id", "name", "region" })
        .build();

    PushPublisher<Object[]> pushPublisher =
ReactiveStreamsIngestion.pushPublisher();
    pushPublisher.subscribe(rsi.subscriber());

    //Ingests byte arrays using the accept method
    pushPublisher.accept(new Object[] { 1, "John Doe", "North" });
    pushPublisher.accept(new Object[] { 2, "Jane Doe", "North" });
    pushPublisher.accept(new Object[] { 3, "John Smith", "South" });

    try {
      pushPublisher.close();
    } catch (Exception e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

    rsi.close();

    workerThreadPool.shutdown();

  }

}
```

## 25.4.2 Flow.Publisher Dynamic Implementations

The following example shows how to use the RSI library when your application implements the Flow.Publisher interface and subscribes to the library.

**Example 25-2 Flow.Publisher Dynamic Implementations**

```
package oracle.rsi.demos;
import java.sql.SQLException;
import java.time.Duration;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Flow.Publisher;
import java.util.concurrent.Flow.Subscriber;
import java.util.concurrent.Flow.Subscription;
import java.util.function.Consumer;

import oracle.rsi.ReactiveStreamsIngestion;

public class SimpleFlowPublisher {

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

    ExecutorService workerThreadPool = Executors.newFixedThreadPool(2);

    ReactiveStreamsIngestion rsi = ReactiveStreamsIngestion
        .builder()
        .url(
            "jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=example.com)(PORT=5521))(CONNECT_DATA=(SERVICE_NAME=myservice.com)))")
        .username(<user_name>)
        .password(<password>)
        .schema(<schema_name>)
        .executor(workerThreadPool)
        .bufferRows(1)
        .bufferInterval(Duration.ofMinutes(60))
        .table("customers")
        .columns(new String[] { "id", "name", "region" })
        .build();

    SimpleObjectPublisher<Object[]> publisher = new
SimpleObjectPublisher<Object[]>();
    publisher.subscribe(rsi.subscriber());

    SimpleObjectPublisher<Object[]> anotherPublisher = new
SimpleObjectPublisher<Object[]>();
    anotherPublisher.subscribe(rsi.subscriber());

    publisher.accept(new Object[] { 1, "John Doe", "North" });
    publisher.accept(new Object[] { 2, "Jane Doe", "North" });
    publisher.accept(new Object[] { 3, "John Smith", "South" });

    anotherPublisher.accept(new Object[] { 4, "John Doe", "North" });
    anotherPublisher.accept(new Object[] { 5, "Jane Doe", "North" });
    anotherPublisher.accept(new Object[] { 6, "John Smith", "South" });

    rsi.close();

    workerThreadPool.shutdown();

  }

}

class SimpleObjectPublisher<T> implements Publisher<T>, Consumer<T> {

  Subscriber<? super T> subscriber;

  Subscription subscription = new SimpleObjectSubscription();

 //Data streaming starts

  @Override
  public void subscribe(Subscriber<? super T> subscriber) {
    this.subscriber = subscriber;
    this.subscriber.onSubscribe(subscription);
  }

  @Override
  public void accept(T t) {
    subscriber.onNext(t);
  }

}
```

```
 // You must provide this subscription
class SimpleObjectSubscription implements Subscription {

  @Override
  public void request(long n) {
    System.out.println("Library requesting: " + n + " records");
  }

  @Override
  public void cancel() {
    // TODO Auto-generated method stub
  }

}
```

## 25.4.3 Flow.Publisher Third-Party implementations

The following example shows how to use the RSI library when a third-party implements the Flow.Publisher interface.

In the following example, the standard JDK `SubmissionPublisher` interface is the third-party interface.

**Example 25-3    Flow.Publisher Third-Party Implementations**

```
package oracle.rsi.demos;
import java.sql.SQLException;
import java.time.Duration;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.SubmissionPublisher;

import oracle.rsi.ReactiveStreamsIngestion;

public class SimpleSubmissionPublisher {

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

    ExecutorService workerThreadPool = Executors.newFixedThreadPool(2);

    ReactiveStreamsIngestion rsi = ReactiveStreamsIngestion
        .builder()
        .url(
            "jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=example.com)(PORT=5521))(CONNECT_DATA=(SERVICE_NAME=myservice.com)))")
        .username(<user_name>)
        .password(<password>)
        .schema(<schema_name>)
        .executor(workerThreadPool)
        .bufferRows(10)
        .bufferInterval(Duration.ofSeconds(20))
        .table("customers")
        .columns(new String[] { "id", "name", "region" })
        .build();
    SubmissionPublisher<Object[]> publisher = new SubmissionPublisher<>();
    publisher.subscribe(rsi.subscriber());

    publisher.submit(new Object[] { 1, "John Doe", "North" });
    publisher.submit(new Object[] { 2, "Jane Doe", "North" });
    publisher.submit(new Object[] { 3, "John Smith", "South" });

    while (publisher.estimateMaximumLag() > 0);

    try {
      publisher.close();
    } catch (Exception e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

    rsi.close();

    workerThreadPool.shutdown();

  }

}
```

## 25.5 Limitations of Java library for Reactive Streams Ingestion

Reactive streams ingestion of data streams may not ingest all the data into the database in case of unexpected crashes or errors because the library may lose some of the records while trying to store them in the database.

Apart from the possibility of data loss due to database crashes, this library has the following limitations:

- It does not support database triggers.
- It does not check referential integrity.
- It does not support clustered tables.
- It does not support loading of remote objects.
- It does not support loading of `VARRAY` columns.
- It does not support `LONG` data type with streams.
- It expects the `LONG` data type to be the last column in the database table.
- It expects all partitioning columns to appear before any column with `LOB` data type.