26
Support for Pipelined Database Operations

Java applications can now asynchronously submit several SQL requests to the server without waiting for the return of the preceding calls.

This chapter discusses more about pipelining in the following topics:

26.1 Overview of Pipelining

Pipelining is a form of network communication in which an application can send multiple requests to a server, without having to wait for a response.

The fundamental idea of pipelining is to keep the server busy and enable an application to use the interleaving requests appropriately. The application can keep sending requests, while the server builds up a queue and processes those requests one by one. Then, the server sends the responses back to the client in the same order in which it received the requests.

As the pipeline does not read responses very often, using the data from a previous SQL response as in-bind data to the subsequent request creates a dependency and breaks the pipeline. So, the application must ensure that the requests are independent because it is an essential requirement for the pipelining functionality.

Pipelining offers the following benefits to your applications:

- Improved response time and throughput
- Improved scalability due to reduced context switching

26.2 JDBC Support for Pipelining

In the previous releases, the JDBC driver did not allow a new database call to start until the current call had been completed. Starting with Oracle Database Release 23ai, JDBC Thin drivers now support pipelined database operations.

As a pipeline is executed by sending multiple requests without waiting for a response, this feature translates to an asynchronous programming model, in which the execution of multiple SQL statements can begin even before a thread consumes the results of those statements.

You can implement pipelining in your JDBC applications, using the following asynchronous programming features of Oracle Database:

> **Note:**
>
> When calls are made to a reactive (asynchronous) API or a standard JDBC batching API, pipelining is enabled by default.

26.3 Pipelining with Reactive Extensions

Starting from Oracle Database Relese 23ai, you can use the Reactive Extensions APIs to carry out pipelined database operations.

The following code provides a simplified example of how you can use the Reactive Extensions APIs to carry out a pipeline:

In the following example, a call to the `unwrap(OraclePreparedStatement.class)` method is used to access the `executeUpdateAsyncOracle` and `executeQueryAsyncOracle` methods. These methods return a `Flow.Publisher` that implements the Reactive Streams specification. As the result of each SQL statement is received, the corresponding Publisher signals that result to a Subscriber.

```
...
void pipelineExample(OracleConnection connection) throws SQLException {

  // Prepare statements to execute
  PreparedStatement delete = connection.prepareStatement(
    "DELETE FROM example WHERE id = 0");
  PreparedStatement insert = connection.prepareStatement(
    "INSERT INTO example (id, value) VALUES (1, 'x')");
  PreparedStatement select = connection.prepareStatement(
    "SELECT id, value FROM example WHERE id = 2");

  // Execute statements in a pipeline
  Flow.Publisher<Long> deletePublisher =
    delete.unwrap(OraclePreparedStatement.class)
      .executeUpdateAsyncOracle();
  Flow.Publisher<Long> insertPublisher =
    insert.unwrap(OraclePreparedStatement.class)
      .executeUpdateAsyncOracle();
  Flow.Publisher<OracleResultSet> selectPublisher =
    select.unwrap(OraclePreparedStatement.class)
      .executeQueryAsyncOracle();
...
}
```

26.4 Pipelining with Java library for Reactive Streams Ingestion

Starting from Oracle Database Relese 23ai, you can use the Java library for Reactive Streams Ingestion to carry out pipelined database operations.

The following code provides a simplified example of how you can use the Java library for Reactive Streams Ingestion to carry out a pipeline:

In the following example, instances of `Mono` and `Flux` publishers are created by calling their `from` methods. The argument to the method is an `org.reactivestreams.Publisher`, which is adapted from a `Flow.Publisher`, by calling `org.reactivestreams.FlowAdapters.toPublisher`. A subscribe method is called to asynchronously consume results with a callback function. To consume a result of multiple rows, a call to `Mono.flatMapMany` converts a stream of a single `ResultSet` into a stream of multiple row values.

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OraclePreparedStatement;
import oracle.jdbc.OracleResultSet;
import org.reactivestreams.FlowAdapters;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.concurrent.CompletionException;
import java.util.concurrent.Flow;

public class ReactiveExample {

  void reactivePipelineExample(OracleConnection connection) throws
SQLException {

    // Push a DELETE into the pipeline
    Mono.using(
        () -> connection.prepareStatement("DELETE FROM example WHERE id = 1"),
        preparedStatement -> Mono.from(publishUpdate(preparedStatement)),
        preparedStatement -> close(preparedStatement))
      .subscribe(deleteCount ->
        System.out.println(deleteCount + " rows deleted"));

    // Push an INSERT operation into the pipeline
    Mono.using(
        () -> connection.prepareStatement(
          "INSERT INTO example (id, value) VALUES (1, 'x')"),
        preparedStatement -> Mono.from(publishUpdate(preparedStatement)),
        preparedStatement -> close(preparedStatement))
      .subscribe(insertCount ->
        System.out.println(insertCount + " rows inserted"));

    // Push a SELECT into the pipeline
    Flux.using(
        () -> connection.prepareStatement(
```

```
            "SELECT id, value FROM example ORDER BY id"),
          preparedStatement ->
            Mono.from(publishQuery(preparedStatement))
              .flatMapMany(resultSet -> publishRows(resultSet)),
          preparedStatement -> close(preparedStatement))
        .subscribe(rowString ->
          System.out.println(rowString));
    }
    Publisher<Long> publishUpdate(PreparedStatement preparedStatement) {
      try {
        Flow.Publisher<Long> updatePublisher =
          preparedStatement.unwrap(OraclePreparedStatement.class)
            .executeUpdateAsyncOracle();

        return FlowAdapters.toPublisher(updatePublisher);
      }
      catch (SQLException sqlException) {
        return Mono.error(sqlException);
      }
    }

    Publisher<OracleResultSet> publishQuery(PreparedStatement
  preparedStatement) {
      try {
        Flow.Publisher<OracleResultSet> queryPublisher =
          preparedStatement.unwrap(OraclePreparedStatement.class)
            .executeQueryAsyncOracle();

        return FlowAdapters.toPublisher(queryPublisher);
      }
      catch (SQLException sqlException) {
        return Mono.error(sqlException);
      }
    }

    Publisher<String> publishRows(ResultSet resultSet) {
      try {
        Flow.Publisher<String> rowPublisher =
          resultSet.unwrap(OracleResultSet.class)
            .publisherOracle(row -> {
              try {
                return String.format("id: %d, value: %s\n",
                  row.getObject("id", Long.class),
                  row.getObject("value", String.class));
              }
              catch (SQLException sqlException) {
                throw new CompletionException(sqlException);
              }
            });

        return FlowAdapters.toPublisher(rowPublisher);
      }
      catch (SQLException sqlException) {
        return Flux.error(sqlException);
      }
    }
```

```
  void close(PreparedStatement preparedStatement) {
    try {
      preparedStatement.close();
    }
    catch (SQLException sqlException) {
      throw new RuntimeException(sqlException);
    }
  }

}
```