29

JDBC Support for Database Sharding

This section describes Oracle JDBC support for the Database Sharding feature.

- Overview of Database Sharding for JDBC Users
- Creating JDBC Connections Using a Sharding Key
- Overview of the Sharding Data Source
- Benefits of the Sharding Data Source
- Limitations of the Sharding Data Source

29.1 Overview of Database Sharding for JDBC Users

Modern web applications face new scalability challenges with huge volumes of data. A commonly accepted solution to this problem is sharding. *Sharding* is a data tier architecture, where data is horizontally partitioned across independent databases. Each database in such a configuration is called a *shard*.

All shards together make up a single logical database, which is referred to as a Globally Distributed Database. Sharding is a *shared-nothing* database architecture because shards do not share physical resources such as CPU, memory, or storage devices.

Sharding uses Global Data Services (GDS), where GDS routes a client request to an appropriate database based on parameters such as availability, load, network latency, and replication lag. A GDS pool is a set of replicated databases that offer the same global service. The databases in a GDS pool can be located in multiple data centers across different regions. A sharded GDS pool contains all shards of an Oracle Globally Distributed Database and their replicas, and appears as a single Globally Distributed Database to the database clients.

Starting from Oracle Database 12*c* Release 2 (12.2.0.1), Oracle JDBC supports database sharding. The JDBC driver recognizes the specified sharding key and super sharding key and connects to the relevant shard that contains the data. Once the connection is established to a shard, then any database operations, such as DMLs, SQL queries and so on, are supported and executed in the usual way.

See Also:

- *Oracle Globally Distributed Database*
- *Universal Connection Pool Developer's Guide*
- The OracleShardingKey Interface
- The OracleShardingKeyBuilder Interface
- The OracleConnectionBuilder Interface

29.2 Creating JDBC Connections Using a Sharding Key

The shard-aware applications must identify and build the sharding key and the super sharding key, which are required to establish a connection to an Oracle Globally Distributed Database.

Example 29-1   Building a Sharding Key

The following example shows how to build a sharding key:

```
import java.sql.Connection;
import java.sql.JDBCType;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ShardingKey;
import java.sql.Statement;

import oracle.jdbc.pool.OracleDataSource;

public class JDBCShardingExample {

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

        OracleDataSource ods = new OracleDataSource();
        ods.setURL("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=myhost)
(PORT=<gsm-listener-port>)
            (PROTOCOL = tcp))(CONNECT_DATA = (SERVICE_NAME = gsmservicename)))
    ");
    ods.setUser(<db_user_name>);
    ods.setPassword(<db_password>);

    int empId = 1234;
    String location = "US";

    // Employee ID is the sharding key column
    ShardingKey shardingKey = ods.createShardingKeyBuilder()
        .subkey(empId, JDBCType.INTEGER)
        .build();

    // Employee location is the super sharding key column
    ShardingKey superShardingKey = ods.createShardingKeyBuilder()
        .subkey(location, JDBCType.VARCHAR)
        .build();

    // Get a direct connection to the shard that contains the record of the
employee
    // with employee id = 1234 and location="US", using sharding key and
super sharding key

    try (Connection connection = ods.createConnectionBuilder()
        .shardingKey(shardingKey)
        .superShardingKey(superShardingKey)
        .build()) {

              PreparedStatement pst = connection.prepareStatement("select *
from employee where emp_id=? and location=?");
```

```
                pst.setInt(1, 1234);
                pst.setString(2, "US");
                ResultSet rs = pst.executeQuery();
                // Retrieve the employee details using resultset
                rs.close();
                pst.close();
            }
        }

}
```

Note:

- There is a fixed set of data types that are valid and supported. If any unsupported data types are used as keys, then exceptions are thrown. The following list specifies the supported data types:
  - `OracleType.VARCHAR2/JDBCType.VARCHAR`
  - `OracleType.CHAR/JDBCType.CHAR`
  - `OracleType.NVARCHAR/JDBCType.NVARCHAR`
  - `OracleType.NCHAR/JDBCType.NCHAR`
  - `OracleType.NUMBER/JDBCType.NUMERIC`
  - `OracleType.FLOAT/ JDBCType.FLOAT`
  - `OracleType.DATE/ JDBCType.DATE`
  - `OracleType.TIMESTAMP/JDBCType.TIMESTAMP`
  - `OracleType.TIMESTAMP_WITH_LOCAL_TIME_ZONE`
  - `OracleType.RAW`
- You must provide a sharding key that is compliant to the NLS formatting specified in the database.

29.3 Overview of the Sharding Data Source

Since Oracle Database Release 21c, the JDBC Thin driver can establish Java connectivity to an Oracle Globally Distributed Database without the need to furnish a sharding key. In this case, you do not need to identify and build the sharding key and the super sharding key to establish a connection.

The `oracle.jdbc.pool.OracleDataSource`, with sharding capability, scales out transparently to an Oracle Globally Distributed Database as it does not involve any change to the application code. When the sharding key can be derived from SQL or PL/SQL, the JDBC driver can identify it without the need for the application to send the sharding key.

For achieving this capability, you must set the connection property `OracleConnection.CONNECTION_PROPERTY_USE_SHARDING_DRIVER_CONNECTION` to `true`. The default value of this connection property is `false`.

29.3.1 Example: How to Use the Sharding Data Source

The example in this section shows how to use the sharding data source.

Example 29-2   Using the Sharding Data Source

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;

import javax.sql.DataSource;

import oracle.jdbc.internal.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;

public class ShardingDataSourceSample {
      public static void main(String[] args) throws SQLException {
            ShardingDataSourceSample sample = new ShardingDataSourceSample();
            DataSource ds = sample.getDataSource();
            // get the details of following customers
            int[] customerIds = new int[] { 100, 101, 102, 103, 104, 105 };
            try (Connection conn = ds.getConnection()) {
                  for (int id : customerIds) {
                        sample.displayCustomerDetails(conn, id);
                  }
                  System.out.println(((OracleConnection)
conn).getPercentageQueryExecutionOnDirectShard());
            }
      }

      private void displayCustomerDetails(Connection conn, int id) throws
SQLException {
            try (PreparedStatement pstmt = conn.prepareStatement("SELECT *
FROM CUSTOMER where ID = ?")) {
                  pstmt.setInt(1, id);
                  try (ResultSet rs = pstmt.executeQuery()) {
                        while (rs.next()) {
                              // print the customer details
                        }
                  }
            }
      }

      private DataSource getDataSource() throws SQLException {
        OracleDataSource ds = new OracleDataSource();
    ds.setURL(<gsmURL>);
    ds.setUser(<userName>);
    ds.setPassword(<password>);
    Properties prop = new Properties();
    // Connection property to enable sharding data source feature
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_USE_SHARDING_DRIVER_CONN
ECTION, "true");
    ds.setConnectionProperties(prop);
    return ds;
  }
}
```

29.4 Benefits of the Sharding Data Source

Following are the benefits of the new sharding data source:

- You do not need to use the sharding APIs to pass the sharding key because the sharding data source derives the sharding key from the SQL statement.
- You do not need to configure the Universal Connection Pool (UCP) because the sharding data source uses the auto tune feature of UCP.
- You do not need to check-in or check-out a physical connection for every new sharding key because the sharding data source does it automatically.
- You do not need to separate cross-shard statements from single-shard statements and create separate connection pools for them because the sharding data source maintains those connections pools.
- The sharding data source enables the prepared statement caching and routes the connection to the direct shard based on the key used in the SQL statement.
- The sharding data source simplifies applications and optimizes application performance without any code change.

29.5 Limitations of the Sharding Data Source

This section describes the limitations of the sharding data source.

- The sharding data source supports only the JDBC Thin driver. It does not support the JDBC OCI driver or the KPRB driver.
- The sharding data source does not support some Oracle JDBC extension APIs such as Direct Path Load, JDBC Dynamic Monitoring Service (DMS) metrics, and so on.
- The sharding data source supports PL/SQL execution only through the catalog database.
- When `AUTO COMMIT` is set to `OFF`, then the execution always happens on the catalog database.
- If the data source property `singleShardTransactionSupport` is set to `TRUE`, then the sharding data source supports local transactions against a single shard, when `AUTO COMMIT` is set to `OFF`.
  The following code snippet shows how to set the `singleShardTransactionSupport` property:

Example 29-3 Single Shard Transaction Support

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import javax.sql.DataSource;
import oracle.jdbc.internal.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;

public class SingleShardTransaction {
       public static void main(String[] args) throws SQLException {
              SingleShardTransaction sample = new SingleShardTransaction();
              DataSource ds = sample.getDataSource();
              // Insert and update the details of following customers in a 
single transaction
              int[] customerIds = new int[] { 100, 101, 102, 103, 104, 105 };

              try (Connection conn = ds.getConnection()) {
                     conn.setAutoCommit(false);
                     for (int id : customerIds) {
                           sample.insertCustomerDetails(conn, id);
                           sample.displayCustomerDetails(conn, id);
                           sample.updateCustomerDetails(conn, id);
                           sample.displayCustomerDetails(conn, id);
                           conn.commit();
                     }

                     System.out.println(((OracleConnection)
conn).getPercentageQueryExecutionOnDirectShard());
              }
       }

       private void insertCustomerDetails(Connection conn, int id) throws 
SQLException {
              String sql = "insert into CUSTOMER values(?, ?, ?, ?)";

              try (PreparedStatement ps = conn.prepareStatement(sql)) {
                     ps.setInt(1, id);
                     ps.setString(2, name);
                     ps.setString(3, email);
                     ps.setString(4, phoneNumber);
                     ps.executeUpdate();
              }
       }

       private void updateCustomerDetails(Connection conn, int id) throws 
SQLException {
              String sql = "UPDATE CUSTOMER SET name = ?, email = ?, 
phoneNumber = ? WHERE customerId = ?";

              try (PreparedStatement ps = conn.prepareStatement(sql)) {
                     ps.setString(1, name);
                     ps.setString(2, email);
                     ps.setString(3, phoneNumber);
                     ps.setInt(4, id);
                     ps.executeUpdate();
              }
       }

       private void displayCustomerDetails(Connection conn, int id) throws 
SQLException {
```

```
             try (PreparedStatement pstmt = conn.prepareStatement("SELECT *
FROM CUSTOMER where ID = ?")) {
                    pstmt.setInt(1, id);

                    try (ResultSet rs = pstmt.executeQuery()) {
                          while (rs.next()) {
                                 // Print the customer details
                          }
                    }
             }
       }

       private DataSource getDataSource() throws SQLException  {
          OracleDataSource ds = new OracleDataSource();
          ds.setURL(<gsmURL>);
          ds.setUser(<userName>);
          ds.setPassword(<password>);
          Properties prop = new Properties();
          // Connection property to enable sharding data source feature

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_USE_SHARDING_DRIVER_CONN
ECTION, "true");

          // Connection property to enable single shard transaction support.
If this property is not set,
          // by default all the transactions are started on catalog DB. When
setting this property value
          // to "true", applications must ensure that all the transactions
span over a single shard only.


prop.setProperty(oracle.jdbc.OracleConnection.CONNECTION_PROPERTY_ALLOW_SINGLE
_SHARD_TRANSACTION_SUPPORT, "true");
          ds.setConnectionProperties(prop);
          return ds;
  }
}
```