19
Result Set

Standard Java Database Connectivity (JDBC) features in Java Development Kit (JDK) include enhancements to result set functionality, such as processing forward or backward, positioning relatively or absolutely, seeing changes to the database made internally or externally, and updating result set data and then copying the changes to the database.

This chapter discusses the following topics:

19.1 Oracle JDBC Implementation Overview for Result Set Support

This section discusses key aspects of the Oracle JDBC implementation of result set support for scrollability, through use of a client-side cache, and for updatability, through use of `ROWID`s.

It is permissible for customers to implement their own client-side caching mechanism, and Oracle provides an interface to use in doing so.

Oracle JDBC Implementation for Result Set Scrollability

Because the underlying server does *not* support scrollable cursors, Oracle JDBC must implement scrollability in a separate layer.

It is important to be aware that this is accomplished by using a client-side memory cache to store rows of a scrollable result set.

> **Note:**
>
> Because all rows of any scrollable result set are stored in the client-side cache, a situation, where the result set contains many rows, many columns, or very large columns, might cause the client-side Java Virtual Machine (JVM) to fail. Do not specify scrollability for a large result set.

Oracle JDBC Implementation for Result Set Updatability

To support updatability, Oracle JDBC uses `ROWID` to uniquely identify database rows that appear in a result set. For every query into an updatable result set, Oracle JDBC driver automatically retrieves the `ROWID` along with the columns you select.

> **Note:**
>
> Client-side caching is not required by updatability in and of itself. In particular, a forward-only updatable result set will not require a client-side cache.

19.2 Resultset Limitations and Downgrade Rules

Some types of result sets are not feasible for certain kinds of queries. If you specify an unfeasible result set type or concurrency type for the query you run, then the JDBC driver follows a set of rules to determine the best feasible types to use instead.

The actual result set type and concurrency type are determined when the statement is run, with the driver issuing a `SQLWarning` on the statement object if the desired result set type or concurrency type is not feasible. The `SQLWarning` object will contain the reason why the requested type was not feasible. Check for warnings to verify whether you received the type of result set that you requested.

Result Set Limitations

The following limitations are placed on queries for enhanced result sets. Failure to follow these guidelines results in the JDBC driver choosing an alternative result set type or concurrency type.

To produce an updatable result set:

- A query can select from only a single table and cannot contain any join operations.

  In addition, for inserts to be feasible, the query must select all non-nullable columns and all columns that do not have a default value.
- A query cannot use `SELECT *`.

  However, there is a workaround for this.
- A query must select table columns only.

  It cannot select derived columns or aggregates, such as the `SUM` or `MAX` of a set of columns.

To produce a scroll-sensitive result set:

- A query cannot use `SELECT *`.

  However, there is a workaround for this.
- A query can select from only a single table.

Scrollable and updatable result sets cannot have any column as `Stream`. When the server has to fetch a `Stream` column, it reduces the fetch size to one and blocks all columns following the `Stream` column until the `Stream` column is read. As a result, columns cannot be fetched in bulk and scrolled through.

Workaround

As a workaround for the `SELECT *` limitation, you can use table aliases, as shown in the following example:

```
SELECT t.* FROM TABLE t ...
```

> **Note:**
>
> There is a simple way to determine if your query will probably produce a scroll-sensitive or updatable result set: If you can legally add a `ROWID` column to the query list, then the query is probably suitable for either a scroll-sensitive or an updatable result set.

Result Set Downgrade Rules

If the specified result set type or concurrency type is not feasible, then Oracle JDBC driver uses the following rules in choosing alternate types:

- If the specified result set type is `TYPE_SCROLL_SENSITIVE`, but the JDBC driver cannot fulfill that request, then the driver attempts a downgrade to `TYPE_SCROLL_INSENSITIVE`.
- If the specified or downgraded result set type is `TYPE_SCROLL_INSENSITIVE`, but the JDBC driver cannot fulfill that request, then the driver attempts a downgrade to `TYPE_FORWARD_ONLY`.
- If the specified concurrency type is `CONCUR_UPDATABLE`, but the JDBC driver cannot fulfill that request, then the JDBC driver attempts a downgrade to `CONCUR_READ_ONLY`.

> **Note:**
>
> Any manipulations of the result set type and concurrency type by the JDBC driver are independent of each other.

Verifying Result Set Type and Concurrency Type

After a query has been run, you can verify the result set type and concurrency type that the JDBC driver actually used, by calling methods on the result set object.

- `int getType() throws SQLException`

  This method returns an `int` value for the result set type used for the query. `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE` are the possible values.

- `int getConcurrency() throws SQLException`

  This method returns an `int` value for the concurrency type used for the query. `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE` are the possible values.

19.3 About Avoiding Update Conflicts

It is important to be aware of the following facts regarding updatable result sets with the JDBC drivers:

- The drivers do not enforce write locks for an updatable result set.
- The drivers do not check for conflicts with a result set `DELETE` or `UPDATE` operation.

A conflict will occur if you try to perform a `DELETE` or `UPDATE` operation on a row updated by another committed transaction.

Oracle JDBC drivers use the `ROWID` to uniquely identify a row in a database table. As long as the `ROWID` is valid when a driver tries to send an `UPDATE` or `DELETE` operation to the database, the operation will be run.

The driver will not report any changes made by another committed transaction. Any conflicts are silently ignored and your changes will overwrite the previous changes.

To avoid such conflicts, use the Oracle `FOR UPDATE` feature when running the query that produces the result set. This will avoid conflicts, but will also prevent simultaneous access to the data. Only a single write lock can be held concurrently on a data item.

19.4 Row Fetch Size

By default, when Oracle JDBC runs a query, it retrieves a result set of 10 rows at a time from the database cursor. This is the default Oracle row fetch size value. You can change the number of rows retrieved with each trip to the database cursor by changing the row fetch size value.

Standard JDBC also enables you to specify the number of rows fetched with each database round-trip for a query, and this number is referred to as the fetch size. In Oracle JDBC, the row-prefetch value is used as the default fetch size in a statement object. Setting the fetch size overrides the row-prefetch setting and affects subsequent queries run through that statement object.

Fetch size is also used in a result set. When the statement object run a query, the fetch size of the statement object is passed to the result set object produced by the query. However, you can also set the fetch size in the result set object to override the statement fetch size that was passed to it.

> **Note:**
>
> Changes made to the fetch size of a statement object after a result set is produced will have no affect on that result set.

The result set fetch size, either set explicitly, or by default equal to the statement fetch size that was passed to it, determines the number of rows that are retrieved in any subsequent trips to the database for that result set. This includes any trips that are still required to complete the original query, as well as any refetching of data into the result set. Data can be refetched, either explicitly or implicitly, to update a scroll-sensitive or scroll-insensitive/updatable result set.

19.4.1 Setting the Fetch Size

The following methods are available in all `Statement`, `PreparedStatement`, `CallableStatement`, and `ResultSet` objects for setting and getting the fetch size:

- `void setFetchSize(int rows) throws SQLException`
- `int getFetchSize() throws SQLException`

To set the fetch size for a query, call `setFetchSize` on the statement object prior to running the query. If you set the fetch size to N, then N rows are fetched with each trip to the database.

After you have run the query, you can call `setFetchSize` on the result set object to override the statement object fetch size that was passed to it. This will affect any subsequent trips to the

database to get more rows for the original query, as well as affecting any later refetching of rows.

19.4.2 Presetting the Fetch Direction

The standard JDBC enables to pre-specify the direction, known as the fetch direction, for use in processing a result set. This allows the JDBC driver to optimize its processing. The following result set methods are specified:

- `void setFetchDirection(int direction) throws SQLException`
- `int getFetchDirection() throws SQLException`

Oracle JDBC drivers support only the forward preset value, which you can specify by entering the `ResultSet.FETCH_FORWARD` static constant value.

The values `ResultSet.FETCH_REVERSE` and `ResultSet.FETCH_UNKNOWN` are not supported. Attempting to specify them causes a SQL warning, and the settings are ignored.

19.5 About Refetching Rows

The result set `refreshRow` method is supported for some types of result sets for refetching data. This consists of going back to the database to re-obtain the database rows that correspond to *n* rows in the result set, starting with the current row, where *n* is the fetch size. This lets you see the latest updates to the database that were made outside of your result set, subject to the isolation level of the enclosing transaction.

Because refetching re-obtains only rows that correspond to rows already in your result set, it does nothing about rows that have been inserted or deleted in the database since the original query. It ignores rows that have been inserted, and rows will remain in your result set even after the corresponding rows have been deleted from the database. When there is an attempt to refetch a row that has been deleted in the database, the corresponding row in the result set will maintain its original values.

> **Note:**
>
> If you declare a `TYPE_SCROLL_SENSITIVE` Result Set based on a query with certain criteria and then externally update the row so that the column values no longer match the query criteria, the driver behaves as if the row has been deleted from the database and the row is not retrieved by the query issued. So, you do not see the updates to the particular row when you call the `refreshRow` method.

Following is the signature of the `refreshRow` method:

```
void refreshRow() throws SQLException
```

You must be at a valid current row when you call this method, not outside the row bounds and not at the insert-row.

The `refreshRow` method is supported for the following result set categories:

- scroll-sensitive/read-only
- scroll-sensitive/updatable
- scroll-insensitive/updatable

> **Note:**
>
> Scroll-sensitive result set functionality is implemented through implicit calls to `refreshRow`.

19.6 About Viewing Database Changes Made Internally and Externally

This section discusses the ability of a result set to view the following:

- Own changes of the result set, referred to as internal changes
- Changes made from elsewhere, either from your own transaction outside the result set, or from other committed transactions, referred to as external changes

> **Note:**
>
> External changes are referred to as other's changes in the standard JDBC specification.

This section covers the following topics:

19.6.1 Visibility versus Detection of External Changes

Regarding the changes made to an underlying database by external sources, there are two similar but distinct concepts with respect to visibility of the changes from your local result set:

- Visibility of changes
- Detection of changes

A "visible" change means that when you look at a row in the result set, you can see new data values from changes made by external sources, to the corresponding row in the database.

A "detected" change, however, means that the result set is aware that this is a new value since the result set was first populated.

Even when an Oracle result set sees new data, as with an external `UPDATE` in a scroll-sensitive result set, it has no awareness that this data has changed since the result set was populated. Such changes are not detected.

19.6.2 Summary of Visibility of Internal and External Changes

Table 19-1 summarizes how a result set object in the Oracle JDBC implementation can see changes made internally through the result set itself, and changes made externally to the underlying database from elsewhere in your transaction or from other committed transactions.

Table 19-1 Visibility of Internal and External Changes for Oracle JDBC

Result Set Type	Can See Internal DELETE?	Can See Internal UPDATE?	Can See Internal INSERT?	Can See External DELETE?	Can See External UPDATE?	Can See External INSERT?
forward-only	no	yes	no	no	no	no
scroll-sensitive	yes	yes	no	no	yes	no
scroll-insensitive	yes	yes	no	no	no	no

Note:

- Remember that explicit use of the `refreshRow` method, is distinct from the concept of visibility of external changes.
- Remember that even when external changes are visible, as with `UPDATE` operations underlying a scroll-sensitive result set, they are not detected. The result set `rowDeleted`, `rowUpdated`, and `rowInserted` methods always return `false`.

19.6.3 Oracle Implementation of Scroll-Sensitive Result Sets

The Oracle implementation of scroll-sensitive result sets involves the concept of a window, with a window size that is based on the fetch size. The window size affects how often rows are updated in the result set.

Once you establish a current row by moving to a specified row, the window consists of the *n* rows in the result set starting with that row, where *n* is the fetch size being used by the result set. Note that there is no current row, and therefore no window, when a result set is first created. The default position is before the first row, which is not a valid current row.

As you move from row to row, the window remains unchanged as long as the current row stays within that window. However, once you move to a new current row outside the window, you redefine the window to be the N rows starting with the new current row.

Whenever the window is redefined, the N rows in the database corresponding to the rows in the new window are automatically refetched through an implicit call to the `refreshRow` method, thereby updating the data throughout the new window.

So external updates are not instantaneously visible in a scroll-sensitive result set. They are only visible after the automatic refetches just described.

> **Note:**
>
> This kind of refetching is not a highly efficient or optimized methodology and it has significant performance concerns. Consider carefully before using scroll-sensitive result sets as currently implemented. There is also a significant trade-off between sensitivity and performance. The most sensitive result set is one with a fetch size of 1, which would result in the new current row being refetched every time you move between rows. However, this would have a significant impact on the performance of your application.