CHAPTER 9

Using SQL in Applications

About this chapter

Previous chapters have described SQL statements as you execute them in Interactive SQL or in some other interactive utility.

When you include SQL statements in an application there are other questions you need to ask. How are query result sets handled in your application? How can you make your application efficient?

While many aspects of database application development depend on your application development tool, database interface, and programming language, there are some common problems and principles. These are discussed in this chapter.

Contents

Executing SQL statements in applications

The way you include SQL statements in your application depends on the application development tool and programming interface you are using. This chapter describes some principles common to most or all interfaces and provides a few pointers for more information. It does not provide a detailed guide for programming using any one interface.

- **ODBC** If you are writing directly to the ODBC programming interface, your SQL statements appear in function calls. For example, the following C function call executes a DELETE statement:

```
SQLExecDirect( stmt,
    "DELETE FROM employee
     WHERE emp_id = 105",
    SQL_NTS );
```

- **JDBC** If you are using the JDBC programming interface, you can execute SQL statements by invoking methods of the **`statement`** object. For example:

```
stmt.executeUpdate(
    "DELETE FROM employee
     WHERE emp_id = 105" );
```

- **Embedded SQL** If you are using Embedded SQL, you prefix your C language SQL statements with the keyword EXEC SQL. The code is then run through a preprocessor before it is compiled. For example:

```
EXEC SQL EXECUTE IMMEDIATE
 'DELETE FROM employee
  WHERE emp_id = 105';
```

- **Sybase Open Client** If you are using the Sybase Open Client interface, your SQL statements appear in function calls. For example, the following pair of calls executes a DELETE statement:

```
ret = ct_command(cmd, CS_LANG_CMD,
                  "DELETE FROM employee
                   WHERE emp_id=105"
                 CS_NULLTERM,
                 CS_UNUSED);
ret = ct_send(cmd);
```

- **Application Development Tools** Application development tools such as the members of the Powersoft PowerStudio family provide their own SQL objects, which use either ODBC (PowerBuilder, Power++) or JDBC (Power J) under the covers.

For more information

For detailed information on how to include SQL in your application, see your development tool documentation. If you are using ODBC or JDBC, consult the software development kit for those interfaces.

For a detailed description of Embedded SQL programming, see "The Embedded SQL Interface" on page 7 of the book *Adaptive Server Anywhere Programming Interfaces Guide*.

Applications inside the server

In many ways, stored procedures and triggers act as applications or parts of applications running inside the server. You can use many of the techniques here in stored procedures also. Stored procedures use statements that are very similar to Embedded SQL.

For information about stored procedures and triggres, see "Using Procedures, Triggers, and Batches" on page 221.

Java classes in the database can use the JDBC interface in just the same way as Java applications outside the server. This chapter discusses some aspects of JDBC. For other information on using JDBC, see "Data Access Using JDBC" on page 503.

Preparing statements

Each time a statement is sent to a database, the server has to carry out several tasks:

- Parse the statement and transforms it into an internal form
- Verify the correctness of all references to database objects, checking that columns named in a query exist, for example.
- If the statement involves joins or subqueries, the query optimizer generates an access plan.
- After all these steps have been carried out, the server can execute the statement.

The steps prior to actually executing a statement are called **preparing** the statement.

Prepared statements can improve performance

If you are using the same statement repeatedly, such as to insert many rows into a table, there is a significant and unnecessary overhead associated with repeatedly preparing the statement.

Some database programming interfaces provide ways of using **prepared statements**. Generally, using these methods requires the following steps:

1. **Prepare the statement** In this step you generally provide the statement with some placeholder character instead of the values.
2. **Repeatedly execute the prepared statement** In this step you supply values to be used each time the statement is executed. The statement does not have to be prepared each time.
3. **Drop the statement** In this step you free the resources associated with the prepared statement. Some programming interfaces handle this step automatically.

Do not prepare statements that are used only once

In general, you should not prepare statements if they are to be executed only once. There is a slight performance penalty for separate preparation and execution, and it introduces an unnecessary complexity into your application.

In some interfaces, however, you do need to prepare a statement in order to associate it with a cursor. For information about cursors, see "Introduction to cursors" on page 205.

The calls for preparing and executing statements are not a part of SQL, and they differ from interface to interface. Each of the Adaptive Server Anywhere programming interfaces provides a method for using prepared statements.

How to use prepared statements

This section provides a brief overview of how to use prepared statements.

❖ **To use a prepared statement:**

1. Prepare the statement.
2. Set up **bound parameters**, which will be used to hold values in the statement.
3. Assign values to the bound parameters in the statement.
4. Execute the statement.
5. Repeat steps 3 and 4 as needed.
6. Drop the statement when finished. This step is not required in JDBC, as Java's garbage collection mechanisms handle the problem for you.

The general procedure is the same, but the details vary from interface to interface. Comparing how you use prepared statements in different interfaces illustrates this point.

Using prepared statements in ODBC

❖ **To use a prepared statement in ODBC:**

1. Prepare the statement using **`SQLPrepare`**.
2. Bind statement parameters using **`SQLBindParameter`**.
3. Execute the statement using **`SQLExecute`**.
4. Drop the statement using **`SQLFreeStmt`**.

☞ For more information, see "Using prepared statements in ODBC" on page 133 of the book *Adaptive Server Anywhere Programming Interfaces Guide* and the ODBC SDK documentation.

To use a prepared statement with JDBC

You can use prepared statements with JDBC both from a client application and inside the server.

❖ **To use a prepared statement in JDBC:**

1. Prepare the statement using the **`prepareStatement`** method of the connection object. This returns a prepared statement object.
2. Set statement parameters using the appropriate **`set`***Type* methods of the prepared statement object. Here, *Type* is the data type being assigned.
3. Execute the statement using the appropriate method of the prepared statement object. For inserts, updates, and deletes this is the **`executeUpdate`** method.

For more information on using prepared statements in JDBC, see "Using prepared statements for more efficient access" on page 520.

To use a prepared statement with Sybase Open Client

❖ **To use a prepared statement in Open Client:**

1. Prepare the statement using the **`ct_dynamic`** function, with a CS_PREPARE type parameter.
2. Set statement parameters using **`ct_param`**.
3. Execute the statement using **`ct_dynamic`** with a CS_EXECUTE type parameter.
4. Free the resources associated with the statement using **`ct_dynamic`** with a CS_DEALLOC type parameter.

For more information on using prepared statements in Open Client, see "Using SQL in Open Client applications" on page 149 of the book *Adaptive Server Anywhere Programming Interfaces Guide*.

Introduction to cursors

When you execute a query in an application, the result set consists of a number of rows. In general, you do not know how many rows you are going to receive before the query is executed. **Cursors** provide a way of handling query result sets in applications.

The way you actually use cursors, and the kinds of cursor available to you, depends on the programming interface you are using. JDBC 1.0 does not provide more than rudimentary handling of result sets, while ODBC and Embedded SQL have many different kinds of cursor. Open Client cursors are limited to moving forward through a result set.

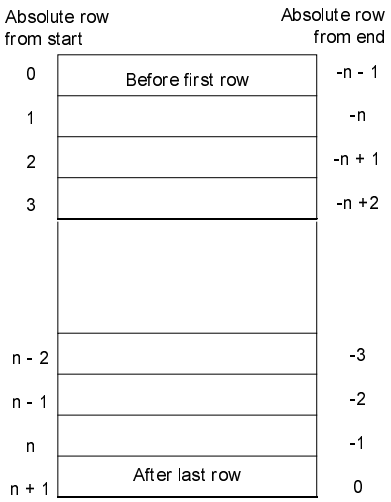
For information on the kinds of cursor available through different programming interfaces, see "Availability of cursors" on page 208.

What is a cursor?

A cursor is a symbolic name that is associated with a SELECT statement or stored procedure that returns a result set. It consists of the following parts:

- **Cursor result set** The set of rows resulting from the execution of a query that is associated with the cursor
- **Cursor position** A pointer to one row within the cursor result set

You can think of a cursor as a **handle** on the result set of a SELECT statement. It enables you to examine and possibly manipulate one row at a time. In Adaptive Server Anywhere, cursors support forward and backward movement through the query results.



What you can do with cursors

With cursors, you can do the following:

- Loop over the results of a query.
- Carry out inserts, updates, and deletes at any point within a result set.
- Some programming interfaces allow you to use special features to tune the way in which result sets are returned to your application. This can provide substantial performance benefits for your application

Steps in using a cursor

The steps in using a cursor in Embedded SQL are different from in other interfaces.

❖ **To use a cursor in ODBC or Open Client:**

1 **Execute a statement** Execute a statement using the usual method for the interface. You can prepare and then execute the statement, or you can execute the statement directly.

2 **Test to see if the statement returns a result set** A cursor is implicitly opened when a statement that creates a result set is executed. When the cursor is opened, it is positioned before the first row of the result set.

3 **Fetch results** A simple fetch operations moves the cursor to the next row in the result set. Adaptive Server Anywhere permits more complicated movement around the result set using fetches. Which of these you can use depends on the programming interface you are using: not all interfaces support advanced cursor-handling operations.

4 **Close the cursor** When you have finished with the cursor, you close it to free resources associated with it.

5 **Free the statement** If you used a prepared statement, free it to reclaim memory.

❖ **To use a cursor in Embedded SQL:**

1 **Prepare a statement** Cursors generally use a statement handle rather than a string. You need to prepare a statement in order to have a handle available.

2 **Declare the cursor** Each cursor refers to a single SELECT statement. When you declare a cursor, you state the name of the cursor and the statement it refers to.

3 **Open the cursor** Opening the cursor executes the query up to the point where the first row is about to be obtained.

4 **Fetch results** A simple fetch operations moves the cursor to the next row in the result set. Adaptive Server Anywhere permits more complicated movement around the result set using fetches. Which of these you can use depends on the programming interface you are using: not all interfaces support advanced cursor-handling operations.

5 **Close the cursor** When you have finished with the cursor, you close it.

6 **Reclaim memory** To free the memory associated with the cursor and its associated statement you need to free the statement.

Prefetching rows

In some cases, the interface library may carry out performance optimizations under the covers (such as prefetching results) so that these steps in the client application may not correspond exactly to software operations:

Types of cursor

You can use one of several kinds of cursor in Adaptive Server Anywhere. You must choose from among these cursor types when you declare the cursor.

- **Unique cursors** When a cursor is declared unique, the query is forced to return all the columns required to uniquely identify each row. Often this means ensuring that all the columns in the primary key are returned. Any columns required but not specified are added to the result set.
- **Read only cursors** A cursor declared as read only may not be used in an UPDATE (positioned) or a DELETE (positioned) operation.
- **No scroll cursors** When a cursor is declared NO SCROLL, fetching operations are restricted to fetching the next row or the same row again.
- **Dynamic scroll cursors** With dynamic scroll cursors you can carry out more flexible fetching operations. You can move backwards and forwards in the result set, or move to an absolute position.
- **Scroll cursors** These are similar to dynamic scroll cursors, but behave differently when the rows in the cursor are modified or deleted after the first time the row is read. Scroll cursors have more predictable behavior when changes happen.
- **Insensitive cursors** Also called **static** cursors in ODBC.

  A cursor declared **insensitive** has its membership fixed when it is opened; a temporary table is created with a copy of all the original rows. Fetching from an insensitive cursor does not see the effect of any other operation from a different cursor. It does see the effect of operations on the same cursor. Also, insensitive cursors are not affected by ROLLBACK or ROLLBACK TO SAVEPOINT; these are not operations on the cursor that change the cursor contents.

  It is easier to write an application using insensitive cursors, since you only have to worry about changes you make explicitly to the cursor. You do not have to worry about actions taken by other users or by other parts of your application.

  Insensitive cursors can be expensive if the cursor is on many rows.

Availability of cursors

Not all interfaces provide support for all kinds of cursors.

- JDBC does not use cursors, although the **`ResultSet`** object does have a **`next`** method that allows you to scroll through the results of a query in the client application.
- ODBC supports all the kinds of cursors.

  ODBC provides a cursor type called a **block cursor**. When you use a block cursor, you can use **`SQLFetch`** or **`SQLExtendedFetch`** to fetch a block of rows, rather than a single row.
- Embedded SQL supports all available cursors.
- Sybase Open Client does not support scrollable (Scroll or Dynamic Scroll) cursors. Also, using updateable cursors that are not unique has a severe performance penalty.

Choosing a cursor type

Each row fetched in a scroll cursor is remembered. If one of these rows is deleted, either by your program or by another connection, it creates a "hole" in the cursor. If you fetch the row at this "hole" with a SCROLL cursor, an error is returned indicating that there is no current row, and the cursor is left positioned on the "hole". In contrast, a dynamic scroll cursor just skips the "hole" and retrieves the next row. Scroll cursors remember row positions within a cursor, so that your application can be assured that these positions will not change.

Example

For example, an application could remember that **Cobb** is the second row in the cursor for the following query:

```
SELECT emp_lname
FROM employee
```

If the first employee (Whitney) is deleted while the scroll cursor is still open, a FETCH ABSOLUTE 2 will still position on Cobb while FETCH ABSOLUTE 1 will return an error. Similarly, if the cursor is on Cobb, FETCH PREVIOUS will return SQLE_NO_CURRENT_ROW.

In addition, a fetch on a SCROLL cursor returns a warning if the row has changed since it was last read. The warning only happens once; fetching the same row a third time will not produce the warning.

Similarly, an UPDATE (positioned) or DELETE (positioned) statement on a row that has been modified since it was last fetched returns an error. An application must fetch the row again before the UPDATE or DELETE is permitted.

An update to any column will cause the warning/error, even if the column is not referenced by the cursor. For example, a cursor on a query returning **emp_lname** would report the update even if only the **salary** column were modified.

No warnings or errors in bulk operations mode
These update warning and error conditions do not occur in bulk operations mode (`-b` database server command-line switch).

More information is maintained about scroll cursors than dynamic scroll cursors. Dynamic scroll cursors are therefore more efficient and should be used unless the consistent behavior of scroll cursors is required.

There is no extra overhead for dynamic scroll cursors versus no scroll cursors.

Working with cursors

This section describes how to carry out different kinds of operation using cursors.

Configuring cursors on opening

You can configure the following aspects of cursor behavior when you open the cursor:

- **Isolation level** You can set the isolation level of operations on a cursor explicitly, to be different from the current isolation level of the transaction.
- **Holding** Unless you explicitly require that a cursor be kept open, cursors are closed at the end of a transaction. Opening a cursor **with hold** allows you to keep it open until the end of a connection.

Fetching rows through a cursor

The simplest way of processing the result set of a query using a cursor is to loop through all the rows of the result set until there are no more rows. The steps in this process are as follows:

1 Declare and open the cursor (Embedded SQL), or execute a statement that returns a result set.

2 Fetch the next row until you get a Row Not Found error.

3 Close the cursor.

The way these operations are carried out depends on the interface you are using. For example:

- In ODBC **`SQLFetch`** advances the cursor to the next row and returns the data.

  For information on using cursors in ODBC, see "Working with result sets" on page 135 of the book *Adaptive Server Anywhere Programming Interfaces Guide*.

- In JDBC, the **`next`** method of the **`ResultSet`** object advances the cursor and returns the data.

  For information on using the **`ResultSet`** object in JDBC, see "Queries using JDBC" on page 519.

- In Embedded SQL, the FETCH statement carries out the same operation.

  For information on using cursors in Embedded SQL, see "Cursors in Embedded SQL" on page 33 of the book *Adaptive Server Anywhere Programming Interfaces Guide*.

- In Open Client, **`ct_fetch`** advances the cursor to the next row and returns the data.

  For information on using cursors in Open Client applications, see "Using cursors" on page 150 of the book *Adaptive Server Anywhere Programming Interfaces Guide*.

Fetching multiple rows

This section discusses how you can use fetching of multiple rows at a time: a technique that can improve performance.

Multiple-row fetches

Some interfaces provide methods for fetching more than one row at a time into the next several fields in an array. In general, the fewer separate fetch operations you can execute, the fewer individual requests the server must respond to, and the better the performance. Multiple-row fetches are also sometimes called **wide fetches**. Cursors that use multiple-row fetches are sometimes called **block cursors** or **fat cursors**.

Multiple-row fetching should not be confused with **prefetching** rows. Multiple row fetches read the next several rows of the cursor into the application at one time.

Using multiple-row fetching

- In ODBC, if you are using a block cursor, several rows are fetched by default whenever you do a **`SQLFetch`**.
- In ODBC, **`SQLFetchScroll`** (**`SQLExtendedFetch`** prior to ODBC 3.0) permits multiple rows to be fetched in a single call. **`SQLFetchScroll`** provides this control at fetch time, in contrast to block cursors, which provide this feature when the cursor is declared.
- In Embedded SQL, the FETCH statement provides control over the number of rows fetched at a time, by providing an ARRAY clause.
- Open Client and JDBC do not support multi-row fetches.

Prefetching rows

Prefetches are different from multiple-row fetches. Prefetches can be carried out without explicit instructions from the client application. Prefetching retrieves rows from the server into a buffer on the client side, but does not make those rows available to the client application until the appropriate row is fetched by the application.

By default, the Adaptive Server Anywhere client library prefetches multiple rows whenever a single row is fetched by an application. The additional rows are stored in a buffer by the Adaptive Server Anywhere client library.

Prefetching assists performance by cutting down on client/server traffic, and increases throughput by making many rows available without a separate request to the server.

For information on controlling prefetches, see "PREFETCH option" on page 169 of the book *Adaptive Server Anywhere Reference Manual*.

Controlling prefetching from an appliation

- You can control whether or not prefetching occurs using the PREFETCH option. This can be set for a single connection to ON or OFF. By default it is set to ON. The number of rows fetched at a time is determined by the server.
- In Embedded SQL, you can control prefetching when you open a cursor and on a FETCH operation, by using the BLOCK clause. (This should not be confused with ODBC block cursors, which fetch blocks of rows into an application.)

  The application can specify a maximum number of rows that should be contained in a single fetch from the server by specifying the BLOCK clause. For example, if you are fetching and displaying 5 rows at a time, you could use BLOCK 5. Specifying BLOCK 0 causes 1 record at a time to be fetched and also cause a FETCH RELATIVE 0 to always fetch the row again.
- In Open Client, you can control prefetching behavior using **`ct_cursor`** with CS_CURSOR_ROWS after the cursor is declared, but before it is opened.

Fetching with scrollable cursors

ODBC and Embedded SQL provide methods for using scrollable and dynamic cursors. These methods allow you to move several rows forward at a time, or to move backwards through the result set.

Scrollable cursors are not supported by the JDBC or Open Client interfaces.

Prefetching does not apply to scrollable operations. That is, if you fetch a row before the current row, you do not get several previous rows prefetched also.

Modifying rows through a cursor

Cursors are not only used for reading result sets from a query. You can also modify data in the database while processing a cursor. These operations are commonly called **positioned** update and delete operations, or **put** operations if the action is an insert.

Not all query result sets allow positioned updates and deletes. If you carry out a query on a non-updateable view then no changes can be made to the underlying tables. Also, if the query involves a join then you must specify which table you wish to delete from, or which columns you wish to update, when you carry out the operations.

Insertions through a cursor can only be executed if any non-inserted columns in the table allow NULL or have defaults.

ODBC, Embedded SQL, and Open Client permit data modification using cursors, but JDBC does not. With Open Client, you can delete and update rows, but you can only insert rows on a single-table query.

Which table are rows deleted from?

If you attempt a positioned delete on a cursor, the table from which rows are deleted is determined as follows:

1. If no FROM clause is included, the cursor must be on a single table only.
2. If the cursor is for a joined query (including using a view containing a join), then the FROM clause must be used. Only the current row of the specified table is deleted. The other tables involved in the join are not affected.
3. If a FROM clause is included, and no table owner is specified, the table-spec value is first matched against any correlation names.
4. If a correlation name exists, the table-spec value is identified with the correlation name.
5. If a correlation name does not exist, the table-spec value must be unambiguously identifiable as a table name in the cursor.
6. If a FROM clause is included, and a table owner is specified, the table-spec value must be unambiguously identifiable as a table name in the cursor.
7. The positioned DELETE statement can be used on a cursor open on a view as long as the view is updateable.

Canceling cursor operations

You can cancel a request through an interface function. From Interactive SQL, you can cancel a request by pressing STOP.

If you cancel a request that is carrying out a cursor operation, the position of the cursor is indeterminate. After canceling the request, you must locate the cursor by its absolute position, or close it, following the cancel.

Bookmarks and cursors

ODBC provides **bookmarks**, which are values used to identify rows in a cursor. Adaptive Server Anywhere supports bookmarks for all kinds of cursor except dynamic cursors.

Before ODBC 3.0, a database could specify only whether it supports bookmarks or not. There was no way for a database server to indicate for what kind of cursor bookmarks were supported. Adaptive Server Anywhere returns that it does support bookmarks. There is therefore nothing in ODBC to prevent you from trying to use bookmarks with dynamic cursors; however, you should not use this combination.

Describing result sets

Some applications build SQL statements, perhaps in response to the user's actions, which cannot be completely specified in the application. For example, a reporting application may allow a user to select which columns they wish to display.

In such a case, the application needs a method for retrieving information about the nature of the result set itself (the number and type of columns), as well as the contents of the result set. This **resultset metadata** information is manipulated using **descriptors**. Obtaining and managing the result set metadata is called **describing**.

As result sets are generally obtained from cursors, descriptors and cursors are closely linked.

In some interfaces, use of descriptors is hidden from the user.

A sequence for using a descriptor with a cursor-based operation is as follows:

1 Allocate the descriptor. This may be done implicitly, although explicit allocation is allowed in some interfaces.

2 Prepare the statement.

3 Declare and open a cursor for the statement (Embedded SQL) or execute the statement.

4 Get the descriptor and modify the allocated area if necessary. This is often done implicitly.

5 Fetch and process the statement results.

6 Deallocate the descriptor.

7 Close the cursor.

8 Drop the statement. This is done automatically by some interfaces.

Typically, statements that need descriptors are either SELECT statements or stored procedures that return result sets.

The data structure that holds the information concerning the expected number and type of columns that are being returned is called a **descriptor**. In different interfaces, the descriptor may be implemented in different ways.

Implementation notes

- In Embedded SQL, a SQLDA (SQL Descriptor Area) structure holds the descriptor information.

☞ For more information, see "The SQL descriptor area (SQLDA)" on page 45 of the book *Adaptive Server Anywhere Programming Interfaces Guide*.

- In ODBC, a descriptor handle allocated using **SQLAllocHandle** provides access to the fields of a descriptor. You can manipulate these fields using **SQLSetStmtAttr**, **SQLSetDescField**, **SQLGetStmtAttr**, and **SQLGetDescField**.

  Alternatively, you can use **SQLDescribeCol** and **SQLColAttributes** to obtain column information.

- In Open Client, you can use **ct_dynamic** to prepare a statement and **ct_describe** to describe the result set of the statemend. However, you can also use **ct_command** to send a SQL statement without preparing it first, and use **ct_results** to handle the returned rows one by one. This is the more common way of operating in Open Client application development.

- In JDBC, the **java.sql.ResultSetMetaData** class provides information about result sets.

Controlling transactions in applications

Transactions are sets of SQL statements that are atomic. Either all the statements in the transaction are executed, or none are.

For information about transactions, see "Using Transactions and Locks" on page 367.

This section describes a few aspects of transactions in applications.

Setting autocommit or manual commit mode

Some database programming interfaces have an **autocommit mode**, also called **unchained mode**. In this mode, each statement is a transaction, and is committed after execution. If you wish to use transactions in your applications, you need to be using **manual commit mode**, or **chained mode**.

The performance and behavior of your application may change, depending on whether you are running in an autocommit mode. Autocommit is not recommended for most purposes.

You can control autocommit behavior in the database using the CHAINED database option. You can also control autocommit behavior in some database interfaces by setting an autocommit interface option.

Using the CHAINED database option

You can set the current connection to operate in autocommit mode by setting the CHAINED database option to OFF.

By default, CHAINED is set to ON in Adaptive Server Anywhere (manual commit mode).

Setting autocommit mode

- **ODBC** By default, ODBC operates in autocommit mode. You can turn off this mode using the SQL_ATTR_AUTOCOMMIT connection attribute. ODBC autocommit is independent of the CHAINED option.
- **JDBC** By default, JDBC operates in autocommit mode. You can turn off this mode by using the **`setAutoCommit`** method of the connection object:

  ```
  conn.setAutoCommit( false );
  ```

- **Embedded SQL** Embedded SQL uses the setting of the user's CHAINED option to govern the transaction behavior. By default, this option is set to ON (manual commit).
- **Open Client** A connection made through Open Client sets the mode to autocommit by default. You can change this behavior by setting the CHAINED database option to ON.

Controlling the isolation level

The isolation level of a current connection can be set using the ISOLATION_LEVEL database option.

Some interfaces, such as ODBC, allow you to set the isolation level for a connection at connection time. This level can be reset later using the ISOLATION_LEVEL database option.

Cursors and transactions

In general, a cursor is closed when a COMMIT is performed. There are two exceptions to this behavior:

- The CLOSE_ON_ENDTRANS database option is set to OFF.
- A cursor is opened WITH HOLD.

If either of these two cases is true, the cursor is not closed on a COMMIT.

ROLLBACK and cursors

If a transaction is rolled back, then cursors are closed except for those cursors opened WITH HOLD. However, the contents of any cursor after a rollback should not be relied on.

The draft ISO SQL3 standard states that on a rollback, all cursors should be closed. You can obtain this behavior by setting the ANSI_CLOSE_CURSORS_AT_ROLLBACK option to ON.

Savepoints

If a transaction is rolled back to a savepoint, and if ANSI_CLOSE_CURSORS_AT_ROLLBACK option is set to ON, then all cursors opened after the SAVEPOINT are closed.

Cursors and isolation levels

You can change the isolation level of a connection during a transaction using the SET OPTION statement to alter the ISOLATION_LEVEL option. However, this change does not affect any cursor that is already opened.