IBM

IT infrastructure Power Systems Software IBM i DB2

# IBM DB2 for i: Tips

## SQL CLI frequently asked questions



### Is CLI the same as ODBC?

For the most part, the SQL CLI is syntactically and semantically equivalent to ODBC. This is not to say that everything in the latest level of ODBC is supported, but an application that uses the most common routines will likely run against the CLI with few changes.

The SQL Call Level Interface is a standard created by the X/Open standards group, and our CLI was built in V3R6 according to that standard. ODBC is the Microsoft implementation of the X/Open CLI standard, and has veered off from the X/Open standard in a few areas. So, even though the CLI has been updated in several releases since V3R6, it will likely never match ODBC exactly, since that is technically not the standard that our CLI is built to comply with.

Back to top

### What is CLI server mode?

Server mode was introduced in V4R2 in a functional PTF. The reason for running in SQL server mode is that many applications have the need to act as database servers. This means that a single job will execute SQL requests on behalf of multiple users. In our current SQL implementation, this does not work

for 3 reasons:

1) A single job can only have one commit transaction per activation group

2) A single job can only connect to an RDB once

3) All SQL statements run under the job's user profile, regardless of the userid passed in on the connect

SQL server mode will solve these 3 problems by routing all SQL statements to a separate job. The job will be taken from a pool of prestart jobs running in the QSYSWRK subsystem. There will be one job per connection, when running in server mode. The mechanism for "waking up" a prestart job includes the passing of a user profile, which causes the prestart job to swap from profile QUSER to the profile specified on the SQLConnect. If no user profile is specified on the SQLConnect, then the prestart job runs in the same user profile as the originating job. Since each connection is a separate job, there is no limitation on the number of connections to each database, and each job will have its own commit transaction.

Since the SQL is processed in a separate job, there will be no error messages sent from runtime SQL modules to your joblog. However, there is a new SQL7908 message sent to your joblog, which gives you the full job name of the prestart job that is being used to handle the SQL requests. This message will be sent for each successful SQLConnect performed while in server mode. Starting with V4R3, as a debugging aid, if any SQL error occurs, the joblog will be printed for the prestart job. Then using WRKSPLF for the effective user profile of the connection, one can find the relevant joblog.

Using SQL server mode also has another benefit. Since the SQL is processed in another job, the amount of shared resources across connections is drastically reduced. Simply put, more work can be done all at once, since they aren't all working with structures and memory in the same process. This allows threaded applications that use a different connection handle per thread to get much better performance, and more effectively exploit threads with SQL.

Read more about CLI Server mode in *NEWS/400*: "Using the SQL Call Level Interface as an SQL Server" by Mark Megerian.

**How do I turn on server mode?**

Here is a simple example of turning on server mode in a CLI program:

```
.
.
SQLHENV henv;
SQLHDBC hdbc;
SQLINTEGER attr;
.
.
SQLAllocEnv(&henv);
attr = SQL_TRUE;
SQLSetEnvAttr(henv,SQL_ATTR_SERVER_MODE,
&attr,0);
SQLAllocConnect(henv,&hdbc);
SQLConnect(hdbc,"SYSTEM1",SQL_NTS,"FRED",
SQL_NTS,"NEWPASS",SQL_NTS);
.
.
```

**Benefits when running in server mode**

There are 40,000 total handles allowed in server mode, compared to 500 total handles when not using server mode.

The SQL statements are processed under the profile specified on the SQLConnect.

Each connection can be committed or rolled back independently. This could also be considered a limitation, since some applications may want to commit or roll back multiple connections simultaneously, and would therefore not be able to use server mode.

There can be multiple connections to the same relational database. If not using server mode, this would be an error condition.

Threaded applications can run multiple threads simultaneously, provided each thread is working within its own connection handle.

**Restrictions when running in server mode**

A job must set Server Mode on at the very beginning of processing, before doing anything else. The SQLSetEnvAttr call to turn on server mode must be done right after SQLAllocEnv, and before any other calls. Once its on, it cannot be turned off.

The SQL all runs in the prestart jobs, as well as any commitment control. Commitment control must NOT be started in the originating job, either before or after entering server mode.

When running server mode, the application MUST use SQL commits and rollbacks, either embedded or via the SQL CLI. They cannot use the CL commands, since there is no commitment control running in the originating job. The job MUST issue a COMMIT before disconnecting, otherwise an implicit ROLLBACK will occur.

Interactive SQL cannot be used from a job in server mode. Use of STRSQL when in server mode will result in an SQL6141 message.

SQLDataSources is unique in that it does not require a connection handle to run. When in server mode, the program must already have done a connect to local, before using SQLDataSources. Since DataSources is used to find the name of the RDB to connect to, we will support passing a NULL pointer for the RDB name on SQLConnect in order to obtain a local connection. This makes it possible for a generic program to be written, when there is no prior knowledge of the system names.

When doing commits and rollbacks via the CLI, the calls to SQLEndTran and SQLTransact must include a connection handle. When not running in server mode, one can omit the connection handle to commit everything, but this is not supported in server mode since each connection (or thread) has its own transaction scoping.

Statement handles must not be shared across threads. If threads are being used with Server Mode, each thread should allocate its own statement handles, or the application should have synch points to prevent multiple threads from working with the same handle at the same time. This is because

one thread could overwrite return data or error information that another thread has yet to process.

---

### Is there support for an exit program?

Yes. When running in server mode, it can be more difficult to set up debug or adjust other job attributes, since the work is done in a prestart job. To make this easier, support was added in V4R5 for a connect exit program. This means that you can have your own program invoked from the prestart job, and set whatever you want to set in that job, prior to it performing any SQL work.

The CLI Connection exit program is called by CLI through the registration facility. The exit program is called before the connection is made to the relational database. CLI must be running in server mode for the exit program to be called. The exit point supports one CLI Connection exit program at a time. This exit program can be used for such things as changing the library list depending on the user making the connection or to enable debug for the prestart job handling the SQL requests. The only parameter passed to the exit program is a CHAR(10) user profile. This allows you to control the actions of the exit program based on the user profile. For instance, if you wanted to run in debug mode, but did not want to affect other users of server mode on the system, you could register an exit program, and have that program only start debug after it determined that your user profile was passed as the parameter.

To register an exit program, use the command WRKREGINF

Required Parameter Group:

1
Connection user profile
Input
CHAR(10)

```
QSYSINC Member Name:  None
Exit Point Name:  QIBM_QSQ_CLI_CONNECT
Exit Point Format Name:  CLIC0100
```

---

### What about teraspace and server mode?

Due to customer requests, the CLI will support teraspace and server mode together, beginning in V5R1. The nature of teraspace storage is that pointers to teraspace data cannot be passed across jobs, since the references are only valid for the job owning the storage. Since server mode is based on jobs passing pointers to each other, CLI programs using teraspace could not run in server mode.

In V5R1, an enhancement has been made to allow the server job to have visibility to the originating job's teraspace, therefore making the pointers valid within the server job. This enhancement allows programs compiled to use TERASPACE to run using SQL server mode.

## Where do I find information on SQLSTATE values?

The first thing to understand is the relationship between the SQLSTATE and the SQLCODE. Every error has an SQLSTATE, which is a platform-independent error state value. SQLCODE is an error value that is returned by the CLI on SQLError as the native error. The SQLCODE error value is not guaranteed to be standard across platforms, but the SQLSTATE value is.

There is more SQLSTATE information in the tech studio. To find the SQLSTATE information, first click on CLI Programs, and then Debugging CLI.

A listing of SQLSTATEs and SQLCODEs is provided in appendix B of the manual

```
SC41-4611   DB2 for i SQL Programming
```

So, if you use SQLSTATE values, use either the manual or the web site indicated above. If you use SQLCODE, then either use the manual, or simply display the message description on your IBM i or AS/400 using the DSPMSGD CL command. Every SQLCODE corresponds to a message on the system. For instance, to display the details of SQLCODE -204, you could do:

```
DSPMSGD RANGE(SQL0204) MSGF(QSQLMSG)
```

## Can the CLI be use in threaded applications?

The short answer to this question is Yes, meaning that the CLI code is thread safe, and you do not have to worry about putting synchronization points in your own code. The CLI implementation has its own mutexing logic to ensure that processing is effectively single-threaded at points in which its needed.

However, this is not to say that multiple threads will be able to simultaneously process SQL requests using the CLI. The only way to get any true multi-threading of SQL requests is to use CLI Server Mode.

## How many CLI handles can I allocate?

The CLI supports up to 500 total handles per job. This includes all types of handles, including the environment handle, connection handles, statement handles, and descriptor handles. When running in CLI Server Mode the limit becomes 40,000 total handles, with up to 500 statement handles per connection.

In version 5, this changes significantly. The total limit becomes 80,000 handles regardless of server mode. The total limit of 500 goes away completely. A remote connection may still only have 500 statement handles.

Be aware that the CLI must internally allocate descriptor handles in some cases. So, for instance, if you allocate a statement handle, then bind variables to that handle, you may get up to 4 descriptor handles allocated for you. These count against the 500 handle limit. The four descriptor handles are the two PARAMETER descriptor handles, and the two ROW descriptor handles. The reason that there are two of each is that the CLI must separately maintain a default descriptor (this is known as the implementation descriptor) as well as a

descriptor to keep track of the variables that have been explicitly bound by the application (this is known as the application descriptor).

So, in all, a single statement handle can have an implementation row descriptor, an application row descriptor, an implementation parameter descriptor, and an application parameter descriptor. To better understand these, think of a prepared statement that looks like this:

```
SELECT  NAME, AGE, SEX FROM PERSONNEL WHERE AGE > ?
```

After the call to SQLPrepare, the implementation row descriptor would contain entries for ordinal positions 1, 2, and 3, and these would describe the data types and lengths of columns NAME, AGE, and SEX in the PERSONNEL table. So, the entry for NAME may be character of length 20. Then the application does an SQLBindCol and indicates the value for column 1 will be bound to a character variable of length 10, which is perfectly legal. Now the application row descriptor has one entry in it (the implementation row descriptor is not affected by the SQLBindCol).

Similarly, after the SQLPrepare, the implementation parameter descriptor has one entry, containing the type and length of the single input parameter, based on column AGE, which is a 4-byte integer. The application then uses SQLBindParameter to say that this value will actually be passed in using the float data type. This information will be stored in the application parameter descriptor.

---

### How can I access stored procedures with result sets?

The SQL CLI fully supports stored procedures with result sets. The easiest example is the stored procedure with a single result set. In the following example, it is assumed that the stored procedure has already been created and registered on the system. This would be accomplished using the CREATE PROCEDURE statement in SQL.

```
.
.
SQLExecDirect(hstmt,"CALL MYLIB.PROC1",SQL_NTS);
SQLBindCol(hstmt,1,SQL_INTEGER,&intval1,4,&fetchlen1);
SQLBindCol(hstmt,2,SQL_INTEGER,&intval2,4,&fetchlen2);
SQLBindCol(hstmt,3,SQL_CHAR,charval,10,&fetchlen3);
rtnc=SQLFetch(hstmt);
for(;rtnc==0;)
{
printf("COLUMN 1 is %i\n",intval1);
printf("COLUMN 2 is %i\n",intval2);
printf("COLUMN 3 is %s\n",charval);
rtnc=SQLFetch(hstmt);
}
SQLCloseCursor(hstmt);
.
.
```

In the preceding example, the stored procedure leaves open a result set that the CLI code can fetch data from. This becomes a true CLI result set, in the sense that SQLFetch can be used to retrieve the data, and SQLCloseCursor must be use to close out the result set.

The example gets a bit more complicated when there is the possibility of more

than one result set for a stored procedure. This requires the use of SQLMoreResults to process through the multiple result sets. Also, the preceding example was built with the assumption that the result set contained 3 columns, and that the first 2 columns were integers and the 3rd column was a character. The next example will check the number of result set columns and bind each column to an integer variable.

```
.
.
results=SQLExecDirect(hstmt,"CALL MYLIB.PROC1",SQL_NTS);
for(;results==0;)
{
SQLNumResultCols(hstmt,&cols);
for(colposition=1;colposition <= cols;colposition++)
  {
   SQLBindCol(hstmt,colposition,SQL_INTEGER,&
intval[colposition-1],4,&fetchlen[colposition-1]);
  }
rtnc=SQLFetch(hstmt);
for(;rtnc==0;)
  {
   for(colposition=1;colposition <= cols;colposition++)
        {
         printf("COLUMN %i is %i\n",colposition,intval
                [colposition-1]);
        }
   rtnc=SQLFetch(hstmt);
  }
/* The current result set has been fully read,
move on to the next one. */
results=SQLMoreResults(hstmt);
}
.
.
```

As with all stored procedures, make sure that the activation group is set to *CALLER for the program object. This is automatic for SQL stored procedures. For non-SQL stored procedures, if the activation group is *NEW or a named activation group, your stored procedure will not work properly. Result sets would be closed by activation group cleanup, and any changes made under commitment control in the stored procedure could not be committed or rolled back by the CLI application invoking the stored procedure.

---

### Why are bound columns better than unbound columns?

This is a very important point for performance. There are many ways to generate a result set using the CLI. One can execute a SELECT statement; one can use the catalog functions, such as SQLTables and SQLColumns; one can execute a CALL statement to a stored procedure that has one or more result sets. When the application is ready to retrieve the data from the result set, it can bind each of the columns to a variable in the application and start using SQLFetch to process each row. This is the recommended approach. For each column in the result set, one should use SQLBindCol to bind each and every column. The SQLNumResultCols procedure can be used to determine how many columns are in the result set, and either SQLColAttributes or SQLDescribeCol can be used to determine the data type of each column. Going through this process results in "bound" columns. This means that when

the SQLFetch is processed, the application has already done SQLBindCol for every column, so the CLI knows the address to store the value for each column in the row being fetched.

The alternative is "unbound" columns, and this is NOT recommended. Using unbound columns means that the user has not used SQLBindCol to bind each column to an address. Then when SQLFetch is processed, the CLI is forced to store each value in a temporary location of storage. Then after the SQLFetch, the application must use SQLGetData (or SQLGetCol) to retrieve each column's value from the temporary location.

Unbound columns are worse than bound columns because:

- The performance is significantly worse since the data has to be moved multiple times. First it must be moved from the database into the temporary storage location. Then it must be moved from the temporary storage location into the application's variables on the SQLGetData.
- System resources are used more heavily for unbound columns since the CLI must dynamically allocate storage for every unbound column. If enough unbound columns are used, and the columns are very large, it is possible to use up the storage allocated to the process, and the next action that requires storage will fail.
- Code complexity is worse with unbound fetches, since after every fetch, the code must perform an SQLGetData for every column in the result set. If each column was bound using SQLBindCol, then after each fetch, the data would already be in local storage, and no additional CLI calls would be required.

Therefore, we strongly recommend finding any places in your code where SQLGetData or SQLGetCol is used after the SQLFetch, and replacing this with SQLBindCol before the SQLFetch. Also note that your performance can be improved by omitting unneeded columns from your result set. Sometimes the shorthand notation "SELECT * FROM ... " is used when only a subset of the columns are really used. In those cases, change the SELECT statement to explicitly name each column that is truly required.

---

### Must I always journal my files?

No. Files only need to be journaled when the user is running with commitment control, and will possibly be making database updates. The reason that journaling is needed in this case, is that when commitment control is being used, the application has the ability to commit a set of changes, or abort (roll back) a set of changes. These changes must be guaranteed to either all happen, or be completely backed out. The mechanism that the database uses to accomplish this is a journal. The journal keeps track of each change that has been made, and this journal entry, along with a lock that is placed on the changed row, ensures that it can be fully committed or rolled back.

If you are getting messages that you cannot open your database files because they are not journaled, you have several options:

- Journal your files. Okay, this is one is pretty obvious, but if you really want to

use commitment control in your application, then journaling cannot be avoided.

Turn off commitment control. The use of commitment control is the default for CLI programs, and must be explicitly turned off if it is not needed. Use the following example:

```
attr = SQL_COMMIT_NONE;
SQLSetConnectAttr(hdbc,SQL_ATTR_COMMIT,&attr,0);
```

Set your cursors to be read-only. This is a nice alternative, because you can still use commitment control for the purpose of locking records, but by designating that you will not update any records, the CLI will be able to run against files that are not journaled. Starting in V4R4, this is the default.

The following code sets a cursor to be read-only (or in SQL terms, FOR FETCH ONLY):

```
attr = SQL_TRUE;
SQLSetStmtAttr(hstmt,SQL_ATTR_FOR_FETCH_ONLY,&attr,0);
```

**This is recommended for all statements that will have result sets, unless you plan to use update-capable cursors. If you do not plan to exploit update-capable cursors, then switching to read-only cursors has several benefits.**

Files do not have to be journaled.

Data can be blocked on retrieval, giving better performance.

The optimizer can use indexes to improve the query performance.

---

### How does AUTOCOMMIT work?

AUTOCOMMIT means that the user of the CLI does not want to do any COMMITs or ROLLBACKs. Every executed statement should be automatically committed upon successful completion.

Prior to V4R4, this was implemented by turning off commitment control. In other words, setting AUTOCOMMIT on was the same thing as setting the commit level to NONE. This means that files need not be journaled, and the user does not need to code their own commits and rollbacks. The limitations of this are that LOB locators cannot be accessed, and if the program calls an SQL stored procedure that does work under commitment control, those changes will not be committed. Prior to V4R4, those limitations may be a reason to not use AUTOCOMMIT.

In V4R4, there is support for true AUTOCOMMIT processing. This means that the AUTOCOMMIT attribute and the commitment control level are completely independent.

The following table summarizes the affects of AUTOCOMMIT and commit level settings:

| AutoCommit setting | Commit level | Results |
|---|---|---|
| Off | None | Statements do not run under commitment control and you do |

| AutoCommit setting | Commit level | Results |
|---|---|---|
| | | not need to journal database files. LOB locators cannot be used. If a statement fires a trigger or UDF that makes changes under commitment control, those changes will be rolled back. |
| On | None | Statements do not run under commitment control and you do not need to journal database files. LOB locators can be used. If a statement fires a trigger or UDF that makes changes under commit, those changes are committed automatically by the database. |
| Off | Not none | Statements run under commitment control and journaling of the database files is required. There are no unique restrictions on LOB locators, triggers, or UDFs. The application must issue its own commit statements. |
| On | Not none | Statements run under commitment control and journaling of the database files is required. There are no unique restrictions on LOB locators, triggers or UDFs. Commits are performed automatically by the database. |

### Can I do a blocked INSERT using the CLI?

Yes, beginning with V4R4, the CLI supports blocked INSERT statements. The application must indicate the number of rows to be inserted by calling the SQLParamOptions procedure. So if your application wants to insert 100 rows of data on a single call, it must first do the following:

```
long offset;
long numRows = 100;
SQLParamOptions(hstmt,numRows,&offset);
/* the third parameter is not currently used */
```

Once that is done, your program does the SQLBindParameter calls and executes the statement using the following guidelines:

The statement must be of the multiple-row form [i.e. INSERT INTO CORPDATA.NAME ? ROWS VALUES(?,?) ]. Even though there is an extra parameter marker (?) to indicate the number of rows, this value must not be bound by your program, since that value is provided by the SQLParamOptions. So in this example, you should only use SQLBindParameter for parameter numbers 1 and 2.

The data to be inserted must be organized in a contiguous, row-wise fashion. This means that all of the data for the first row is contiguous in storage, followed by all of the data for the next row, etc.

The addresses provided on SQLBindParameter will be used to reference the first row of data. All subsequent rows of data will be obtained by incrementing those addresses by the length of the entire row.

The indicator pointer passed on the SQLBindParameter must also reference contiguous storage for all of the indicator values for all of the rows. If you are inserting 100 rows, with 2 columns per row, there would be 200 4-byte indicators. The SQLBindParameter for parameter 1 would pass the address of the first of those 200 indicators, and the SQLBindParameter for parameter 2 would pass the address of the second of those 200 indicators. The system would then obtain the addresses of all subsequent indicators by incrementing

the initial addresses. That is why the storage for the indicators must be contiguous.

The indicators should be initialized with 0. Any indicator can then be individually set if you want to indicate a NULL value for a particular column and row.

---

### How do I use SQLExtendedFetch?

The function SQLExtendedFetch can provide a huge performance boost for applications that retrieve a large number of records. The basic purpose of SQLExtendedFetch is to give the application control over how many records should be retrieved on a fetch. If you use SQLFetch, then only one row will be retrieved, and the application need only provide space for one row of data. When using SQLExtendedFetch, the application must set aside storage for all the data that is going to be retrieved.

For an example, let's say that the application wants to retrieve 10 rows on one call to the CLI. This is a good idea for performance reasons, since you replace 10 calls down through the CLI and the underlying SQL runtime engine with a single call. Let's also say that the SELECT statement is going to return 3 columns, and they are 2 integers and a CHAR(10).

```
.
.
typedef struct
{
SQLINTEGER col1;
SQLINTEGER col2;
SQLCHAR    col3[10];
} colstruct;
colstruct rows[10];
SQLINTEGER fetchlen[30];
SQLSMALLINT outstat[10];
SQLINTEGER attr;
.
.
.
/* Allocate the statement handle   */
SQLAllocHandle(SQL_HANDLE_STMT,hdbc,&hstmt);
/* Set FOR FETCH ONLY attribute to allow
blocking and give better performance */
attr = SQL_TRUE;
SQLSetStmtAttr(hstmt,SQL_ATTR_FOR_FETCH_ONLY,&attr,0);
/* Prepare and Execute the Statement
(could have also used SQLExecDirect) */
SQLPrepare(hstmt,"SELECT EMP_NUM, DEPT_NUM,
DEPT_NAME FROM PERSONNEL",SQL_NTS);
SQLExecute(hstmt);
/* Bind each column to an address in the first
array entry of our structure */
SQLBindCol(hstmt,1,SQL_INTEGER,&rows[0].col1,4,
&fetchlen[0]);
SQLBindCol(hstmt,2,SQL_INTEGER,&rows[0].col2,4,
&fetchlen[1]);
SQLBindCol(hstmt,3,SQL_CHAR,rows[0].col3,10,
&fetchlen[2]);
/* Indicate that 10 rows will be fetched  */
numrows=10;
                stmt,SQL_ATTR_ROWSET_SIZE,&numrows,0);
```

```
/* Fetch 10 rows.  */
SQLExtendedFetch(hstmt,SQL_FETCH_NEXT,1,&numrows,outstat);
.
.
.
```

This puts 10 rows of data into the rows structure. Using subscript notation, the application can now process each row in a loop, with any further calls to the CLI. An important thing to remember is that there are 3 chunks of storage that the application must set aside for this. The first is the data itself, which must be contiguous storage large enough to accomodate the maximum number of rows to be fetched. The second piece is the length values, which are passed as the last parameter on the SQLBindCol procedure. This must also be contiguous storage at least this large:

```
(# rows) * (# columns) * (4 bytes) = total amount of storage.
```

Lastly, there is row status storage, which is 2 bytes per row. In our example, these three pieces of storage were declared as rows, fetchlen, and outstat.

If the application uses SQLExtendedFetch, but does not allocate enough storage for all of this data, there will be unpredictable results. This is because you are, in effect, telling the CLI to use that storage, starting at the addresses provided. If the CLI ends up going beyond the allocated storage, this will likely corrupt application storage. Also, EVERY column must be bound using SQLBindCol, prior to calling SQLExtendedFetch, or an error will occur.

---

### How do I specify that a cursor is read-only?

Setting your cursors to be read-only is accomplished like this:

```
attr = SQL_TRUE;
SQLSetStmtAttr(hstmt,SQL_ATTR_FOR_FETCH_ONLY,&attr,0);
```

where hstmt is the statement handle that you want to make read-only.

This is recommended for all statements that will have result sets, unless you plan to use update-capable cursors. If you do not plan to exploit update-capable cursors, then switching to read-only cursors has several benefits:

- Files do not have to be journaled.
- Data can be blocked on retrieval, giving better performance.
- The optimizer can use indexes to improve the query performance.

An important thing to remember is that this attribute ONLY applies to SELECT statements. Therefore, if the same statement handle is going to be used for a SELECT and then later for an UPDATE statement, this attribute is still recommended. This is because an SQL statement such as UPDATE, INSERT, or DELETE is known to update data, and there is no decision that has to be made by the optimizer on whether blocking and index usage should be allowed. On the other hand, a SELECT statement can be used for either reads or updates, and the optimizer does not know in advance what the application intends to do, so it must assume that the application MIGHT want to perform updates via the cursor. Setting the FOR FETCH ONLY attribute takes away the assumption, and allows the optimizer to make the choices that will result in the

best possible performance.

In V4R4, the default behavior is for cursors to be read-only.

---

### How do I use update-capable cursors?

First, a few definitions:

> A **cursor** is a term used to designate a result set. After executing a SELECT statement, your program has an open cursor. Executing a CREATE TABLE statement does not use cursors at all.
>
> A **read-only cursor** is a cursor that can only be used for reading (fetching) rows of data.
>
> Finally, an **update-capable cursor** is a cursor that can be used to fetch a row, then update the row on which the cursor is positioned. This type of processing is not very commonly used, which is why most applications can safely set all their cursor to be read-only cursors.

For those applications that need to use update-capable cursors, here is a code fragment:

```
.
.
SQLHSTMT hstmt1,hstmt2;
SQLCHAR  curname[SQL_MAX_CURSOR_NAME_LEN];
SQLCHAR  updateStmt[200];
SQLSMALLINT outlen;
long attr;
.
.
.
/* Allocate 2 statement handles   */
SQLAllocHandle(SQL_HANDLE_STMT,hdbc,&hstmt1);
SQLAllocHandle(SQL_HANDLE_STMT,hdbc,&hstmt2);

/* Set FOR FETCH ONLY attribute to FALSE to allow updates */
attr = SQL_FALSE;
SQLSetStmtAttr(hstmt1,SQL_ATTR_FOR_FETCH_ONLY,&attr,0);

/* Prepare and Execute the Statement
(could have also used SQLExecDirect) */
SQLPrepare(hstmt1,"SELECT EMP_NUM,
                   SALARY FROM PERSONNEL",SQL_NTS);
SQLExecute(hstmt1);
/* Bind each column to an address */
SQLBindCol(hstmt1,1,SQL_INTEGER,&col1,4,&fetchlen1);
SQLBindCol(hstmt1,2,SQL_INTEGER,&col2,4,&fetchlen2);
/* Fetch a row.  */
SQLFetch(hstmt1);
.
.
/* Obtain the cursor name from the SELECT statement,
 and append it to the UPDATE statement */

strcpy(updateStmt,
"UPDATE PERSONNEL SET SALARY = SALARY + 5000 WHERE CURRENT OF ");
SQLGetCursorName(hstmt1,curname,
SQL_MAX_CURSOR_NAME_LEN,&outlen);
strcat(updateStmt,curname);
/* Prepare and Execute the Statement
```

```
(could have also used SQLExecDirect) */
SQLPrepare(hstmt2,updateStmt,SQL_NTS);
SQLExecute(hstmt2);
.
.
```

The UPDATE statement will add 5000 to the SALARY column for the current row of the cursor. In the above example, there is only one call to SQLFetch, but a complete application would likely give the user the ability to go through several calls to SQLFetch, until the cursor was positioned on the row to be updated.

---

### Does the CLI use extended dynamic SQL?

No. Extended dynamic SQL is a type of SQL processing in which prepared statements are stored away for future use, without the need for a re-prepare. The CLI does not use this technique. Every statement that needs to be processed must either be prepared each time, or executed directly using SQLExecDirect.

However, for performance reasons, the SQL engine has a prepared statement cache, which allows some programs to behave more like extended dynamic SQL. Using the cache, the SQL engine stores away information about prepared statements, and keeps this information in system-wide storage. Then, when the same statement is executed again, even if its by a different user and a different job, the statement will run much faster. This is because the information stored away, such as internal structures, and optimizer information, can sometimes be very CPU-intensive to reconstruct every time. Storing the information for reuse saves system resources, and speeds up CLI applications that use the same SQL syntax multiple times. Another benefit of this function is that the user does not have to turn it on, or configure it in any way. It is part of normal SQL processing.

---

### How can I improve my CLI performance?

Several other questions in this FAQ deal with performance issues. We will attempt to list the more important ones here, along with others that were not mentioned elsewhere:

Avoid using unbound columns (see unbound columns)

If the same SQL statement is to be used multiple times, use SQLPrepare once, then use SQLExecute each time you want to run the statement. This is much faster than using SQLExecDirect every time.

Allocate your CLI environment at the very beginning of processing, and avoid freeing it unless the application (or job) is really ending. This is because there is some setup that is done when the CLI environment is allocated, and this must be cleaned up when the environment is freed. If the job is going to process this many times, there are resources wasted by continually setting up and freeing the environment. Furthermore, leaving it around has no drawbacks, since you can call SQLAllocEnv many times over, it just returns

the environment handle if one already exists.

Make use of SQLExtendedFetch when processing large numbers of rows (see SQLExtendedFetch)

Consider turning off null-termination. By default, the CLI will null-terminate all output strings. This can be costly, especially when fetching many rows that have character fields. It can be turned off with the following 2 lines:

```
int attr = SQL_FALSE;
SQLSetEnvAttr(henv,SQL_ATTR_OUTPUT_NTS,&attr,0);
```

Set all cursors to be read-only, unless you have an explicit use of update-capable cursors in your application (see read-only cursors)

If you need to use the catalog functions, try to limit the number of times they are used. Also, try to limit the scope of the queries by providing very narrow search criteria. Catalog functions on the IBM i and AS/400 go out to system-wide cross reference files which can get to be extremely large. Also, the queries of these files are very complex, and CPU-intensive. Therefore, since so many rows of data need to be processed, these functions are among the longest running functions in the CLI.

Turn off commitment control, unless your application relies on it. Commitment control processing adds code path for locking, journaling, inter-component system calls, etc.

---

### How can I map between character and numeric types?

Beginning in V4R5, the CLI allows you to convert between character types and the various numeric types, such as float, integer, and decimal. This conversion can occur on input parameters, and can also occur for output columns of a result set.

Mapping numeric columns in a result set to character variables in your CLI program is very easy. All you have to do is bind the columns like this:

```
char c1[10],c2[10];
long outl1,outl2;
SQLExecDirect(hstmt,"SELECT numval1,
       numval2 FROM TESTDB.NUMTABLE", SQL_NTS);
SQLBindCol(hstmt, 1,  SQL_C_CHAR,   c1,     10, &outl1);
SQLBindCol(hstmt, 2,  SQL_C_CHAR,   c2,     10, &outl2);
SQLFetch(hstmt);
```

Regardless of whether numval1 and numval2 are float, or integer, or decimal columns, the CLI will convert each value into its character representation. You should make sure that the data is formatted properly to suit your needs, since different data types can result in different output attributes, such as leading and trailing blanks, location of decimal point, and truncation of insignificant digits.

Mapping input character strings to numeric is a little bit different. On the bind, you must specify the target data type, and the CLI layer will perform the conversion. For example:

```
char outval1[5],outval2[5],outval3[5];
long outl1,outl2,outl3;
                 stmt,1,1,SQL_C_CHAR,SQL_DECIMAL,5,0,
```

```
       outval1  ,0,&outl1);
 SQLBindParameter(hstmt,2,1,SQL_C_CHAR,SQL_FLOAT,0,0,
       outval2  ,0,&outl2);
 SQLBindParameter(hstmt,3,1,SQL_C_CHAR,SQL_INTEGER,0,0,
       outval3  ,0,&outl3);
 outl1 = SQL_NTS;
 outl2 = SQL_NTS;
 outl3 = SQL_NTS;
 strcpy(outval1,"123");
 strcpy(outval2,"456");
 strcpy(outval3,"789");
 SQLExecDirect(hstmt, "INSERT INTO TESTDB.NUMTABLE2
       VALUES( ?, ?, ?)",SQL_NTS);
```

This demonstrates converting the input character strings to decimal(5,0), float, and integer. Note that only for SQL_DECIMAL was there a precision and scale provided. For integer and float the lengths are implied by the data type.

---

### Where is the CLI header file?

The CLI header file is required to be included in every program that uses the CLI. C programs bring this in using:

```
#include <sqlcli.h>
```

To actually browse the header file on the IBM i and AS/400, look in member SQLCLI in the file QSYSINC/H.

---

### Is there a CLI trace?

Yes. The CLI has a trace function built into the code, starting with V4R3. To turn on the trace, use the following command:

```
CHGCPATRC JOBNUMBER(*CURRENT) SEV(*INFO)
```

This example turns on *INFO level tracing for the current job. The *INFO level is what is needed for the CLI to start writing out trace records. For a long-running program that uses many CLI calls, this trace can fill up quickly. The CHGCPATRC also has parameters for specifying the buffer size for the trace, whether the trace should wrap when full, and whether the buffer should be resized. Also, note that trace can be turned on for jobs other than *CURRENT, by simply specifying the job number. This allows you to run the CLI trace against batch jobs, prestart jobs, etc.

After turning on the trace, and running a CLI program, use the DSPCPATRC command to see the trace records. You will see every CLI function call, along with a display of the parameters. This may be helpful in debugging problems, or just to identify how many times a certain function is called.

An alternative for those that do not have the CHGCPATRC and DSPCPATRC commands on their system is to use the CHGUSRTRC and DMPUSRTRC commands instead. However, there is one extra step. When doing the CHGUSRTRC to set the buffer options, you will notice that this command does not have a way to set the level of tracing. This is accomplished by doing:

```
CALL QSYS/QP0WUSRT parm('-l 2' '-c 0' 'xxxxxx')
```

where the 'xxxxxx' is the job number of the job you want to trace. Make sure

you specify the program name correctly, or simply cut and paste the above line. The program is spelled Q, P, then the number zero, and the first parameter is a lower case L.

All the CLI trace records start with 'SQL CLI:' in the text. Since there may be trace records from non-CLI functions, such as threads APIs, you can use that value to quickly locate CLI trace records. Another option is to query the temporary file that contains the data, and use search criteria to just display the CLI trace records.

For instance, if your DSPCPATRC output shows the following:

```
                        Display Physical File Member
File . . . . . :   QAP0ZDMP         Library  . . . . : QTEMP
Member . . . . . :   QP0Z886434       Record . . . . . : 1
Control  . . . . .                    Column . . . . . : 1
Find . . . . . .
*..+..1..+..2..+..3..+..4..+..5..+..6..+..7..+...
886434:222784 SQL CLI: SQLAllocHandle ( 1 , 0 , SQLINTEGER* )
886434:246288 Qp0zSGetEnv(QIBM_USE_DESCRIPTOR_STDIO)
886434:246976 Qp0zSGetEnv(QIBM_USE_DESCRIPTOR_STDIO):
        Could not find string
886434:513040 SQL CLI: SQLSetEnvAttr ( 1 , 10004 , 1 , 0 )
886434:513352 SQL CLI: SQLAllocHandle ( 2 , 1 , SQLINTEGER* )
```

You can filter out everything except the CLI trace records. Run the following SQL statement using either Interactive SQL or any query tool

```
> SELECT * FROM FROM QTEMP/QAP0ZDMP
  WHERE SUBSTR(QAP0ZDMP,17,8) = 'SQL CLI:'
```

## Is there a way to dump all the CLI handles in use?

Yes, there is. Then again, if there wasn't, would this question be in the FAQ? :-)

In release V4R5, a tool was added to dump all CLI handles currently allocated. This can be done for the current job, or for any other active job on the system. The output of this tool is a spooled file. Display the spooled file, and you will see all allocated environment handles, connection handles, statement handles, and descriptor handles. The tool also displays the attributes of each, such as telling you which statement handles have been allocated for each connection.

To dump the CLI handle array, CALL QSYS/QSQDMPHA. Then work with your spooled files to see the output.

To dump the CLI handle array for a job other than the one from which you are issuing the command, CALL QSYS/QSQDMPHA PARM('xxxxxx'), where 'xxxxxx' is the job number of the job that is currently running CLI.

## Common errors

### Connection errors

#### Local relational database not defined

In order to use the SQL CLI, or any SQL interface, there must be an entry in the system's relational database directory that names the local database. In almost every case, this local entry should match the name of the system.

Run the following command:

ADDRDBDIRE RDB(SYSNAME) RMTLOCNAME(*LOCAL)

Where SYSNAME is the name of your system's Default Local Location name (as specified in the DSPNETA command).

**Relational database not defined (SQLSTATE 42705 SQLCODE -950)**
Any data source that is specified on the SQLConnect must be defined in the system's relational database directory. Use the WRKRDBDIRE command to see entries already defined, or to add new entries to this directory.

---

## Package creation errors

**Obsolete packages exist**
If your connections are failing, and the message SQL0144 is appearing, this indicates that the server you are connecting to has obsolete CLI packages. These can be deleted by running the following statement on the server system that is returning the error:

```
DLTSQLPKG SQLPKG(QGPL/QSQCLI*)
```

The error can also be identified by an SQLSTATE of 58003, and native error & SQLCODE of -144. Once you have deleted the obsolete packages from that server, the next successful connection to that server will automatically create a new SQL package.

SQL packages are used by the CLI to communicate information to remote systems. An early release of the CLI only used 200 handles. Therefore the packages were built with 200 entries (called section numbers). When the CLI increased to 500 handles in non-server mode, and 500 statement handles per connection in server mode, the packages needed to be rebuilt.

**Packages do not create when connected using TCP/IP**
There are limitations in the TCP/IP support when using Distributed Relational Database (DRDA) processing. In some releases, support of 2-phase commit via TCP/IP is not supported. This means that remote connections using TCP/IP get classified as read-only connections. In order for these remote connections to be update-capable connections, there needs to be support for 2-phase commit. As a result, package creation, which is not allowed for read-only connections, will fail. A solution is to connect using SNA protocol, just once, in order to get the packages created.

There is a Java program supplied by the system to create the packages on a remote system. To use the tool, you just have to run the following command from the CL command line. Specify the system you wish to access in the 'system' parm. The 'user' and 'password' parms are the values that will be used to get the connection and create the packages needed for later use. You only need to run this tool once for any system that you are going to use CLI to connect to.

```
JAVA CLASS(com.ibm.db2.jdbc.app.DB2PackageCreator)
PARM('system' 'user' 'password')
```

**Packages do not create when connecting with no commitment control**
The package creation phase is a step that must run under commitment control. Therefore, if the very first connection to a remote system is with COMMIT

NONE, then the packages will not create properly. A simple solution is to make sure that you connect at least once to a new system with the default level of commitment control, which is COMMIT CHANGE. Once this has been done, a connection to that system will recognize that the package already exists, and will not attempt to create one.

As shown above, you can also use the Java tool to create the packages.

---

## Commitment control errors

### Using SQLEndTran and SQLTransact

A common error is that an application changes to not use commitment control, but still has the SQLEndTran (or SQLTransact) statement in their code. This will fail, because there is no "commit definition" that the system can commit or roll back. If the application sets the commitment control level to NO COMMIT, or turns on AUTO COMMIT mode, then any calls to SQLEndTran or SQLTransact can be removed.

The same error in reverse is when an application makes changes under commitment control, but does commit the work. If the job then ends with these changes still pending, the transaction will be rolled back, and the changes lost.

If the commit is performed like this:

```
SQLEndTran(SQL_HANDLE_ENV,henv,SQL_COMMIT);
```

Then all cursors have been implicitly closed by the SQL runtime engine. If this is done before your application has performed an SQLCloseCursor on each open cursor, then there will be a mismatch in the CLI state of that handle. Therefore, you should always close cursors before using option SQL_COMMIT or SQL_ROLLBACK. An alternative is to use SQL_COMMIT_HOLD or SQL_ROLLBACK_HOLD. These options do not close cursors, and allow your application to leave cursors open, if desired.

---

## Problems with closing cursors

### Cursor already open (SQLSTATE 24502 SQLCODE -502)

If a statement with a result set has been processed, then the application must close the cursor. This can be accomplished using SQLCloseCursor, or by using SQLFreeStmt with the SQL_CLOSE option. Both of these will close an open cursor, if one exists. This allows the statement handle to prepare and execute a new SELECT statement. If the previous cursor was never closed, you will either get a Function Sequence Error (error code 10) from the CLI, or a Cursor Already Open (SQLSTATE 24502) from runtime SQL.

### Cursor not open (SQLSTATE 24501 SQLCODE -501)

This can be caused when the cursor has been closed by some other system function, before the application issued the explicit close. Doing a COMMIT or ROLLBACK, without the HOLD option, can cause this. Also, the termination of an activation group can cause this. To avoid this error, always close cursors when the application has completed processing the result set.

Another situation in which this error can occur is when the SQL cleanup routine

has been executed. This occurs when your application has ended, even if you intended to continue using the cursors. If you have a main program on the stack, and this program stays active for the duration of the application, you should place your SQLAllocEnv in this main program or procedure. This prevents the SQL cleanup routine from closing your open cursors.

## Data mapping errors

**Trying to map numeric data into character variables**

The SQL engine does not map numeric data to character variables, prior to V4R5. Anytime an application binds a character variable to a numeric column, either the SQLExecute or SQLFetch will fail with a type mismatch error. When running on V4R5 or later releases, the CLI handles this conversion. For more information, see character/numeric conversion.

**Using DATE, TIME, and TIMESTAMP column types**

The SQL engine always returns DATE, TIME, and TIMESTAMP data in character form. So there is no reason to bind these columns to anything other than SQL_CHAR. A character variable of length 10 can accomodate any date value. For TIME and TIMESTAMP, use character strings of length 8 and 26, respectively.

**Working with DECIMAL and NUMERIC column types**

There are some common errors for using DECIMAL (packed decimal) and NUMERIC (zoned decimal). When you use SQLBindCol, SQLBindParam, or SQLBindParameter with these types, you need to tell the CLI what your variable is defined as. So the only time to use SQL_DECIMAL and SQL_NUMERIC on these functions is when you have actually declared the variable as either packed or zoned decimal in your program. The packed decimal is the far more common case, so we will focus on that example. Let's say you have used the proper header file and C syntax to declare a packed decimal with precision and scale of 12,2. This means that for SQLBindCol, the 3rd parm must be SQL_DECIMAL, and the 5th parameter must be (12 * 256) + 2 = 3074. The 3074 is needed because SQLBindCol does not have separate parms for precision and scale, so the values must be passed as one composite value. For SQLBindParam, the SQL_DECIMAL constant must be used for both the 3rd and 4th parameters. 12 and 2 should be passed as the 5th and 6th parameters to indicate precision and scale. For SQLBindParameter, similar rules apply. The SQL_DECIMAL constant must be passed for the 4th and 5th parameters, and the precison and scale (12 and 2) are passed as the 6th and 7th parameters.

## Running out of CLI workspace

**SQLDA not valid (SQLSTATE 51004 SQLCODE -822)**

This error may be an indication that the CLI has exhausted its internal workspace. This situation can be caused by extremely high use of unbound fetches, or by an application that runs for a very long time without ever freeing the CLI environment, or disconnecting.

To avoid this error on systems running V2R2 or earlier releases, remove any unbound fetches (see unbound columns), try to periodically reset the CLI environment using SQLFreeEnv, or simply reduce the number of handles being used. Another heavy user of internal workspace is SQLPutData, so analyze your application to see if any calls to SQLPutData can be removed. Taking one or more of the above actions may solve the problem entirely, or may just allow your program to run longer before getting the error.

In V4R3, the amount of internal workspace was increased significantly, and the CLI is far more efficient in its use of workspace. The main differences in V4R3, with respect to workspace are:

There is 48MB of workspace per connection. In V4R2, there was 16MB of workspace for the entire CLI environment.

SQLPutData() and SQLParamData() no longer use up workspace.

The workspace for a connection is completely reclaimed on an SQLDisconnect(). In V4R2, the workspace could not be reclaimed until an SQLFreeEnv().

---

### References

```
SC41-5806   DB2 for i SQL Call Level Interface
SC41-4611   DB2 for i SQL Programming
SC41-4612   DB2 for i SQL Reference
Tech Studio

Information Center
```

---