Dynamic SQL For DB2

By Craig S. Mullins

What type of query will allow DB2's optimizer to make the best use of the distribution statistics gathered by RUNSTATS in DB2 Version 2.2? When a distributed query is executed at its remote site, what type of SQL is used? When SQL is issued via QMF, SPUFI and DBEDIT, what type of SQL is used? The answer to these questions is dynamic SQL.

The continued enhancement of the capabilities of dynamic SQL, coupled with its growing acceptance and usage, will create a need for more programmers and analysts to become versed in the powers of dynamic SQL. This article presents the distinguishing characteristics of static SQL, dynamic SQL and the four flavors of dynamic SQL, and points out when to use dynamic SQL from both a performance and a capability standpoint.

Static Vs. Dynamic SQL: Differences

To compare and contrast these two types of SQL it is necessary to establish a framework. First, this article presents SQL only as it applies to DB2. When SQL (either dynamic or static) is mentioned, it is assumed to be embedded SQL. Embedded SQL is coded into an application program as opposed to interactive SQL, which is issued via SPUFI or QMF. SPUFI and QMF are both programs that contain embedded dynamic SQL. When mentioning programs into which SQL is to be embedded, it is assumed there is a COBOL II environment. VS/COBOL, FORTRAN, C, PL/I and Assembler are the other languages supported by DB2. However, FORTRAN and VS/COBOL do not provide the capability to execute certain dynamic SQL statements. These specific deficiencies will be pointed out in the appropriate places. Throughout the article, various examples are presented. In each case, these examples use the sample tables provided with DB2 and available at the user's site.

It is also assumed the basics of programming static SQL for DB2 are known and well understood. For this reason, basic DB2 coding conventions such as the SQLCA and error handling will not be covered.

The primary difference between static and dynamic SQL is described by the names static and dynamic. A static SQL statement is hard-coded and nonchanging. The columns, tables and predicates are all known beforehand and cannot be changed. Only host variables which provide values for the predicates may be changed. A dynamic SQL statement, conversely, can change throughout the course of a program's execution. The algorithms in the program can alter the SQL prior to its execution. This means that, based on the flavor of dynamic SQL being used, the columns, tables and complete predicates can be changed "on the fly" within an application program.

As might be expected, dynamic SQL varies from static SQL in the way the SQL within the program is bound into application plans. At this time, a basic refresher on DB2 program preparation procedures is in order. DB2 program preparation comprises the steps necessary to make a DB2 program executable. Refer to Example 1 for a graphic representation of DB2 program preparation. A source program containing SQL must be passed through the DB2 precompiler. This creates a Database Request Module (DBRM) containing all the static SQL in the program. A modified source program is also created with the SQL commented out and calls to DB2 substituted. The normal compile/link process is executed on the modified source. The DBRM, however, must be passed through an additional step: the DB2 BIND process. BIND will validate SQL syntax and authorization, select access paths based on statistical information in the DB2 Catalog and build an application plan from the SQL and the access path chosen.

Dynamic SQL, on the other hand, cannot be bound prior to execution. If the SQL is not known until the program executes, how can it be verified beforehand? Further, how can access paths be chosen for tables that may not be known until the program executes? For this reason, dynamic SQL is not bound, but is prepared at execution time. A PREPARE is functionally equivalent to dynamic BIND. At execution time, the program issues a PREPARE statement prior to issuing dynamic SQL (with one exception as noted in the EXECUTE IMMEDIATE section). PREPARE will verify, validate and determine access paths dynamically.

Different Types Of Dynamic SQL

There are four distinct flavors of dynamic SQL: EXECUTE IMMEDIATE, non-SELECT PREPARE/EXECUTE, Fixed-List SELECT and Varying-List SELECT. The first two do not allow SELECT statements, whereas the last two are geared specifically for SELECT statements.

EXECUTE IMMEDIATE

A program issuing an SQL statement by means of the EXECUTE IMMEDIATE flavor of dynamic SQL is limited to a subset of SQL statements. The most important are DELETE, INSERT, UPDATE and COMMIT. If these are the only types of SQL statements a program needs to execute, the code could be simple. Load the dynamic SQL statement into a host variable and issue an EXECUTE IMMEDIATE. The statement will be automatically prepared and executed.

EXAMPLE 1

DB2 Program Preparation Process

```
Source          DB2                                          DB2
Program ------> Precompiler ------> DBRM ------> BIND ------> Catalog
                    |
                    v
             Precompiled Source
                    |
                    v
           Compiler (or Assembler)
                    |
                    v
              Compiled Source
                    |
                    v
              Linkage Editor
                    |
                    v
                Load Module
```

Example 2 shows how to DELETE rows from a table. Note the simplicity of the code. STRING-VARIABLE is a host variable. It must be a character variable of varying length.

It is important to note that despite EXECUTE IMMEDIATE's ease of use, it is usually not the best choice for most application programs due to potential performance problems. This is primarily true if the same SQL statement will be executed multiple times in one invocation of an application program. After an EXECUTE IMMEDIATE is performed, the executable form of the SQL is destroyed. This means each time an EXECUTE IMMEDIATE statement is issued, it must be prepared again. This is done automatically and can involve a significant amount of overhead.

A better choice might be to PREPARE and then EXECUTE the non-SELECT dynamic SQL statement.

Non-SELECT PREPARE/EXECUTE

All dynamic SQL can be logically broken down into two steps: PREPARE and EXECUTE. This flavor of dynamic SQL accomplishes this breakdown. Example 3 depicts Example 2 modified to use PREPARE/EXECUTE instead.

A useful feature of dynamic SQL, known as a parameter marker, is used as a place holder for host variables in a dynamic SQL statement. For example, the SQL in Example 3 substitutes a question mark (?) in place of the 'A00' in the predicate. The question mark is the parameter holder. When the statement is executed, a value is moved into the host variable (in this case :TVAL) and coded as a parameter to the CURSOR by means of the USING clause.

On the surface this appears to be more difficult to code and it is, slightly. It is not, however, overly complicated and can provide huge performance benefits. Consider, for example, if a program is executing SQL statements as in Examples 2 and 3, but is basing the DELETE on input from a file. A loop reads the values from the file and issues the DELETE. With EXECUTE IMMEDIATE, a PREPARE is done for every new DELETE inside the loop. With PREPARE/EXECUTE the PREPARE can be isolated outside the loop. The value that provides the condition for deleting can be substituted via a host variable and a parameter marker. If there were thousands of DELETE statements to be executed, then thousands of PREPAREs could be avoided using this technique. Reduced overhead, reduced run time and increased efficiency would result.

However, most embedded SQL programs that issue DELETE statements also perform DB2 COMMIT processing to increase concurrency and limit reprocessing in the event of an abend. After each COM-

SystemView™ and Automated Operations

Technical Seminar

Confused by all the claims made about Automated Operations? Technology Management Associates give you more than the "sales pitch" given by software vendors. We offer an intensive, vendor-independent one day seminar on automation of the enterprise and preparing for IBM's SystemView systems management structure. Topics Include:

- ☐ IBM's SystemView
- ☐ How to Evalute an AO Product
- ☐ Quality and Automation
- ☐ Planning and Automation Project
- ☐ How to Prepare an Organization for SystemView
- ☐ Justification for Automation

January 6	*Houston, TX*	*January 10*	*Washington, DC*
January 17	*Atlanta, GA*	*January 24*	*Chicago, IL*
February 3	*Orlando, FL*	*February 10*	*San Francisco, CA*
February 14	*Dallas, TX*	*February 24*	*Boston, MA*
February 28	*Los Angeles, CA*		

Seminar Fee: $250
Call (504) 755-2267
to register or to receive more information

Technology Management Associates
P.O. Box 86777, Baton Rouge, LA 70879-6777

™ SystemView is a registered trademark of the IBM Corporation

CIRCLE #281 on Reader Service Card ▲

MIT issued, dynamic SQL statements must be prepared again.

Fixed-List SELECT

The first two flavors of dynamic SQL are useful for performing many different types of SQL statements. They, however, are not helpful for executing the most frequently used SQL statement: the SELECT statement. SELECT is the only SQL statement that will read data from a DB2 table.

There are two flavors of dynamic SQL that allow SELECT statements to be issued. The first and simplest is the Fixed-List SELECT. To use a Fixed-List SELECT, the exact columns to be returned must be known and unchanging. This is necessary to create the proper working-storage declaration for host variables in the COBOL II program. If the user does not know in advance the exact columns to be accessed, he cannot use Fixed-List SELECT, but must use Varying-List SELECT.

Fixed-List SELECT is illustrated in Example 4. A SELECT statement is formulated within the application program and moved to the STRING-VARIABLE. Following this, a cursor is declared and the SELECT statement is prepared. The cursor is then opened after which a loop to FETCH rows is invoked. When complete, the cursor is closed.

The previous example simplifies things considerably. The true benefit of dynamic SQL is the ability to modify the SQL. For example, the SELECT statement in Example 4 could be modified by the application program in many ways. One way is changing the columns referenced in the WHERE clause. It is important, however, that the host variables passed as parameters in the OPEN statement be of the same data type and length as the columns in the WHERE clause. If the data type and length of the columns in the WHERE clause change, the OPEN statement must be recoded with new USING parameters.

If parameter markers are not used in the SELECT statements, they could be eliminated and specific values could be substituted in the SQL to be executed. No parameters would then be passed in the OPEN statement.

The OPEN statement could also be recoded to pass parameters using an SQL Descriptor Area (SQLDA). Using the SQLDA in this manner is beyond the scope of this article.

Quite a bit of flexibility is offered by Fixed-List SELECT dynamic SQL. If, however, the additional flexibility of changing the columns to be accessed while executing is needed, DB2 provides that functionality with Varying-List SELECT.

Varying-List SELECT

Varying-List SELECT provides the most flexibility for dynamic SELECT statements. Changes can be made "on the fly" to tables, columns and predicates. Because everything about the query can change during one invocation of the program, the number and type of host variables needed to store the retrieved rows cannot be known beforehand. This will add considerable complexity to application programs. In fact, FORTRAN and VS/ COBOL programs *cannot* perform Varying-List SELECT dynamic SQL statements (unless an Assembler routine is called to handle the address variable pointers).

E X A M P L E 2

EXECUTE IMMEDIATE Example

```
SQL to execute: DELETE FROM DSN8220.PROJ
                WHERE DEPTNO = 'A00';
Move the "SQL to execute" to STRING-VARIABLE
EXEC SQL
     EXECUTE IMMEDIATE :STRING-VARIABLE;
```

E X A M P L E 3

Non-SELECT PREPARE/EXECUTE Example

```
SQL to execute: DELETE FROM DSN8220.PROJ
                WHERE DEPTNO = ?
Move the "SQL to execute" to STRING-VARIABLE
EXEC SQL PREPARE CSR1 FROM :STRING-VARIABLE;
Move 'A00' to TVAL
EXEC SQL EXECUTE CSR1 USING :TVAL;
```

Announcing . . .

DOS/VSE Testing and Debugging Tools Now Available for VSE/ESA

MIS *Managers... Are you meeting development project deadlines? Do you have application backlogs? Is your production environment stable?*

If these issues concern you, Compuware can help with a complete line of Testing and Debugging Tools. For users who work in CICS or Batch COBOL environments, Compuware has the solution.

[illegible]

Abend-AID®(Batch) — intercepts, analyzes abends then recommends corrective action. Allows "Snap" at run time to display working storage.

CICS Abend-AID®— intercepts, analyzes transaction abends then provides a concise diagnostic report from an on line directory.

XPEDITER®/CICS (formerly CICS dBUG-AID) — for interactive CICS debugging, program logic correction, and storage violation protection. Now supports COBOL II. File utility supports VSAM and DL/I.

CICS PLAYBACK™ and **CICS SIMULCAST™**— captures, customizes "scripts" for network testing. Supports training and help desk.

For more information, or a no-cost 30-day evaluation, call 1-800-535-8707 and ask for your VSE representative.

COMPUWARE®
Sophisticated Software Used Every Day

CIRCLE #67 on Reader Service Card ▲

EXAMPLE 4

Fixed-List SELECT Example

```
SQL to execute: SELECT PROJNO, PROJNAME, RESPEMP
                FROM DSN8220.PROJ
                WHERE  PROJNO    = ?
                AND    PRSTDATE  = ?
Move the "SQL to execute" to STRING-VARIABLE
EXEC SQL DECLARE CSR2 CURSOR FOR FLSQL;
EXEC SQL PREPARE FLSQL FROM :STRING-VARIABLE;
EXEC SQL OPEN CSR2 USING :TVAL1, :TVAL2;
Loop until no more rows to FETCH
   EXEC SQL
        FETCH CSR2 INTO :PROJNO, :PROJNAME, :RESPEMP;
EXEC SQL CLOSE CSR2;
```

EXAMPLE 5

SQLDA Data Element Definitions

Name	Usage in DESCRIBE or PREPARE statement
SQLDAID	descriptive only
SQLDABC	length of SQLDA
SQLN	total number of occurrences of SQLVAR available
SQLD	total number of occurrences of SQLVAR actually used
SQLTYPE	indicates data type and whether or not NULLs are allowed for the column
SQLLEN	external length of the column value
SQLDATA	address of a host-variable for a specific column
SQLIND	address of NULL indicator variable for the above host-variable
SQLNAME	contains the name or label of the column

EXAMPLE 6

Varying-List SELECT Example

```
SQL to execute: SELECT PROJNO, PROJNAME, RESPEMP
                FROM DSN8220.PROJ
                WHERE  PROJNO    = 'A00'
                AND    PRSTDATE  = '1988-10-10';
Move the "SQL to execute" to STRING-VARIABLE
EXEC SQL DECLARE CSR3 CURSOR FOR VLSQL;
EXEC SQL
   PREPARE VLSQL INTO SQLDA FROM :STRING-VARIABLE;
EXEC SQL OPEN CSR3;
Load storage addresses into the SQLDA
Loop until no more rows to FETCH
     EXEC SQL FETCH CSR3 USING DESCRIPTOR SQLDA;
EXEC SQL CLOSE CSR3;
```

The vehicle for communicating information about dynamic SQL between DB2 and the applications program is called the SQLDA. It will contain information about the type of SQL statement to be executed, the data type of each column accessed and the address of each host variable needed to retrieve the columns. The SQLDA must be hard-coded into the COBOL II program. Refer to Example 5 for a definition of each item in the SQLDA when it is used with Varying-List SELECT.

The steps needed to code Varying-List SELECT dynamic SQL into an application program will vary according to the amount of information known about the SQL beforehand. Example 6 details the steps necessary when it is known that the statement to be executed is indeed a SELECT statement. The code differs from Fixed-List SELECT in three ways: the PREPARE step and the FETCH statement use the SQLDA and a new step is added to store host variable addresses into the SQLDA.

When the PREPARE is executed, DB2 returns information about the columns that are being returned by the SELECT statement. This information is in the SQLVAR group item of the SQLDA. Of particular interest is the SQLTYPE field. For each column to be returned, this field will tell the data type and whether or not NULLs are permitted.

The applicable values for SQLTYPE can be coded as 88-level COBOL structures to aid in the detection of the data type for each column. Consult the *DB2 Application Programming Guide (SC26-4377)* for a list of valid values. The application program issuing the dynamic SQL must interrogate the SQLDA analyzing each occurrence of SQLVAR. This information is used to determine and derive the address of some storage area of the proper size to accommodate each column returned. This address is then stored in the SQLDATA field of the SQLDA. If the column can be NULL, the address of the NULL indicator is stored in the SQLIND field of the SQLDA. When this analysis is complete, rows can be fetched from it using Varying-List SELECT and the SQLDA information.

Note that the group item, SQLVAR, occurs 300 times, the limit for the number of columns that can be returned by any one SQL SELECT. This number can be modified by changing the SQLN field to a smaller number, but not a larger one. Coding a smaller number will reduce the amount of storage required, but if a greater number of columns is returned by the dynamic SELECT, the SQLVAR fields will not be populated.

It is also possible to code dynamic SQL without knowing anything at all about the statement to be executed. This is the case if there is a program that needs to read SQL statements from a terminal and execute them regardless of statement type. This is done by coding two SQLDAs: one minimal SQLDA to PREPARE the statement and determine if it is a SELECT or not. If it is not, simply EXECUTE the non-SELECT statement. If it is a SELECT, PREPARE it a second time with a full SQLDA and follow the steps in Example 6.

General Dynamic SQL Issues

Dynamic SQL is a complex topic and can be difficult to comprehend and master. It is important to keep the following general rules in mind when deciding whether or not to use dynamic SQL:

- Dynamic SQL can change during the course of a program
- Dynamic SQL is *always* prepared during the execution of the program; preparation (BIND) is accomplished prior to execution for static SQL.

An organization's specific procedures for dynamic SQL should always be followed.

Also, not every SQL statement can be executed as dynamic SQL. Most are SQL statements that provide for the execution of dynamic SQL or row-at-a-time processing.

Performance Implications Of Dynamic SQL

The performance of dynamic SQL is one of the most widely debated DB2 issues. Some shops avoid it altogether and most of those who allow it place strict controls on its usage. This is wise as of DB2 2.2. However, as new and faster versions of DB2 are released, some restrictions on dynamic SQL usage should be eliminated.

It may be best to avoid most dynamic SQL. Dynamic SQL statements are usually just a series of static SQL statements in disguise. Admittedly, the static SQL *might* take more time to code, but it will usually take less time to execute. If there is a compelling reason to use dynamic SQL, ensure the argument is sound and complies with the guidelines that follow.

Reasons To Use Dynamic SQL

Most users are aware that in certain circumstances there are valid reasons for prohibiting dynamic SQL. Sometimes, however, there are valid performance reasons for requiring dynamic SQL.

DB2 2.2 will populate the DB2 Catalog (SYSIBM.SYSFIELDS table) with distribution statistics. For columns that participate as the first column in an index, the 10 most frequently appearing values in that column will be stored along with the number of occurrences for each. For DB2 2.2 this information will only be used by the optimizer for dynamic SQL and static SQL with hard-coded, literal predicates. Static SQL predicates using host variables will still assume even distribution. Therefore, if indexed data is greatly skewed from even distribution, using dynamic SQL may prove advantageous.

As of DB2 2.2, when the LIKE predicate is used in static SQL with a host variable, an index will *never* be used. This is because the optimizer cannot determine what value will be placed into the host variable. If it begins with a wild card (that is, _or %) an index cannot be used because the first character could be any character. If the LIKE clause is built into the dynamic SQL, the DB2 optimizer can determine whether that first character is a wild card or not. If it is not, an index might be used (if all other conditions for index usage are met). DB2 2.3 will enable LIKE predicates using host variables to be indexable when the host variable does not begin with a wild card character.

Dynamic SQL may also be in order for access to active tables that fluctuate between many and few rows between RUNSTATS executions. Increasing the frequency of RUNSTATS before using dynamic SQL is recommended, but this may not always be possible for every case.

Another reason to use dynamic SQL is to allow programs to take advantage of the capabilities of QMF using the QMF Command Interface (QMFCI). Dynamic SQL is used whenever QMF is used to access DB2 data. Some of the functionality provided by the QMFCI includes scrolling and formatting data. In certain circumstances, the addition of these capabilities may offset the potential performance degradation caused by dynamic SQL.

The continued enhancement of dynamic SQL, coupled with its growing acceptance and usage, will create a need for more programmers and analysts to become versed in dynamic SQL.

Reasons Not To Use Dynamic SQL

Usually dynamic SQL will be less efficient than static SQL because of the PREPARE required during program execution with dynamic SQL. Static SQL is prepared (bound) before execution.

Also, on-line transaction-based systems require well-designed SQL to execute with sub-second response time. If dynamic SQL is allowed, the chances that the system will have well-designed SQL diminishes. If a program can change the SQL "on the fly," then the control required for on-line systems is relinquished and performance may suffer.

Fixed-List SELECT dynamic SQL usually reduces the chance for index-only scan access being chosen. This will be true if a program contains one SELECT statement per table regardless of which columns are chosen. This type of SELECT retrieves all columns in the table and returns to the user only those needed. All the columns are always returned causing DB2 to incur a read to the tablespace as well as to any index identified for the access path. An index-only scan can never be chosen under these circumstances. This is an unwise design choice and should be avoided.

Most programmers will not take the time to design a dynamic SQL application properly if it requires SELECTs. Often, Varying-List SELECT is needed for proper performance and Fixed-List SELECT is used instead to avoid using the SQLDA and pointer variables.

Dynamic SQL is more difficult to tune because it changes with each execution of a program. It cannot be traced using the DB2 Catalog tables (SYSDBRM, SYSPLANREF, SYSPLAN) because it is not bound into any application plan, further complicating the issue of tuning.

Proper administration of the Resource Limit Facility (RLF) is needed to control DB2 resources when dynamic SQL is executed. Thresholds for CPU usage are coded into the RLF on an application basis. When the RLF threshold is reached, the applications program will not abend. An SQL error code will be issued when any statement exceeds the predetermined CPU usage. This environment requires additional support from a DBA standpoint for RLF administration and maintenance, as well as additional work from an application standpoint for enhancing error handling procedures.

Finally, IBM's recommendation is to use dynamic SQL only if its flexibility is required.

Conclusion

Dynamic SQL is an important component of DB2. It is used in many commercial products today and, as such, a sound understanding of its capabilities is required.

ABOUT THE AUTHOR

Craig S. Mullins, with more than six years of database management systems experience, is a senior database analyst for Duquesne Light Company, a large, eastern electric utility. He is also vice president and cofounder of a consulting and software development firm, ASSET, Inc., P.O.Box 2547, Pittsburgh, PA 15230.