DB2 users who are avoiding dynamic SQL should reconsider its advantages

BY CRAIG S. MULLINS

Dismissing Dynamic Dangers

WHEN PERFORMANCE is an issue, don't use dynamic SQL. Such statements can be found in DB2 texts, articles, and presentations everywhere. But unlike generic "rules of thumb," the real-life decision whether to use static or dynamic SQL isn't so simple. With each new release of DB2, prohibiting dynamic SQL becomes even harder to justify as the costs of using it decline. It's time we take a closer look at some of dynamic SQL's aspects before deciding to maintain the common—but dubious—blanket rejection of its potential.

What makes dynamic SQL different from static SQL? Static SQL is optimized prior to program execution. Each static SQL statement in a program is analyzed and optimized during the DB2 BIND process. During this process, the DB2 optimizer finds the best access path, coding it into a package or plan. When the program executes, the package or plan is retrieved and the stored access path is executed.

However, dynamic SQL is optimized at run time. Prior to its execution, a dynamic SQL statement must be processed by the DB2 Optimizer so that an optimal access path can be created. (This is the PREPARE process.) You can think of PREPARE as a dynamic BIND. Many analysts believe this process is too costly for inclusion in production application programs. Hopefully, I can help you reconsider this issue.

DYNAMIC SQL OVERVIEW

Four types of dynamic SQL exist:

- ☐ EXECUTE IMMEDIATE
- ☐ Non-SELECT
- ☐ Fixed-list SELECT
- ☐ Varying-list SELECT.

Let us briefly review these types. EXECUTE IMMEDIATE will (implicitly) prepare and execute complete dynamic SQL statements coded in host variables. Its drawbacks are that it can't be used to retrieve data using the SELECT statement; and because the PREPARE is implicit within the EXECUTE IMMEDIATE, optimization must occur whenever the statement is executed.

Non-SELECT can be used to prepare and execute SQL statements in an application program. PREPARE and EXECUTE are separated, so once a statement is prepared, it can be executed over again without reoptimization. However, as its name implies, non-SELECT dynamic SQL can't issue the SELECT statement.

The third type of dynamic SQL is fixed-list SELECT. It can be used to PREPARE and execute SQL SELECT statements where we know in advance the exact columns to be retrieved by the application program. We must know what these columns are at the time the program is being coded, and they can't change during execution. This situation is necessary to create the proper working-storage declaration for host variables in your program.

If you don't know in advance the columns to be accessed, you can use the fourth dynamic SQL type, varying-list SELECT. In this case, pointer variables maintain the selected columns list. Though varying-list SELECT is the most complicated, this dynamic SQL type 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 might change during one program invocation, the number and type of host variables needed to store the retrieved rows can't be known beforehand, adding complexity to application programs.

REASONS TO RECONSIDER

The cost of the dynamic PREPARE must be added to the overhead of all dynamic SQL programs. However, this cost may not be quite as expensive as many believe. Running some queries using SPUFI with the DB2 Performance Trace switched on produced results for the cost of a PREPARE. The SQL statements prepared for the four tests shown in Figure 1 are described as follows:

1. A simple SELECT of one column from one table containing two predicates.
2. A join of two tables selecting two columns and using two predicates.
3. A join of two tables selecting four columns and using three predicates.
4. A three-table join selecting all columns.

Of course, the results will vary based upon environment, the type of dynamic SQL, and the complexity of the statement being prepared. Before proceeding with a dynamic SQL project, perform some similar tests at your shop to determine the potential impact. To obtain this type of information, you must start Performance Trace Class

Measurement	Test #1	Test #2	Test #3	Test #4
Elapsed time	0.2436	0.5633	0.8477	0.9326
TCB time	0.04520	0.09391	0.13073	0.19333

FIGURE 1. *The cost of a* PREPARE. *(All measurements are in seconds.)*

Query no.	QB no.	Step no.	Meth.	Table name	TB no.	AT	M col.	Sort new comp. UJOG UJOG	TS lock	Preftech
1	1	1	0	Employee	1	I	1	NNNN NNNN	IS	L
2	1	1	0	Employee	1	I	1	NNNN NNNN	IS	

FIGURE 2. *Explanation of results for queries 1 and 2.*

(3) and also run DB2 Presentation Manager SQL Trace Reports (or its equivalent with another performance monitoring tool).

Another misconception surrounding dynamic SQL is that a PREPARE must occur every time SQL is executed dynamically, which isn't true. If proper dynamic SQL types are chosen and the program is correctly coded, you only need one PREPARE for each SQL statement. Of course, EXECUTE IMMEDIATE dynamic SQL always prepares the statement prior to execution. For other types of dynamic SQL, the PREPARE can be isolated outside the loop performing the SQL statement. So, overhead is reduced by lowering the number of times dynamic binding occurs. However, if you want the access path to change, you must issue a PREPARE. But, for many applications, a single PREPARE is sufficient for each dynamic SQL statement within the program.

Another reason to consider dynamic SQL is performance improvement. Dynamic SQL queries accessing data not distributed uniformly could outperform an equivalent static SQL statement. Why? Because dynamic SQL can employ nonuniform distribution statistics (NUDS) when it isdetermining access paths. NUDS are stored in the DB2 Catalog in .SYSFIELDS (DB2 v. 2.3), or in .SYSCOLDIST and .SYSCOLDISTSTATS (DB2 v. 3).

Consider the following query:

```
SELECT FIRST_NAME, ADDRESS, TITLE
FROM EMPLOYEE
WHERE LAST_NAME = :HOST VARIABLE
```

In a static SQL environment, different values will be placed into the host variable. However, the access path never changes since it was determined prior to execution at BIND time. In a dynamic SQL environment, however, the access path can change based upon the value specified for the predicate. Consider the following two queries:

Query 1
```
SELECT FIRST_NAME, ADDRESS, TITLE
FROM EMPLOYEE
WHERE LAST_NAME = 'SMITH'
```

Query 2
```
SELECT FIRST_NAME, ADDRESS, TITLE
FROM EMPLOYEE
WHERE LAST_NAME = 'JAWORSKI'
```

If we assume the data in the EMPLOYEE table is skewed so more employees are named Smith than Jaworski, then a different access path may be chosen for each query when using dynamic SQL. The actual access paths for each query using dynamic SQL are shown in Figure 2. Note that the query accessing the name occurring more frequently chose an access path specifying LIST PREFTECH, while the other didn't. In this case, LIST PREFTECH can reduce the amount of I/O by sorting the RIDs before accessing the data pages, thereby enhancing performance.

MATHEMATICAL REASONS

Even if decreasing costs don't compel you to use dynamic SQL, one situation exists where dynamic SQL should almost always be chosen over static SQL: When a user can choose numerous combinations of predicates at run time.

Consider the following: What if, for a certain query, 20 predicates are possible. The program's user is permitted to choose up to six predicates for any given request. How many different static SQL statements must be coded to satisfy these specifications?

First, let us determine the number of different ways you can choose six predicates out of 20. To do so, we must use combinatorial coefficients. So, if n is the number of different ways, then:

$$n = (20 \times 19 \times 18 \times 17 \times 16 \times 15) / (6 \times 5 \times 4 \times 3 \times 2 \times 1)$$

$$n = (27{,}907{,}200) / (720)$$

$$n = 38{,}760$$

This answer shows the total number of different ways we can choose six predicates out of 20 if the predicate ordering doesn't matter. (For all intents and purposes, it doesn't. For performance, you may want to put the predicate with the highest cardinality within each type of operator first, but we won't concern ourselves with this approach here.) However, since the specifications clearly state a user can choose up to six, we must add in the different ways to choose:

- Six predicates out of 20: $(20 \times 19 \times 18 \times 17 \times 16 \times 15)/(6 \times 5 \times 4 \times 3 \times 2 \times 1) = 38{,}760$.
- Five predicates out of 20: $(20 \times 19 \times 18 \times 17 \times 16)/(5 \times 4 \times 3 \times 2 \times 1) = 15{,}504$.
- Four predicates out of 20: $(20 \times 19 \times 18 \times 17)/(4 \times 3 \times 2 \times 1) = 4{,}845$.
- Three predicates out of 20: $(20 \times 19 \times 18)/(3 \times 2 \times 1) = 1{,}140$.
- Two predicates out of 20:

$(20 \times 19)/(2 \times 1) = 190$.

☐ One predicate out of 20: $20/1 = 20$

This tally brings the grand total of static SQL statements that must be coded to 60,459. In this situation, if static SQL is forced upon us, we have one of two options:

1. Code for 40 days and 40 nights, hoping to write 60,459 SQL statements.

2. Compromise on the design and limit the users' flexibility.

I guarantee that 99.99 percent of the time the second option will be chosen. My solution is abandon static SQL, using dynamic SQL in this situation. How would this decision ease the development situation? Consider these advantages:

☐ With dynamic SQL, the 20 predicates must be coded only once; in working storage.

☐ As the program runs, the application logic can build the complete SQL statement based upon user input.

☐ The Database Request Module's (DBRM's) size decreases dramatically. The DBRM for the static SQL program would be huge if it contained all 60,459 SQL statements. Even if a compromise number is reached, chances are the DBRM will be large. I guarantee it will be larger than the DBRM for the dynamic SQL program, regardless of the compromise number.

☐ Systemwide performance may improve due to large packages and plans using the EDM pool less often.

☐ Although additional runtime overhead is required to perform dynamic PREPARE, performance usually won't suffer. Remember, for SQL issued against nonuniform data, performance improved.

In performance terms, degradation must be phenomenal to offset the savings accrued by minimized applications-development time.

To sum up, when should you consider using dynamic SQL?

☐ When the program's nature is truly changeable, as in the earlier example.

☐ When the columns to be retrieved can vary from execution to execution. This case is similar to the earlier example, where a user might choose multiple predicate combinations.

☐ When benefit can be accrued from interacting with another dynamic SQL application. For example, those applications using the QMF callable interface.

☐ When SQL must access nonuniform data. You can use the NUDS stored in the DB2 Catalog to generate different access paths based on different predicate data values.

BAD BECOMES GOOD

Dynamic SQL isn't always bad. As your DB2 applications-development needs mature, consider dynamic SQL when it makes sense. Don't pledge allegiance to a "no dynamic SQL at our shop" rule without realizing its huge potential for the applications-development life-cycle and providing flexibility in DB2 program design.

I don't mean to imply dynamic SQL should be used if unmerited. Hang on to your common sense, but remember, rules with "never" in them are "usually" unwise! ■

Craig S. Mullins is a researcher and developer in the education department of Platinum Technology Inc.