Dynamic SQL Performance for Db2 for z/OS

By Craig S. Mullins

Over the years, the performance of dynamic SQL in DB2 applications has been one of the most widely debated DB2 issues. In the early days of DB2, dynamic SQL was avoided like the plague. Even today, some shops still try to avoid it. But in this day and age of ERP systems and web applications, both of which typically rely on dynamic SQL, its use is becoming more mainstream.

Still, many shops that allow dynamic SQL try to place some controls on its use. But as new and faster versions of DB2 are released, with improved functionality for using and managing dynamic SQL programs, many of the traditional restrictions on dynamic SQL can be eliminated.

I suppose that you can still find some valid reasons for prohibiting dynamic SQL. For example, you should avoid dynamic SQL when the dynamic SQL statements are just a series of static SQL statements in disguise. Consider an application that needs two or three predicates for one SELECT statement that is otherwise unchanged. Coding three static SELECT statements can be more efficient than coding one dynamic SELECT with a changeable predicate. The static SQL takes more time to code but probably less time to execute.

Another reason for avoiding dynamic SQL is that it can require more overhead to process than equivalent static SQL. Dynamic SQL incurs overhead because the cost of the dynamic bind, or PREPARE, must be added to the processing time of all dynamic SQL programs. But this overhead is not quite as costly as many people seem to think it is.

To determine the cost of a dynamic bind, consider running some queries using SPUFI with the DB2 Performance trace turned on. Then examine the performance reports or performance monitor output to determine the elapsed and TCB time required to perform the PREPARE. The results should show elapsed times less than 1 second and subsecond TCB times. The actual time required to perform the dynamic prepare will vary with the complexity of the SQL statement. In general, the more complex the statement, the longer DB2 will take to optimize it. So be sure to test SQL statements of varying complexity.

Of course, the times you get will vary based on your environment, the type of dynamic SQL you use, and the complexity of the statement being prepared. Complex SQL statements with many joins, table expressions, unions, and subqueries take longer to PREPARE than simple queries. However, factors such as the number of columns returned or the size of the table being accessed have little or no effect on the performance of the dynamic bind.

And prepared dynamic SQL can be cached in the dynamic statement cache so that the same SQL statement can reuse the mini plan for the statement the next time it runs. Of course, the statement has to be exactly the same in order to benefit from the dynamic SQL cache.

Keep in mind, too, that performance is not the only factor when deciding whether or not to use dynamic SQL. For example, if a dynamic SQL statement runs a little longer than a static SQL statement but saves days of programming cost then perhaps dynamic SQL is the better choice. It all depends on what is more important -- the cost of development and maintenance or squeezing out every last bit of performance at any cost.

Overhead issues notwithstanding, there are valid performance reasons for favoring dynamic SQL, too. For example, dynamic SQL can enable better use of indexes, choosing different indexes for different SQL formulations. Properly coded, dynamic SQL can use the column distribution statistics stored in the DB2 catalog, whereas static SQL is limited in how it can use these statistics. Use of the distribution statistics can cause DB2 to choose different access paths for the same query when different values are supplied to its predicates.

The REOPT bind parameter can be used to allow static SQL containing host variables or special registers to behave like incremental-bind statements. When dynamic reoptimization is activated, a dynamic bind similar to what is performed for dynamic SQL is performed. This means that these statements get compiled at the time of EXECUTE or OPEN instead of at BIND time. During this compilation, the access plan is chosen, based on the real values of these variables.

There are four options from which to choose when specifying REOPT:

REOPT(NONE) -- DB2 will not reoptimize SQL at run time to factor in the values of host variables, parameter markers, and special registers. REOPT(NONE) is the default; choose this option when static SQL access paths are fine. NOREOPT(VARS) can be specified as a synonym of REOPT(NONE).

REOPT(ALWAYS) -- DB2 will reprepare every time the statement is executed. This means that statements containing host variables, parameter markers, or special registers will be prepared using actual values, which should improve the optimization. Subsequent OPEN or EXECUTE requests for the same statement will reprepare the statement, reoptimize the query plan using the current set of values for the variables, and execute the newly generated query plan. Statements in plans or packages that are bound with REOPT(ALWAYS) cannot be saved in the cache. Additionally, KEEPDYNAMIC(YES) is not compatible with REOPT(ALWAYS). Consider using REOPT(ALWAYS) for

dynamic SQL with parameter markers and to avoid dynamic statement caching. REOPT(VARS) can be specified as a synonym for REOPT(ALWAYS).

REOPT(ONCE) -- DB2 will prepare SQL statements only once, using the first set of host variable values, no matter how many times the statement is executed by the program. The access path is stored in the dynamic statement cache and will be used for all subsequent executions of the same SQL statement. This option was introduced in DB2 V8.

REOPT(AUTO) -- This option directs DB2 to attempt to formulate the optimal access path in the minimum number of prepares. The basic premise of REOPT(AUTO) is to re-optimize only when host variable values change significantly enough to make reoptimization worthwhile. Using this option, DB2 will examine the host variable values and will generate new access paths only when host variable values change and DB2 has not already generated an access path for those values. This option was introduced in DB2 9.

Additionally consider dynamic statement caching, the intent of which is to reduce the overhead of dynamic SQL by avoiding a full prepare, along with the cost of optimizing the SQL, whenever possible. There are three types of dynamic statement caching:

1. Global dynamic statement cache
2. Local dynamic statement cache
3. Both local and global combined

The **global dynamic statement cache** is allocated in the dynamic statement cache pool (EDMSTMTC), which is allocated above the 2 GB bar. Global dynamic statement caching causes the skeleton copy of a prepared SQL statement to be held in the cache. Only one skeleton copy of the same statement is held. The skeleton copy can be used by user threads to create user copies. An LRU algorithm is used for replacement. If an application issues a PREPARE or an EXECUTE IMMEDIATE (and the statement has not been executed before in the same commit scope), and the skeleton copy of the statement is found in the global statement cache, it can be copied from the global cache into the thread's storage instead of requiring a full prepare.

A **local dynamic statement cache** is allocated in the storage of each thread in the DBM1 address space. The KEEPDYNAMIC bind parameter is used to control usage of the local dynamic statement cache.

And, of course, you can deploy **both** local and the global dynamic statement caching. A prepare can only be avoided when using both caches. As the full prepared statement is kept across commits, when issuing a new EXECUTE statement (without a PREPARE after a COMMIT) nothing needs to be prepared. The full executable statement is still in the

thread's local storage (assuming it was not removed from the local thread storage because MAXKEEPD was exceeded) and can be executed as such.

There are three types of prepares to consider:

1. A *full prepare* occurs when the skeleton copy of the prepared SQL statement does not exist in the global dynamic SQL cache (or the global cache is not active). A full prepare can be caused to occur explicitly by a PREPARE or an EXECUTE IMMEDIATE statement or implicitly by an EXECUTE when using KEEPDYNAMIC(YES).
2. A *short prepare* occurs, if the skeleton copy of the prepared SQL statement in the global dynamic SQL cache can be used.
3. An *implicit prepare* can occur when an application using KEEPDYNAMIC(YES) issues a new EXECUTE after a COMMIT, but a prepare cannot be avoided (if this statement is not in the cache or was removed). In such as case, DB2 will issue the prepare implicitly on behalf of the application.

The KEEPDYNAMIC parameter is used to control whether dynamic SQL is kept across a COMMIT point.

KEEPDYNAMIC(NO) -- DB2 will not keep dynamic SQL statements after COMMIT points. Dynamic SQL must be prepared after each COMMIT. This is the simple, standard technique for coding dynamic SQL.

KEEPDYNAMIC(YES) -- DB2 will keep dynamic SQL statements after COMMIT points. You will not need to code your program to prepare dynamic SQL statements after each COMMIT. You will need to re-prepare after a ROLLBACK, though.

With KEEPDYNAMIC(YES) DB2 will keep the dynamic SQL statement until the application ends, a ROLLBACK is issued, or the application explicitly issues another PREPARE.

How does the KEEPDYNAMIC parameter interact with the dynamic statement cache? If the prepared statement cache is active, KEEPDYNAMIC(YES) will cause DB2 to keep a copy of the prepared statement in the cache. If the prepared statement cache is not active, DB2 keeps only the SQL statement string past a COMMIT point. DB2 will implicitly prepare the SQL statement the next time the application executes an OPEN, EXECUTE, or DESCRIBE operation for that statement.

Generally speaking, specifying KEEPDYNAMIC(YES) will improve performance and simplify program design.

Summary

Dynamic SQL usually provides the most efficient development techniques for applications with changeable requirements (for example, numerous screen-driven

queries). In addition, dynamic SQL generally reduces the number of SQL statements coded in your application program, thereby reducing the size of the plan and increasing the efficient use of system memory.

So, if you have a compelling reason to use dynamic SQL, then by all means, go ahead and code up your program to use dynamic SQL. I mean, after all, it is no longer the early, dark days of DB2 when dynamic SQL almost certainly meant performance problems. And, as I mentioned in the beginning, dynamic SQL is likely to be foisted on you in your new, more modern applications even if you continue to desperately keep it out of your COBOL programs.