
Fall 2009

## *THE BUFFER POOL*

### Making Static SQL More Dynamic… and Vice Versa

By Craig S. Mullins

SQL can be either static or dynamic. Static SQL is rigidly formed and *usually* provides a known access path to data. On the other hand, dynamic SQL, as you might expect, is more flexible. This flexibility brings with it important trade offs, the most important of which is the potential for more accurate access path formulation at the cost of not knowing what that access path will be before the dynamic SQL runs.

Of course, the differences between static and dynamic SQL are lessening over time. Over the years, IBM has added various options and features that blur the differences between the two. Today, there are fewer hard and fast lines between what can be accomplished with dynamic SQL.

### Making Dynamic SQL More Static

As the DB2 ability to optimize SQL has improved, the cost of preparing a dynamic SQL statement has grown. For some SQL requests, the processing cost of preparing dynamic SQL statements can be significant. Compounding the potential performance problem, applications might be forced to pay the cost of preparation more than once for each dynamic SQL statement. Indeed, the cost of preparation historically has been one of the greatest impediments to the acceptance of dynamic SQL.

It stands to reason, therefore, that one of the biggest boons to the growing number of dynamic SQL applications was the introduction of the dynamic statement cache (DSC) in DB2 V5. With DSC, dynamic SQL, which is called repeatedly, can be prepared and cached for use by subsequent application processes. You must specify BIND options to control when to reevaluate access paths after the values of host variables or parameter markers have been determined.

After an SQL statement has been prepared and is automatically saved in the cache, subsequent prepare requests for that same SQL statement can avoid the costly preparation process by using the statement that is in the cache. Statements that are saved in the cache can be shared among different threads, plans, or packages. When the prepared statement is found in the dynamic statement cache, it is possible to accrue significant savings.

Data manipulation statements can be saved in the cache. Distributed and local SQL statements are eligible, as are prepared, dynamic statements that use DB2 private protocol. Prepared statements cannot be shared among data sharing members. Because each member has its own EDM pool, a cached statement on one member is not available to an application that runs on another member.

The dynamic SQL statement must be exactly the same as the one which caused the statement to be cached in order for it to be reused. This means everything must be the same, including any parameter marker values. Even an extra space in the new SQL statement will cause the cached statement not to be used.

*Types of Dynamic Statement Caching*

The intent of dynamic statement caching 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:

- Global dynamic statement cache
- Local dynamic statement cache
- 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:

- 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).
- A *short prepare* occurs, if the skeleton copy of the prepared SQL statement in the global dynamic SQL cache can be used.
- 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.

*Effects of KEEPDYNAMIC on dynamic SQL*

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 DSC? 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. However, keep in mind that the local dynamic statement cache takes up DBM1 virtual storage below 2GB. This means there is a trade-off between storage for the cache versus avoiding a PREPARE after issuing a COMMIT. Monitor the hit ratio on the local cache, as well as the memory required by the cache when making your decision on local caching.

*Effects of REOPT on dynamic SQL*

You can gain additional optimization for dynamic SQL using the REOPT parameter of the BIND command. REOPT specifies whether to have DB2 determine an access path at run time by using the values of host variables, parameter markers, and special registers. 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.

**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(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.

After migrating to DB2 9, consider specifying REOPT(AUTO) for SQL statements that at times can take a relatively long time to execute, depending on the values of parameter markers. In particular, you should especially consider doing this when parameter markers refer to non-uniform data that is joined to other tables.

Also, consider re-evaluating programs bound specifying REOPT(ONCE). In some cases, switching to REOPT(AUTO) from REOPT(ONCE) can produce performance improvement by reoptimizing when it makes sense, instead of just sticking with a single access path based on the first values supplied to the parameter markers.

### Making Static SQL More Dynamic

Static SQL is bound before it is run. This means that the values for any host variables, parameter markers, or special registers are not taken into account when the program is bound. But sometimes you will want DB2 to factor host variable and special register values into the optimization process. This can be accomplished using the REOPT parameter.

REOPT(ALWAYS) or REOPT(NONE) apply to static SQL, but REOPT(ONCE) and REOPT(AUTO) are not valid for static SQL because DB2 does not cache static plans. The REOPT parameters and to which types of SQL they apply is summarized in the following table.

*REOPT Applicability*

| REOPT Parameter | Dynamic SQL | Static SQL |
|---|---|---|
| NONE | YES | YES |
| ALWAYS | YES | YES |
| ONCE | YES | NO |
| AUTO | YES | NO |

Consider binding with REOPT(ALWAYS) when the values for your program's host variables or special registers are volatile and make a significant difference for access paths. 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.

Be sure to factor in the overhead to prepare the access plan for all the SQL in the program at run time [md] the more complex the SQL the greater the overhead will be. A

typical bind for a single SQL statement likely will require from 1ms to 100ms to execute on a z9. The actual time to bind depends on the complexity of the SQL statement, and some very complex statements may exceed 100ms.

If you have only one or two static SQL statements that would benefit from reoptimization at runtime consider creating separate package. Put the statements that can benefit from reoptimization into a package that can be bound REOPT(ALWAYS) or REOPT(AUTO), and put the remaining statements into a program that can be bound with REOPT(NONE). Doing so will cause your application to incur the cost of reoptimization only for those statements that may benefit.

**Summary**

The days of avoiding dynamic SQL "at all costs" are long behind us. Dynamic SQL is a mainstay at most organizations today because of its use in web-based applications, ERP systems, and by many Java applications.

But keep in mind what we've learned in this article: dynamic doesn't have to be totally dynamic… and static doesn't have to be totally static. You can take measure to make dynamic SQL more static, and static SQL more dynamic.

And that is a good thing!