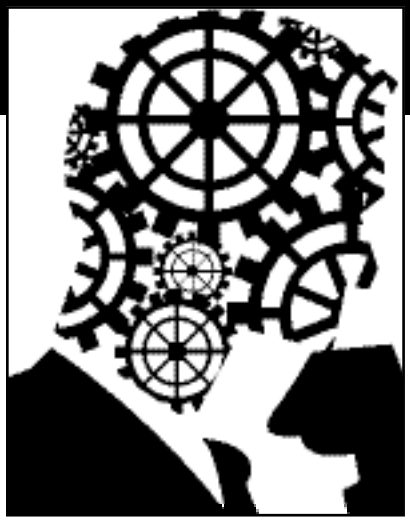

Insurance for Your Access Paths Across REBINDs

BY TERRY PURCELL AND GENE FUH

A known benefit of static SQL, more common in a DB2 for z/OS environment, is that the access path is predetermined and is, therefore, stable. At some point, though, it will become necessary to perform a REBIND, and thus the access paths will be re-evaluated. REBIND may be required at various times, such as after applying DB2 maintenance, upgrading your DB2 release, or to exploit new or updated statistics or indexes. Whatever the reason, you can guarantee that one of three things will happen to the performance of your SQL statements – performance will either improve, remain the same or get worse.

Fortunately, the percentage of access paths that get worse is generally very small, but unfortunately there is some risk involved. As intelligent as the DB2 for z/OS query optimizer is, there are numerous reasons why the optimizer may choose a less than desirable access path. With static SQL, the lack of the "right" statistics can present challenges for the optimizer, as can the presence of host variables that render frequency statistics virtually unusable and turn range predicate estimation into a game of potluck. For example, suppose 99 percent of the values for the Status column are 'Y', and 1% are 'N'. How can the optimizer determine which value will be used for WHERE STATUS = :HV? Range predicates with host variables are even more daunting, even without such skews in the values. Consider, for example, the predicate WHERE BIRTH_DATE < :HV. Will the user specify '9999-12-31', '0001-01-01', or some value in between at run time? The selectivity of the predicate, and hence the plan chosen, will vary greatly depending upon that value, but at compile time, the optimizer has no clue as to what value might be given.

Such unknowns are more easily overcome in a dynamic SQL environment. Ad-hoc SQL generally uses the actual literal values, and dynamic SQL with parameter markers can take advantage of REOPT(ONCE) or REOPT(ALWAYS), whereas static only has REOPT(ALWAYS). And REOPT(ALWAYS) pretty much defeats the purpose of static binding, because it effectively rebinds for every new literal value.

While the DB2 for z/OS query optimizer is continually improving, a REBIND of static SQL moves you from access path stability to the opportunity for change. Wouldn't it be nice if you could fall back to your previous good access path (assuming it's still good) if the new BIND resulted in a less desirable access path? This article will help demonstrate how to provide that kind of insurance on your access paths across a REBIND.

But first – a recap on packages, collections and package resolution.

PACKAGES AND COLLECTIONS

When precompiling your application programs, the SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler, and the database request module (DBRM) that is the input to the SQL BIND process.

The DBRMs are then bound into a package, which are then bound into the application plan. Alternatively the DBRMs are bound directly into a plan. You may even use a combination of packages and DBRMs in your application plan. The DB2 Application Programming and SQL Guide provides more detail, and also lists the benefits of using packages. This article will focus mostly on packages, and how they can be used to provide access path insurance.

BINDing into package collections allows you to add packages to an existing application plan without having to BIND the entire plan again. A collection is a group of associated packages. If you include a collection name in the package list when you BIND a plan, any package in the collection becomes available to the plan.

If the special registers CURRENT SERVER, CURRENT PACKAGE PATH, or CURRENT PACKAGESET are not used, as is generally the case for local applications, DB2 will search for the required package in the package list specified in the BIND PLAN. Otherwise, DB2 uses the value of CURRENT SERVER to determine the location of the required package or DBRM. You can also use either CURRENT PACKAGE PATH or CURRENT PACKAGESET in your application to specify the collections that are to be used for package resolution. The CURRENT PACKAGESET special register contains the name of a single collection, and the CURRENT PACKAGE PATH special register contains a list of collection names.

KEEPING YOUR CURRENT ACCESS PLAN AS INSURANCE

Performing a REBIND and/or BIND REPLACE of the existing package destroys the previous instantiation of the package along with your previous access path, replacing it with the new package and (possibly) new access paths. Once the package is destroyed it cannot be recovered. One can try to restore the previous access path using optimization hints, but wouldn't it be nice to simply fall back to the previous package itself?

To do that, instead of performing a REBIND, perform a BIND into an alternative collection, so that your existing "good" plan is not lost. It is not possible to perform a REBIND into another collection. This must be done with BIND PACKAGE (with or without the COPY option), specifying the list of packages to BIND into the new collection name.

For example, assume package PROG1 is to be rebound, and you want to preserve the current package with its acceptable access plan for fallback purposes. Simply BIND the package with the COPY option from the old collection to the new collection:

BIND PACKAGE (NEWCOLL) COPY (OLDCOLL.PROG1) EXPLAIN(YES)

Alternatively, if you still have the list of BIND commands that were used to BIND from the DBRM into the original collection (OLDCOLL), then this list can be altered to point to the new collection:

BIND PACKAGE (NEWCOLL) MEM BER(PROG1) EXPLAIN(YES)

Regardless of which of the two BIND methods you use, the result is that you now have a copy of the old access plan before the BIND, and a copy as a result of the BIND.

Although it is not mandatory that EXPLAIN(YES) be specified to take advantage of the approach outlined in this article, it is recommended to permit future access path analysis or comparison.

If a precompile/compile/bind of your program is required, then the precompile step will generate a new DBRM and new source. The new program is compiled and link-edited into a new load module. Trying to use the old package/plan with the new load module receives a SQLCODE -805 for packages, or a SQLCODE of -818 for DBRMs bound to plans. Thus, the technique described in this article cannot be used to provide fallback to a prior access plan. This is one situation where the PLAN_TABLE output from the prior BIND can be used with optimization hints to provide access path insurance.

PACKAGE RESOLUTION WITH PACKAGE LISTS

Prioritizing your application plan to select the new collection is very simple if you use package lists in the plan, rather than the special registers in your application. Simply REBIND the plan to replace the package list, specifying the new collection first, followed by the original collection.

For example, the following REBIND plan achieves the desired priority for package resolution:

REBIND PLAN (APPLPLAN)
PKLIST(NEWCOLL.*, OLDCOLL.*)

When executing the application, DB2 will first search collection NEWCOLL to find the package. If not found, the search will resume with collection OLDCOLL. Given the prior BIND PACKAGE examples, the application will find the package with the new access plan in NEWCOLL.

PACKAGE RESOLUTION WITH SPECIAL REGISTERS

If you use the special registers CURRENT PACKAGESET or CURRENT PACKAGE PATH to control package resolution within your application, then these would need to point to the new collection to pick up the new access plan.

For example, the following demonstrates SET commands for either of these special registers:

SET CURRENT PACKAGESET = 'NEW COLL';

SET CURRENT PACKAGE PATH = 'NEWCOLL', 'OLDCOLL';

Note: The effectiveness of being able to fall back to a prior plan are diminished if the application contains hard-coded values for CURRENT PACKAGESET, as this will require application changes whenever the collection changes. CURRENT PACKAGE PATH does offer more flexibility because it only requires changing if a collection is to be added or removed from the path. However, it is preferable that these are parameterized within the application, so the parameter can be altered externally.

FALLING BACK TO THE PRIOR GOOD ACCESS PLAN

At this point, all of the preparation is in place to allow the application to switch from the new application plan to the old application plan quickly and easily.

In the event that the new application plan results in worse performance, the initial priority may be to get the application up and running as soon as possible, without allowing time to analyze the root cause of the access path change.

To fall back to the prior good access plan, simply free the package from the new collection:

FREE PACKAGE (NEWCOLL.PROG1)

Now, when DB2 searches the collections to find the package, using the package list or CURRENT PACKAGE PATH, DB2 will not find the package in NEWCOLL, and will therefore search the original collection (OLDCOLL), in which the package will be found. Thus, performance will be restored to the point prior to the most recent BIND.

If the application has been coded with special register SET CURRENT PACKAGE SET, then this will need to point to OLD COLL instead of NEWCOLL to find the original copy of the package. In this scenario, the FREE PACKAGE command won't be needed.

After freeing this package, you are back to the point you were before the BIND. But all other packages have now been bound successfully. So the insurance you took out for all packages has now paid off. For the regression packages, you have bought yourself some time to analyze the problem access path and to choose the most appropriate solution before attempting to BIND this package again.

VARIATIONS OF ACCESS PATH INSURANCE

The technique in the prior section is reactive, in that you wait until you see a change in performance before doing this analysis. This reactive technique is the recommended approach if you do not have the tools, the

time, or the skill to analyze all new access paths before running the application, or if you do not have a copy of the existing access path to allow before/after comparisons of access paths.

For customers willing to proactively analyze their access paths, who also have a copy of the existing access path, BINDing to a new collection with EXPLAIN(YES) will allow the comparison of old plans to new ones without disrupting the application. Once you are satisfied that all access paths offer the same or improved performance, the new collection can be added to the plan as previously described.

Analyzing the new access path without having an EXPLAIN of the current plan makes identifying regression cases much more difficult. This is one reason why it is beneficial to insure that a copy of at least the current access path is kept in the PLAN_TABLE. Another reason is that optimization hints may be used in many situations as a method of ensuring the good access path can be restored for the new BIND, if necessary.

If you are not using packages, but instead BINDing your DBRMs directly to a plan, then you can still benefit from the access path insurance concept outlined here. Simply don't BIND / REBIND into the same plan, but into a new plan. Of course, a change must then be made so that the application executes the new plan. And this change needs to be reversed if required to revert to the prior plan.

CONCLUSION

This article illustrates that with some planning and preparation, customers can preserve the "previous good access paths," using them as an insurance policy against access path regression.

The concept presented is quite simple "best practice" to allow fallback to the prior access path. Don't REBIND over the existing package, but instead BIND to a new collection. Prioritize your package list, with the new collection followed by the original collection. And in the event of regression, free the package from the new collection, allowing application execution to fall back to executing the original package.

With this insurance, customers who currently avoid rebinds can be more aggressive about exploring the potential for new and improved access paths, without the fear of unrecoverable regressions in performance, often at the most inconvenient times.

While this should provide some comfort to customers with concerns about their access path stability across REBIND, it is particularly timely advice for those about to undertake large scale REBINDs, especially customers wishing to benefit from the many optimization enhancements in DB2 UDB for z/OS V8.

ABOUT THE AUTHORS

Terry Purcell is a senior software engineer in DB2 for z/OS SQL Query Technology Team with the IBM Silicon Valley Lab. He has more than 14 years experience with DB2 in Database Administration and application development as a customer, consultant and DB2 developer.

Gene Fuh is an IBM Distinguished Engineer specializing in the area of Query Technology for DB2 z/OS. He has been working on Database technology for 12 years after four years of service in Compiler development. He received an M.S. degree and a Ph.D. degree, both in Computer Science, from State University of New York at Stony Brook in 1986 and 1989, respectively.