PLANs + COLLECTIONs + VERSIONs + PACKAGEs + DBRMs = Confusion

Revisiting program preparation on DB2

July 15, 2011

by Bonnie Baker

Long ago, I wrote a three-part column to explain the program preparation process on the DB2 for z/OS platform. I received more e-mail referencing that column than all of my other columns put together. Now I'd like to revive that popular series, which is no longer available in the magazine archives. So, with some changes to reflect DB2 10, I am going to share that same information in a condensed column, using the basic preparation process for a COBOL batch program as my basis. This much-wanted information will be back on the Internet, and I will have a link for those who ask.

Write the program

When we write a program that needs data from DB2 tables, we embed SQL in the program at the appropriate execution spot and surround it with delimiters. The delimiters for COBOL are EXEC SQL and END-EXEC. Between pairs of delimiters, we also code INCLUDE member-name (which is similar to coding COPY copybook-name) to specify each member we want brought into our program from the INCLUDE library. The INCLUDE library holds DCLGENs. We include DCLGENs for each referenced DB2 table because a DCLGEN consists of an image of a DB2 table (the way it looks in DB2-speak) and a COBOL working storage structure into which rows can be read and from which rows can be written.

With the advent of multi-row processing, many development groups now have two DCLGENs for each table, one standard DCLGEN and one multi-row DCLGEN in which each column occurs approximately 100 times.

Prepare the program

Once the program is written, you must put it through a process so that it can run and so that it can connect to DB2 whenever it needs to. The first step of this process is to PRECOMPILE your COBOL source code. Both your program source code in your COBOL SOURCELIB and your DCLGENs in your INCLUDE library are input to the DB2 precompiler.

The precompiler does three things. To do these three things it must read your program top to bottom in one pass, looking for your SQL delimiters. If it finds an INCLUDE member-name between the delimiters, it copies the member into your program. If it finds SQL, it does a modest syntax check.

Why modest? Because it has only the table description from the DCLGEN as a reference. You see, the precompiler is not connected to DB2, which means that running PRECOMPILE does not add any DB2 overhead. The third thing that the precompiler does seems contrary to reason: You have very carefully embedded SQL into your COBOL program—the precompiler just as carefully removes it. The precompiler replaces any executable SQL that was in your source with a COBOL CALL to a COBOL-DB2 interface module. Non-executable SQL statements (like your DECLARE CURSORs and the DECLARE portion of the DCLGENs) are simply commented out.

At the end of the PRECOMPILE step, you have two entities—fraternal twins. One twin is the modified COBOL source code with its INCLUDEd members and its calls to DB2, but no SQL. The other twin is a database request module (DBRM) containing all of that SQL that was removed from your program. To make matters more complicated, the precompiler gives these fraternal twins the same name, the name of the program, and they go down separate paths outside of DB2's control.

Eventually, when Twin 1, the COBOL program, runs, it will need to find and identify Twin 2. Therefore, the precompiler "tattoos" each twin with a modified timestamp. This "tattoo" will be used at run time for the COBOL twin (the searcher) to identify the SQL twin (who is waiting to be found) to make sure that both twins came out of the same PRECOMPILE step.

Twin 1: From SOURCE to LOAD

To create a load module, the newly generated, modified COBOL source code must go through the standard COBOL COMPILE and LINK EDIT. One key module that must be LINK EDITed into the program is the COBOL-DB2-TSO batch interface module. The resulting load module (with its "tattoo") is put into a LOAD library where it waits to be executed.

Twin 2: From DBRM to PACKAGE

The SQL in its DBRM now must go through a process similar to COMPILE to produce run-time code. In DB2, the "compile" process is called BIND, and the "load module" is called a PACKAGE. BIND is to your DBRM what COMPILE is to your COBOL. BIND must take your source code (the SQL) and turn it into executable code (a PACKAGE).

To convert your SQL into executable run-time instructions, BIND attaches to DB2 and does three things. It first verifies that you are authorized to do a BIND. Second, it checks the syntax of your SQL. We know that a syntax check may have been done at PRECOMPILE time, but that only happened if we INCLUDEd a DCLGEN. Also, that syntax check was only as accurate as the DCLGEN. This BIND-time check is much more thorough and sophisticated because it will use data from the DB2 catalog. Third, it will perform the most important step: optimization. During this step the DB2 Optimizer will turn our SQL into executable code.

The DB2 Optimizer creates run-time instructions for each SQL statement. It will evaluate your SQL, asking questions like: What columns did you SELECT? What table(s) do you want to read? What conditions are in your WHERE clause? It will check the DB2 catalog for useful statistical information and to see what indexes are available. Using this information, the DB2 Optimizer will determine the cost of using each index. It will also check the sizes of your RIDPOOL and your BUFFERPOOL, find out if you used BIND parameters like DEGREE ANY or ISOLATION UR, and much, much more. The DB2 Optimizer considers all of this information and for each possible path assigns a cost in terms of I/O, CPU, RID SORT, and DATA SORT overhead. Then it will either pick the cheapest access path or (as of DB2 10) a slightly more costly but less risky path. (More on that in a future column.)

Once the path is chosen, the run-time code can be generated and put into a PACKAGE. So let's review the process so far: First, all of the SQL for a single program becomes a single DBRM. That DBRM becomes a PACKAGE (our DB2 load module). And all of these entities have the same name, and that very important "tattoo" has been copied into each.

A copy of the DBRM is stored in the DB2 catalog (our DB2 "SOURCELIB") for documentation and future REBINDs (a topic left for another column). The PACKAGE is written to the DIRECTORY (our DB2 "LOADLIB") where it sits and waits for its twin, the program's load module, to find it.

COLLECTIONs

Many PACKAGEs have characteristics in common. Maybe one group of PACKAGEs is used by the payroll application. Maybe another group is very popular and called by every program regardless of application area. Maybe they share a special, not so common BIND parameter like OPTHINT or REOPT(ALWAYS). DB2 has anticipated this situation by providing a high-level grouping name for your PACKAGEs. This is called a COLLECTION. You specify which COLLECTION you would like each PACKAGE to be part of by using the INTO collection_name clause in your BIND statement.

The reunion

How do our twins reconnect? Often programs call other programs, which then call even more programs. The PACKAGEs for all of these programs may have been bound into the same COLLECTION or different COLLECTIONs. To tell our load module (Twin 1) where to look for the PACKAGE (Twin 2), we give it a list of COLLECTIONs that it is allowed to search, by creating a PLAN. To create the PLAN, we perform another BIND. This BIND (BIND PLAN) specifies the search chain. It lists each COLLECTION that is a possible location for the PACKAGE(s) that will be needed by our EXECUTE PROGRAM statement in our JCL. By providing this PLAN, we tell our COBOL load module exactly where to search for its PACKAGE. And at run time, at the first CALL to DB2, Twin 1 will search for Twin 2 in each COLLECTION named in the PLAN until it finds its twin—with the same name and the same "tattoo." If it does not find its twin, the program will receive a -805 SQLCODE. A -805 means, "I searched for my twin in every COLLECTION that you gave me in the PLAN list, and I could not find him."

Multiple VERSIONs?

Sometimes we would like to keep multiple VERSIONs of the same PACKAGE. For example, in a fallback situation, it would be nice to have both the current VERSION and the prior VERSION of a

PACKAGE available. If we have both, for quick fallback, we could just point to the old load module in our JCL. Maybe we want one VERSION bound with ISOLATION UR and another bound with ISOLATION CS. If we have both, we could run with UR during read-only windows when all of the rows are committed and run with CS when maintenance is being performed and we need to be protected from “dirty” data.

Confusion eliminated

There are infinite possibilities for setting up your PLANs, COLLECTIONs, VERSIONs, and PACKAGEs. I hope this column has not only clarified the confusion but has also helped you see those possibilities.