DB2 MAGAZINE THE ENGINE FOR INFORMATION ON DEMAND

Quarter 4, 2003 Vol. 8, Issue 4

Say What?

By Bonnie Baker

Plans + DBRMS + Packages + Collections + Versions = Confusion

No matter how long programmers have worked with DB2 for z/OS and OS/390, they still ask me to tell them the difference between a plan and a package — and what in the heck a collection is. I planned to write a column on this topic so I could just point them to *DB2 Magazine*, Quarter 4, 2003 and let the column answer the questions. Well one column became two, and two morphed into three. And, if they keep changing DB2 (as they will do), the three may one day become four.

THE EARLY DAYS

The complexity and breadth of this topic is best understood from an historical perspective. Long ago, when DB2 for MVS (now OS/390 and z/OS) was in its infancy, many companies had different machines for development and production. Development often took place on an older model machine, and the LOAD modules were moved onto a bigger production machine. In fact, this process still takes place in some companies if you take a broader perspective — some companies develop software offsite and then run it in entirely different locations for entirely different corporations. Come up with a need and companies will fill that need in unexpected ways.

COBOL was the language of choice in the old days; however, companies rarely had a COBOL license on the production machine. So, DB2 had to allow development on one box *with* a COBOL compiler but without a DB2 subsystem and allow production runs on a different box with a DB2 subsystem but *without* a COBOL compiler. That's how the concepts of the DB2 Precompiler and the database request module (DBRM) were born — out of necessity.

Programmers could write COBOL programs on the smaller, non-DB2 machine. They could embed SQL between the COBOL SQL delimiters (`EXEC SQL` and `END_EXEC`) in both the COBOL working storage and procedural divisions. When they were ready to compile the program to look for COBOL errors, they'd run the program through the DB2 Precompiler to strip out the SQL, leaving only COBOL.

THE PRECOMPILER

The DB2 Precompiler does *not* need DB2 to run. It carries out three primary tasks as it reads the program serially, top-to-bottom, looking for DB2 delimiters.

First, if the delimiters surrounded an `INCLUDE` statement, the Precompiler goes to the `INCLUDE` library named in the job control language data definition statement and pulls the included `MEMBERNAME` into the program. This *function* is the same as a COBOL `COPY MEMBERNAME`, but the timing is different. COBOL `COPYBOOK`s get copied in at `COMPILE` time; DB2 `INCLUDE`s get copied in at precompile time. The only difference between an SQL `INCLUDE` and a COBOL `COPY` is timing. The most common item `INCLUDE`d in a program was (and is) a `DCLGEN`. `DCLGEN`s are structures that describe a table. One `DCLGEN` is usually

included for each table that the program will access at run time. Each DCLGEN is a two-part structure consisting of a DECLARE TABLE statement, which describes the table in DB2 SQL language, and a COBOL structure that describes the table using an 01-Level COBOL working storage structure (much like a typical copybook for a VSAM file).

Second, if the delimiters surround an SQL statement, the precompiler does a very basic syntax check to make sure that the column and table names are valid (that they're spelled correctly and that the columns and the table exist). Many DBAs and programmers think that this validation is done by reading the DB2 CATALOG, but they're wrong. Remember, the precompiler doesn't need DB2 or its CATALOG. DB2 might not even be installed on the machine. The DB2 Precompiler uses the top part of the DCLGEN to validate the SQL syntax.

Resources

DB2 for z/OS and OS/390

The third, and most important, task performed by the DB2 Precompiler is to split the program into two parts: a COBOL and a DB2 part. All of the SQL that the programmer carefully embedded is stripped out of the program and put into its own partitioned data set (PDS) member, called a DBRM. A single program containing two languages, COBOL and SQL, goes into the DB2 Precompiler and two pieces come out. Twins, but fraternal twins — much like Arnold Schwarzenegger and Danny DeVito. Arnold looks just like his COBOL mother, and Danny looks just like his DB2 father. COBOL Arnold, with all of the SQL commented out, goes down one path in life. SQL Danny, containing only SQL, goes down a different path in life.

The twins, separated at birth, have a tendency to lose each other. To help the twins find each other later in life (in other words, at run time), the precompiler engraves each with identical tattoos. The tattoo is carried forward with COBOL Arnold, through compile and link edit, into the LOAD module in the LOAD library. The tattoo is part of the run-time executable code of the LOAD module. The same tattoo is carried forward with SQL Danny through BIND. BIND is to SQL what COMPILE is to COBOL. The purpose of COBOL COMPILE is to come up with run-time code for the COBOL. The purpose of BIND is to come up with run-time executable code for the SQL. Both sets of code bear identical tattoos (timestamps or consistency tokens).

So, the COBOL twin becomes a transportable load module in the COBOL LOADLIB and the SQL becomes a transportable DBRM in the DBRMLIB. Just as the COBOL twin had to be compiled, the DBRM twin has to go through BIND to create the run-time executable code for the DB2 portion of the COBOL program and put that executable code into the "right" DB2 subsystem.

ALL ABOUT BIND

BIND connects to the DB2 in which the program's LOAD module will run, reads the DBRM serially, and then performs three tasks.

The first of the BIND tasks is an authorization check. DB2 must make sure that the programmer has the BIND authority *and* the SQL authority to perform the requested SQL task (for example, updating the payroll master). When using standard authorization procedures, DB2 won't let you BIND a DBRM if you don't have the authority to execute the SQL that's in the DBRM. This is why you may have the authorization to BIND in development (accessing development tables) but don't have authorization to BIND in production, where the SQL accesses production tables. The second BIND task is a bit redundant. BIND, like precompile, must also check the syntax of the SQL, but the BIND check is more sophisticated. Instead of using the top, DECLARE TABLE portion of the DCLGEN, BIND uses the DB2 CATALOG table information to make sure that the column names are valid, that comparisons are numeric-to-numeric, and so on. This second syntax check occurs because you can't trust the one done by the precompiler because the precompiler check used the DCLGEN. You could have a DCLGEN and not have the DB2 table.

The third, and most important, BIND task is to come up with run-time instructions for the SQL in the

DBRM. Each SQL statement is parsed and all of the possible (realistic) methods for retrieving the desired columns and rows from the table are weighed, measured, and evaluated based on possible estimated I/O, CPU, and SORT overhead. A ton of information is used as input to the BIND process, not just CATALOG information put there by running the RUNSTATS utility. BIND input includes, for example:

- Indexes (what columns are in the indexes?)
- Columns (how long is this column and how much room will it occupy in a SORT record?)
- System resources (how big are the system resources, buffer pool, and RIDPOOL?)
- Processors (how big are they and how many engines do they have?)
- DB2 (what release is running?)
- Parameters (what are the values of the BIND parameters?)

After all that input (and more) is weighed and compared, the cheapest, most costeffective access path is chosen, and the runtime instructions for that one path are created. (Interestingly, DB2 BIND sometimes generates instructions for more than one path.) This process is called optimization, and it's repeated for each SQL statement in the DBRM until all access paths are decided and the run-time instructions are created for each. As the optimizer decides on each path, writes are done to DB2.

BIND checks to see if you bound with the parameter EXPLAIN(YES); if so, it writes documentary evidence about the chosen path to the PLAN_TABLE and to the DSN_STATEMNT_TABLE for your edification.

BIND also writes a lot of information to multiple CATALOG tables, documenting the fact that the BIND did occur. In fact, the tattooed DBRM, which is not used at run time, is moved into the CATALOG. Objects chosen by the optimizer are documented in the CATALOG in cross-reference tables. And BIND parameters are recorded in the CATALOG also.

WHERE ARE THE INSTRUCTIONS?

It's interesting that the actual instructions for the access path are not written to the CATALOG. You can't look at information in the DB2 CATALOG to figure out whether a query will do synchronous or asynchronous reads at run time. You can't tell if the query will match on three columns of an index or five. The actual run-time instructions aren't stored in the CATALOG. They're definitely not in the DBRM, which is input to the BIND. So, where are they stored?

This question is one of the many reasons that this column grew into two parts. Where in the heck are the run-time instructions? Should you ever use the PLAN_TABLE at run time? Are the run-time instructions in a package, a plan, or a version? All of the above? The infamous "it depends"? Where are the instructions stored while you wait for the LOAD module to run? How has this process changed over time? What program preparation options are currently available in version 7? What's coming in version 8?

Stay tuned for answers to these and other questions.

Bonnie Baker is a consultant and educator specializing in applications performance issues on the DB2 OS/390 and z/OS platforms. She is an IBM DB2 Gold Consultant, a five-time winner of the IDUG Best Speaker award, and a member of the IDUG Speakers' Hall of Fame. She is best known for her series of seminars entitled "Things I Wish They'd Told Me Eight Years Ago" and the Programmers Only column in *DB2 Magazine*. You can reach her through Bonnie Baker Corp. at 813-876-0124 or

_____.

Return to Article

DB2 MAGAZINE THE ENGINE FOR INFORMATION ON DEMAND

Quarter 1, 2004 Vol. 9, Issue 1

Say What? Part 2

By Bonnie Baker

How the separated twins - modified source code Arnold and DB2 SQL Danny - find each other again.

In the last issue of *DB2 Magazine* I began writing about the program preparation process for DB2 Universal Database (UDB) for OS/390 and z/OS. If you haven't read Part 1, please do so now. Part 2 will make much more sense if you do. If you're still confused after reading Parts 1 and 2, maybe, just maybe, next quarter's Part 3 will clear up any remaining confusion and make you a program preparation expert.

By the end of Part 1, we had done a `PRECOMPILE` to separate our COBOL program into twins: Arnold (that's Governor Arnold now) as the modified source code (the SQL commented out and `DCLGEN`s now `INCLUDE`d) and Danny as the DBRM containing the SQL that used to be in the source code. The COBOL source code (without any SQL) was compiled into run-time executable instructions for the COBOL portion of the program; DBRM Danny (all the SQL that used to be in the source module) went through `BIND` to generate run-time instructions for the SQL in our COBOL program.

I didn't mention how Arnold, over in the COBOL `loadlib`, will find his long-lost twin — nor did I mention exactly what we were binding and where we would put it when we finished. And that's what this column is about.

The Modified Source Code (Arnold)

At precompile time, when the SQL was stripped out of our program and moved into the DBRM (Danny) leaving only COBOL in the modified source code, you must have wondered how COBOL Arnold would ever find DB2 SQL Danny; in other words, how the COBOL would ever execute any SQL.

The explanation is simple. All of the executable SQL (and not all SQL is executable — a `DECLARE CURSOR` isn't, for example) in the COBOL program was replaced with a `CALL` statement (in our example, a COBOL `CALL`). The modified source code, complete with its calls to DB2 SQL Danny, was compiled and "linked" into a `LOAD` module. When this `LOAD` module is executed and hits a paragraph that once contained SQL (but now contains a `CALL` to Danny), there will be run-time executable code that knows how to link to a COBOL-DB2 interface module and connect to DB2, where it will find Danny and the run-time executable code for the SQL statement that previously had been in the paragraph.

Remember that we tattooed the twins? Well, this `CALL` contains the information DB2 needs to confirm not only that this `LOAD MODULE` is Arnold (complete with requisite tattoo), but also that it's the exact same Arnold that came out of the exact same precompile step as Danny. The `CALL` looks at the `PLAN` named in the execute statement of the job control language (JCL) and searches for the Danny out in DB2 with the same tattoo.

The DBRM (Danny)

At BIND time, DB2 created run-time executable instructions for the SQL portion of the program. But where are those instructions, and what are they called now that the term DBRM no longer applies?

The truth is, you have a choice. You can BIND the instructions for the SQL that was in the DBRM into a PLAN (the old way), or you can BIND the instructions into a PACKAGE (the not-so-old-but-no-longer-new way). The reason for this choice is historical. Back when knights were bold, dragons walked the Earth, and DB2 and some of us were young, DBRMs were bound directly into PLANs. In today's DB2 (since V2R3), there are two ways of doing BINDs. You may continue to BIND DBRMs into PLANs, or you may BIND DBRMs into PACKAGEs. With the second option, you keep your PLAN but use it only as a search chain. This column explains why things changed in V2R3. A future column will explain how things have continued to change in v.7 and v.8.

In the early releases of DB2, the DBRM (SQL originally embedded in our COBOL program but separated at precompile time into a PDS member) was bound into a PLAN (PLANA). This method worked just fine as long as the program was a standalone program. You coded the JCL to execute program PGMA naming the PLAN "PLANA," and at run time the twins found each other. However, things got a bit complicated when PGMA needed to CALL PGMB. Because only one PLAN could be named in an execute statement, the PLAN had to contain run-time instructions for both PGMA and PGMB. This problem was solved by having the BIND instruction for PLANA name a memberlist; the DBRMs for both PGMA and PGMB were listed as members. And if PGMB called PGMC, then the three would be listed as members. And if C called D, which could call E, F, G, or H, which could call I, J, K, L, or M, which could .... Well, you get the idea.

Memberlists got longer and longer. What were (and still are, if you cling to the old technique) the drawbacks of having a very long list?

1. Remember that DB2 authorization and SQL syntax are checked at BIND time. Access path alternatives are also weighed, the least-cost path is chosen (based on current statistics and system resources), and run-time instructions are created for the chosen path. Well, if the PLAN contains one member, A, this process should be quick. But what if there are 20 or 50 or 500 DBRMs in the memberlist? The BIND could take hours.
2. So, the PLAN, which took more than a while to BIND, contains 500 members. What if one of the programs, PGMA, changes? When the source code changes, the program must be precompiled. And precompile changes the tattoo and creates a new DBRM. That new DBRM must be bound into the PLAN. When the PLAN is bound, all 500 DBRMs (not just the modified PGMA) will be rebound. It could take hours to BIND a PLAN, even though 499 of the 500 programs haven't changed.
3. Also, remember that BIND is an opportunity to reassess and change the access paths for not only the modified program, but also every single program in the memberlist. If one program changes, every program in the list will go through BIND.
4. What if you want to modify PGMB to call a new program, PGMZ? You must not only precompile modified B and new Z, you also must add new PGMZ to the memberlist and BIND the whole list — all 501 DBRMs.
5. You want to remove PGMT? Edit the memberlist and then BIND the PLAN again with the remaining 500 members in the list and wait impatiently while DBRMs that haven't changed go through the BIND process.
6. Okay, modified program A turns your processor over on its back, casters up. You want to fall back to the original version of PGMA. And exactly how would you do that quickly? If (and it's a big if) you have the old DBRM with its old tattoo, you could move it into the DBRMlib and BIND the entire 500-member list (even though only A had regressed) and replace the new loadlib member A with the prior loadlib member A. Or, if you have the old source code for A, you could precompile it to recreate both the modified source code and the DBRM, and then COMPILE, LINK, and BIND, which would BIND all 500 DBRMs in the member list.
7. Program Q has to run against two sets of tables, one for Corporation 1 and a second for

Corporation 2. The sets of tables have identical names but different high-level qualifiers. You could use synonyms, but they're unwieldy; binding during a time when a synonym points to the wrong set of tables could cause disaster.

These (and other) quandaries faced many DBAs in the days before V2R3 and the advent of PACKAGEs.

Packages and Collections

Resources

DB2 UDB for z/OS and OS/390

"Say What?" Part I, Quarter 4, 2003

In V2R3, the DB2 developers solved the seven problems I listed (and many more) by introducing another layer in the program preparation procedure. The precompile step still split the program into tattooed twins, and the treatment of the modified source code stayed the same. But the DBRM could now be bound either into a PLAN (the old way) or into a PACKAGE. Although the relationship between a DBRM and a PLAN was one-to-many, the relationship between a DBRM and a PACKAGE was always one-to-one. Most of the work of BIND PLAN was moved into BIND PACKAGE. Therefore, if PGMA changed, only PACKAGE A would have to be bound.

If only one DBRM could be bound into a PACKAGE, but PGMA could still CALL PGMB, then a structure was needed to gather all of the PACKAGEs into a searchable list. This structure became a packagelist, rather than a memberlist, bound into a PLAN.

This tiny change solved many, but not all, of the problems inherent in the "memberlist of DBRMs bound into a PLAN" technique. To solve a few more problems, IBM introduced the concept of COLLECTIONs. A collection is simply a way of grouping packages into meaningful (for you) groups. You could BIND all of the packages that must run against Corporation 1's tables into one COLLECTION and all of the packages that must run against Corporation 2's tables into another. Or, you could use COLLECTIONs to separate programs for different application areas, such as payroll and inventory.

Another use might be to separate programs bound with ISOLATION UR from programs bound with ISOLATION CS. COLLECTIONs are simply high-level grouping names to designate that this group of packages share something, anything, in common.

So, now, with the introduction of COLLECTIONs, our BIND PLAN process (think search chain) can now include a packagelist of fully qualified package names such as COLLPAYROLL.PGMA, COLLPAYROLL.PGMB, and COLLCOMMON.PGMX. Or, if we choose, we can just substitute an asterisk for the program name and list (COLLPAYROLL.*, COLLCOMMON.*). Then any program bound into the COLLECTION will be accessible by the named PLAN.

Where Are Packages and Plans?

When you BIND a DBRM into a PLAN, or BIND a PACKAGE into a COLLECTION and then the COLLECTION(s) into a PLAN, the information must be stored somewhere safe inside DB2 until it's needed at run time. These items (PLANs and PACKAGEs) aren't stored in the DB2 Catalog. Rather, they're stored in the DB2 Directory. In fact, you can think of the Directory as the DB2 loadlib for the Danny portion of your program, complete with tattoo.

Run Time

At run time, the load module starts up and eventually hits a paragraph containing a CALL to DB2. This CALL contains information such as a description of the tattoo, the content of your SQL host variables (now populated), the statement number, and so on. The CALL invokes the COBOL-DB2 interface program, which connects to DB2. And if the run-time code necessary to execute your SQL isn't currently resident inside DB2 (in the EDMPOOL), we go to the buffer pool (BP0) assigned to the DB2 Directory and

look there. If we don't find Danny there, we go to VSAM to disk to look in the COLLECTIONs named in the PLAN for the PACKAGE with the same name and the same tattoo, also known as the consistency token or timestamp.

And if you don't find the twin anywhere in DB2 (not that this has ever happened to anyone reading this column), you get a -805 error. If you're still using the older technique of binding DBRMs directly into PLANs via a memberlist, then an unsuccessful search for Danny will result in a -818 error code.

But What About Versions?

In my next column, I'll give you some ideas about the various ways of using the concept of the COLLECTION. I'll also write about VERSIONs and how they're used. Version 7 introduced some more useful variations that will carry us gracefully into v.8, and I'll cover them all next quarter.

Bonnie Baker is a consultant and educator specializing in application performance issues on the DB2 OS/390 and z/OS platforms. She is an IBM DB2 Gold Consultant, a five-time winner of the IDUG Best Speaker award, and a member of the IDUG Speakers' Hall of Fame. She is best known for her series of seminars entitled "Things I Wish They'd Told Me Eight Years Ago" and the Programmers Only column in *DB2 Magazine*. You can reach her through Bonnie Baker Corp. at 813-876-0124 or _____.

Return to Article

DB2 MAGAZINE THE ENGINE FOR INFORMATION ON DEMAND

Quarter 2, 2004 Vol. 9, Issue 2

Say What? Part 3

By Bonnie Baker

What to do when you find yourself in a bind.

Two columns ago, I wrote about the way we used to do program preparation for DB2 for z/OS and OS/390. Therefore, if you haven't read Part 1 of this series, please do so now (Programmers Only, Quarter 4, 2003). And while you're doing that, you should also read Part 2 (Programmers Only, Quarter 1, 2004). I promise that this column will make much more sense if you do. And if you're still confused after reading the two prior columns and this column, let me know via email, and I'll do a Part 4 just for you.

When we finished Part 1, we had done a PRECOMPILE to separate our COBOL program source code into twins. Twin 1 was Arnold (oops, Governor Arnold now), the modified source code (the SQL now commented out and DCLGENs now INCLUDEd). Twin 2 was Danny, the DBRM containing the SQL that used to be in the source code. The COBOL Arnold source code (with all the SQL commented out and the executable SQL replaced by a COBOL CALL to DB2) was compiled into runtime executable instructions for the COBOL portion of the program and became a load module in the COBOL loadlib. DBRM Danny (all the SQL that used to be in the source module) went through BIND to generate runtime instructions for the SQL and became a PLAN in the DB2 Directory database.

By the end of Part 2, we had learned how this program preparation process changed in DB2 V2R3. We learned how Arnold over in the COBOL loadlib found his long-lost twin, Danny, at runtime. We also found out about the difference between a PACKAGE and a PLAN. And, we learned that PACKAGEs are stored in COLLECTIONs. Finally, we learned where these things are stored when waiting for the lead program to be run.

In this column, I'll give you some ideas about the various ways of using the concept of the COLLECTION. I'll also write about VERSIONs and how they are used. DB2 v.7 introduced more useful variations that will carry us gracefully into v.8. I'll give you the high-level view.

Resources

DB2 UDB for z/OS and OS/390

Say What, Part 1

Say What, Part 2

Collection Ideas

In Part 2, I said that a COLLECTION is simply a way of grouping PACKAGEs into meaningful (for you) groups. You could use COLLECTIONs to separate programs for different application areas, such as payroll and inventory. Another use might be to separate programs bound with ISOLATION UR from programs bound with ISOLATION CS. COLLECTIONs are simply high-level grouping names to designate that this group of packages share something, anything, in common.

COLLECTIONs enable you to organize your PACKAGEs into like-kind groups. In DB2's younger days, with multiple DBRMs being bound into PLANs, all the DBRMs in a single PLAN had to be bound with the same

BIND parameters. However, today you can BIND each PACKAGE into a COLLECTION that has a customized set of BIND parameters associated with it. An example would be to ISOLATE all programs using REOPT(VARS) in one COLLECTION and all programs using OPTHINT in another. DEGREE(ANY) is another BIND parameter that you may want to be a bit more vigilant in monitoring. An easy way of keeping an eye on programs (or children) is to put them in a room together. In other words, BIND parameters are much more granular today than they were when DB2 was young.

With the advent of BIND PACKAGE and the one-to-one relationship of a program to a package, we were given the ability to name the high-level qualifier for the tables accessed by the program. Therefore, the DBRM for one program could be bound into two different COLLECTIONs. The DBRM for program ABC123 could be bound into a COLLECTION called colcorp01, using corp01 as the table high-level qualifier. The same DBRM could be bound into a COLLECTION called colcorp02, using corp02 as the high-level qualifier. Or, you could BIND the same DBRM into colstress to run it against stress test tables and BIND it into colstress to run against regular test tables. Or, you could BIND a DBRM into a COLLECTION called colur to use when you access read-only decision support tables using ISOLATION UR and into a COLLECTION called colcs when you use active production data. There are dozens of examples. Just use your imagination.

So, at runtime, whichever approach you chose, you now have a PACKAGE with the exact same tattoo/timestamp/consistency-token in two different COLLECTIONs. How do you tell DB2 in which collection to search for Danny? Normally, DB2 would search through all of the COLLECTIONs in the named PLAN. But, if you want to search only one COLLECTION, you simply tell DB2 in your program. You can specify which COLLECTION to search by using an SQL statement called SET CURRENT PACKAGESET. PACKAGESET is simply a synonym for COLLECTION. Therefore, if you set the current PACKAGESET to colcorp01, you will access corp01's tables. If you set the current PACKAGESET to colcorp02, you will access corp02's tables. And the beauty of this is that you only have to maintain one program.

Versions

In Part 1, I mentioned the need for rapid fallback. Suppose program A is changed and moved back into production. Before the program was changed it ran in 10 minutes and never bothered anyone. After the change, all the other programs running at the same time are experiencing -911 timeouts. How do you fall back gracefully and rapidly to the prior version of the program?

At precompile time you can specify a VERSION ID. If the VERSION ID is the same as the current version, BIND will overlay the PACKAGE in its COLLECTION. But, if the VERSION ID is different from the current VERSION ID, BINDing the DBRM will produce a new PACKAGE that won't overlay the prior PACKAGE for the program. You'll have two PACKAGEs for the same program in the same COLLECTION. If you also move the current LOAD module with its old tattoo timestamp into a different loadlib (COBOL.BACKUP), the compile will not overlay it. Then, when you compile the modified source code, you'll have a LOAD module with the new tattoo/timestamp in the current loadlib. If you execute the new LOAD module, you'll find the new PACKAGE. If the system suffers, you can cancel the job and move the old LOAD module back into production simply by pointing to COBOL.BACKUP.

Copies

There are too many other nuances and possibilities to mention; however, one feature that may be useful is the ability to copy a package from one collection to another. If the statistics on your table vary greatly from daytime to evening or beginning of the month to end of the month, you can BIND a PACKAGE in a COLLECTION called colday or colbegin when the statistics in the CATALOG are representative of your daytime or beginning of the month table. You can then COPY that PACKAGE into another COLLECTION called colnight or colend when the statistics in the CATALOG are representative of your nighttime or end of the month table. COPY does a REBIND and uses the DBRM in the CATALOG as its input. Therefore, the

tattoo/timestamp doesn't change. If you check the time of day or the day of month at the beginning of the program, you can SET CURRENT PACKAGESET to the appropriate COLLECTION for DB2 to search for Danny.

Versions 7 and 8

In v.7, the developers introduced a new way of doing program preparation. The newer compilers (such as the Enterprise COBOL V3 compiler) let you skip the stand-alone PRECOMPILE. Why? This new compiler has an application program interface (API) to the DB2 precompiler. As a result, there is one less step in the program preparation process. The COMPILER is now more aware of your DB2 calls, and your SQL can take advantage of some of the host language features. For example, your DCLGENs can now have COBOL structures that look more like COBOL copybooks. The DCLGENs, once limited to 01 and 10-level fieldnames can now have field definitions that are more than two-levels deep. A 10-level trandate can be broken into 15-level year, editchar, month, editchar, day. Also, the DCLGEN fields can contain OCCURS clauses, which is much more programmer friendly.

This new capability carries us gracefully into the recently announced DB2 v.8 for z/OS. If you want to exploit the new multi-row fetch in v.8, you must fetch your rows into array structures. With the new compiler and its API, DCLGENs may now have the needed arrays.

Got it?

For those programmers who are just beginning their work with mainframe DB2, the multiple layers and possibilities of DBRMs, PLANs, PACKAGEs, COLLECTIONs, and VERSIONs can be confusing. The seemingly never-ending timestamp errors can be frustrating. I hope these three columns will clear up some of the questions about these DB2 entities and introduce possibilities never before considered.

Bonnie Baker is a consultant and educator specializing in application performance issues on the DB2 OS/390 and z/OS platforms. She is an IBM DB2 Gold Consultant, a five-time winner of the IDUG Best Speaker award, and a member of the IDUG Speakers' Hall of Fame. She is best known for her series of seminars entitled "Things I Wish They'd Told Me Eight Years Ago" and the Programmers Only column in DB2 Magazine. You can reach her through Bonnie Baker Corp. at 813-876-0124 or _____.

Return to Article