Information Management software

IBM

DB2 9 for z/OS: Packages Revisited

- Learn all about packages, collections, plans
- Manage local and remote packages
- Understand and maintain access plan stability

Paolo Bruni
Sean A. Dunn
Howard Hirsch
Norihiko Nakajima
Suresh Sane

Redbooks

ibm.com/redbooks

International Technical Support Organization

DB2 9 for z/OS: Packages Revisited

March 2009

SG24-7688-00

Note: Before using this information and the product it supports, read the information in “Notices” on page xix.

First Edition (March 2009)

This edition applies to Version 9.1 of IBM DB2 for z/OS (program number 5635-DB2).

Contents

Figures

Examples

Tables

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

CICS®
DB2 Connect™
DB2®
developerWorks®
Distributed Relational Database Architecture™
Domino®
DRDA®
IBM®
Informix®
Lotus®
OMEGAMON®
OS/390®
RACF®
Redbooks®
Redbooks (logo) ®
RETAIN®
Tivoli®
WebSphere®
z/OS®
z9®

The following terms are trademarks of other companies:

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

Java, JDBC, JDK, JRE, JVM, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

DB2® packages were introduced with DB2 V2.3 in 1993. During the 15 years that have elapsed, a lot has changed. In particular, there is a more widespread use of distributed computing, Java™ language, new tools, and upgrades in the platform software and hardware. The best practices back then just might not be optimal today. In this IBM® Redbooks® publication, we take a fresh look at bringing packages into the 21st century.

We begin with an overview of packages and explain the advantages of using packages. Because database request module (DBRM) based plans have been deprecated in DB2 9, you need to convert to packages if you did not use packages already. We provide guidance on using a DB2 provided function to convert from DBRM-based plans to packages.

We re-examine the application development frameworks for packages: program preparation, package setup, and execution. For distributed applications we include a discussion of a utility to identify and remove deprecated private protocol and converting to DRDA® as well as an introduction to the new pureQuery function of Data Studio. We also discuss common problems and their resolutions.

We then explore administration and operational activities dealing with packages, including security, access path management (where we discuss the newly introduced package stability feature to allow for a seamless fallback), and management and performance aspects.

The appendixes include useful queries and mention tools for managing packages effectively.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Paolo Bruni is a DB2 Information Management Project Leader with the ITSO since 1998. He is based in Silicon Valley Lab, San Jose. Paolo has authored many IBM Redbooks publications about DB2 for z/OS® and related tools and has conducted workshops worldwide, mostly on DB2 performance.

Sean A. Dunn is a DB2 consultant with NFF Inc., based in Washington DC. Sean has worked with DB2 since 1985 as a Developer, a DBA, and a DB2 Systems Programmer. He has consulted on DB2 projects in the UK, Australia, Switzerland, Saudi Arabia, and the USA. For the last seven years, he has been working with the United States government providing support for DB2 installation, DBA tasks, DB2 and system performance, database design, and application architecture. He has presented at various conferences, including International DB2 Users Group (IDUG).

Howard Hirsch is a Senior IT specialist with IBM in the United States. He has 29 years of experience in information technology. He has been involved as a DB2 DBA for 21 years. He has worked at IBM for two years. His areas of expertise include database administration and SQL performance tuning.

Norihiko Nakajima is an IT specialist working for IBM Global Services Japan Solution and Services Company (ISOL) since 2003. In these five years, Nori provided technical support for DB2 application development and system administration at a banking financial information system integration project. He has also worked on DB2 application development and devised techniques for package tuning.

Suresh Sane is a Database Architect with DST Systems in Kansas City, Missouri, USA. He has co-authored the IBM Redbooks publications, *Squeezing the Most Out of Dynamic SQL*, SG24-6418, *DB2 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7083, and *Data Integrity with DB2 for z/OS*, SG24-7111. He has presented worldwide and belongs to the International DB2 Users Group (IDUG) User Speakers Hall of Fame for having won the Best User Speaker award twice. He holds a bachelor's degree in Electrical Engineering from the Indian Institute of Technology, Mumbai, India, and an MBA from the Indian Institute of Management, Kolkutta, India.

Special thanks to Rob Crane for helping in getting this project jump-started.

Thanks to the following people for their contributions to this project:

Rich Conway
Bob Haimowitz
Emma Jacobs
International Technical Support Organization

Willie Favero
Jerry Goldsmith
Maggie Lin
Roger Miller
Fred Orosco
Jim Pickel
Tom Ross
Hugh Smith
Jay Yothers
Li Xia
Dan Weis
Li Zhang
IBM Silicon Valley Lab

Mike Todd
DST Systems, Kansas City

Rick Butler
Bank of Montreal (BMO) Financial Group, Toronto

Rob Crane
FedEx Freight System, Colorado Springs

Angelo Sironi
DB2 for z/OS Consultant, Italy

Thanks to the authors of the previous book on DB2 packages, *DB2 Packages: Installation and Use*, GG24-4401-00, published in October 1993:

- Viviane Anavi-Chaput
- Terry Gallagher
- Anthony June Wen Kwang
- Mika Suzuki

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review IBM Redbooks publications form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition might also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-7688-00
for DB2 9 for z/OS: Packages Revisited
as created or updated on September 7, 2022.

March 2009, First Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

The revisions of this First Edition, first published on March 2 and updated June 2, 2009, reflect the changes and additions described below.

February 2012, First Update

This revision reflects the addition, deletion, or modification of changed and new information described below. Change bars reflect the changes.

Changed information

- Corrected text and in section “Detecting number of copies saved for a package” on page 219.
- Corrected Example 10-1 on page 219 to reflect the presence of versions.

New information

- Added clarification for Figure 10-4 on page 218.

August 2012, Second Update

This revision reflects the addition, deletion, or modification of changed and new information described below. Change bars reflect the changes.

Changed information

- Corrected the sequence of chapters and appendixes.
- Corrected Example 4-7 on page 55 with the right table names.

New information

- None.

Part 1

Introduction to packages

In this part of the book, we provide an introduction to packages.

We first define what packages are. We then discuss some of the options available during the BIND process. Next we cover the various steps involved in the execution of the package. Finally, because DBRM-based plans have been deprecated and might not be supported in some future version of DB2, we discuss the migration to packages, especially the utility to automate this migration.

This part includes the following chapters:

1

Overview of packages

In this chapter, we introduce packages. We describe what they are and what benefits they bring. We also show how packages are dealt with during the program preparation phase.

Here we mainly deal with packages of local applications on z/OS. See Chapter 6, “Developing distributed applications” on page 101 for packages of distributed applications.

We cover the following topics:

- The plan
- What a package is
- Package benefits
- Program preparation process
- DB2 precompiler

1.1 The plan

The *application plan* is a flexible and dynamic control structure that is produced during the bind process. DB2 uses the application plan to process SQL statements that it encounters during statement execution.

For applications on z/OS, a BIND uses the database request module (DBRM) from the DB2 precompile step as input and produces either a package or an application plan. It also checks the user's authority and validates the SQL statements in the DBRM.

Plans can have one or more DBRMs directly bound into them, packages or both. The list of packages in the plan is called a PKLIST (package list).

Figure 1-1 shows plan, collection, and package relationship.

- A plan points to many collections.
- A collection contains one to many packages and package versions.
- A package path points to many collections to search for execution of packages.

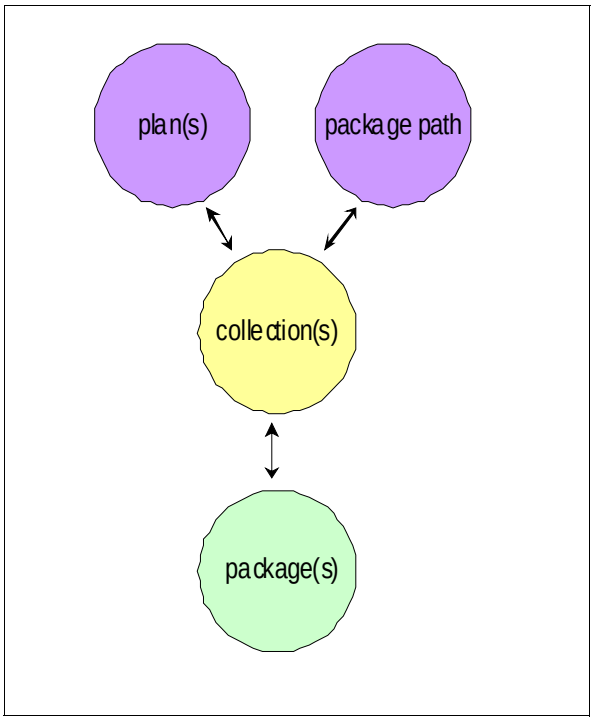

Figure 1-1 Relationship among plans, collections, and packages

1.2 The package

In this section we explain what a package is, and what it is used for.

What a package is

A *package* is an object containing a set of SQL statements that have been statically bound and that is available for processing. A package is sometimes also called an *application package*.

What a package is used for

A package is used by an application program to execute SQL statements. The package is part of the database layer of a program and is separate from the application logic. The separation between the application layer and the database layer is an important part of database application processing. This separation allows for a program to be able to access data at various levels of the development and testing process, as well as to execute against the production database. The program remains unchanged throughout the process, but the package might be pointed at the different environments.

The package also contains an optimized access path. The process that creates the package is called the *BIND* process (BINDing also creates an association between the package and the *load module*).

When the package is bound, DB2 evaluates all of the possible options to access the data, chooses the least costly path to the data, and stores that path in the package for each SQL statement.

Note that remote applications do not use DBRMs and do not go through the DB2 precompiler.

1.3 Package benefits

A number of significant benefits are associated with the use of packages. Packages provide enhancements in application availability, program backout, and application enabling. In this section we discuss the possible benefits that users can derive from packages:

- Reduced bind time
- Reduced need for BIND PLAN
- Support for multiple copies of programs
- Increased application availability
- Granularity in bind options
- Access path persistence
- Simplified access to mirror tables
- Ability to support remote access with static SQL
- Ability to fall back from access path regression
- Ability to use routines (stored procedures, UDFs, and triggers)

A major benefit is the access control provided by packages. The person with EXECUTE authority on the package is not required to own the objects privileges of the package OWNER. See Chapter 9, “Security considerations” on page 173 for details.

Reduced bind time

Implementing packages on z/OS allows a single DBRM to be optimized. When there is a change in a program, the package associated with the program is the only package that needs to be bound or rebound. No other packages associated with the plan are impacted This is very beneficial in an environment where the package is shared in many plans or where there is a considerable number of packages in one plan. Contention for the DB2 catalog, the workload, the amount of time the plan is unavailable, and the time needed for binding are kept to a minimum. In the example in Figure 1-2, when PROGB (included in PLAN 1 and PLAN2, is modified, COL1.PROGA and COL1.PROGC do not need to be bound. PLAN1 and PLAN2 automatically pick up the updated package for PROGB.

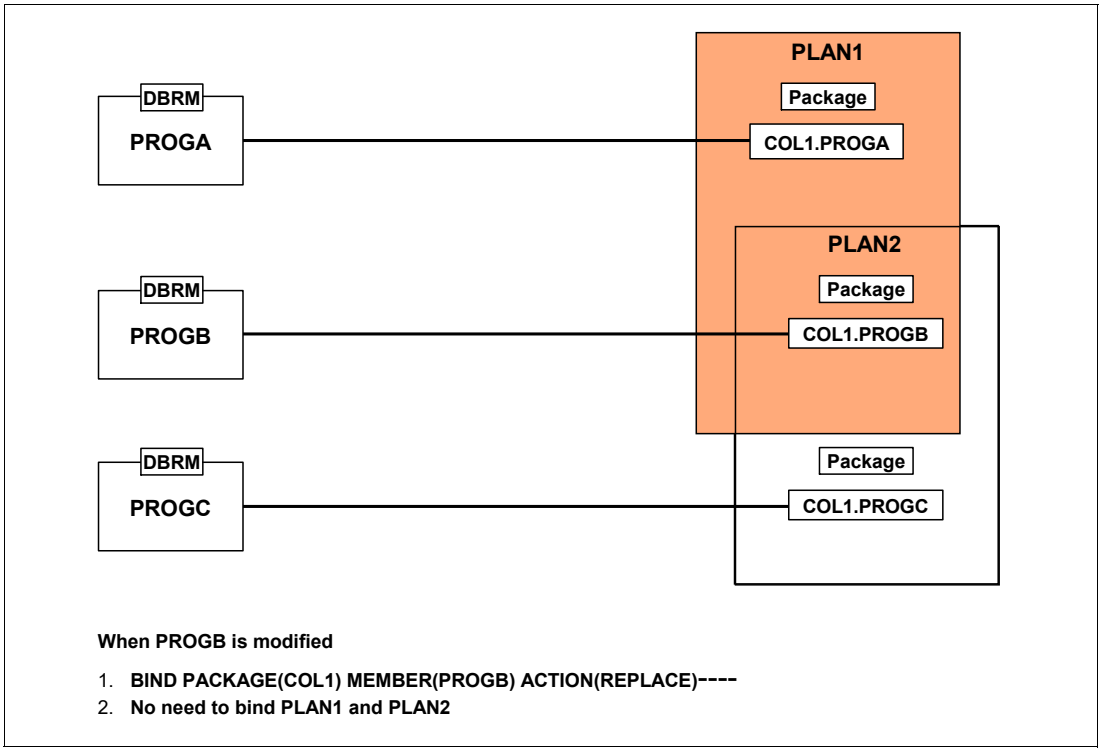

Figure 1-2 Binding packages

Reduced need for BIND PLAN

By using package lists (collection-id.*) in your BIND PLAN statements, you have the ability to add new programs into your plan without rebinding the PLAN. When you bind a package into a specified collection, DB2 looks in that collection for your new package at execution time. This improves availability because there is no outage needed to put a new program into an environment, just some additional runtime cost.

Multiple PLANs can refer to same collection-id.*. You can rebind an existing package, and all the affected plans immediately benefit from one rebind package, thus also reducing the need for a BIND PLAN.

Support for multiple copies of programs

Multiple copies of a program can be supported by using multiple collections. DB2 for z/OS accepts identical package versions if they are in different collections. The collection can be changed for each copy when binding a package.

With the use of the package attribute, VERSION, you can keep also multiple copies of a single package in one collection. In order to do this, each copy of the program must be precompiled with a different VERSION value and put into a separate DBRM library. Each load module must be linked into a separate library. At execution time, DB2 reads the consistency token from the load module and then searches for the matching package. Reverting to earlier versions of the code is much easier and less time consuming, because DB2 keeps the existing versions and the new version of a package in the same collection.

For further discussion about the use of collection and version, see Chapter 5, “Application development framework” on page 69.

Increased application availability

As long as a package is not allocated, it can be rebound, even if the associated plan is still allocated. When changing PROGB, (see Figure 1-2 on page 6), PLAN1 as well as PROGA and PROGC do not need to be bound. By not having to bind PLAN1 and packages PROGA and PROGC, they are still available to users while PROGB is being changed. This increases the availability of your applications while being changed.

Granularity in bind options

You can specify bind options at the package level. This provides the flexibility of allowing each package in a plan to use different options. One frequently executed program can be bound as a package with different parameters than a relatively infrequent one. With a plan, the same options apply.

Access path persistence

Different costs are attributed to each possible access path by DB2, and the optimal one is then chosen. You optimize once and then you package the whole thing. Then you invalidate the code if things change that impact the access strategy.

Persistence frees the application developer from having to tell the database in a consistent manner how to get to the data quickly every time table rows are queried/updated/delete.

Simplified access to mirror tables

Mirror tables are tables (different from the clone tables in this case) that have identical structures and names with different data and qualifiers. One reason to use mirror tables can be to eliminate contention between different areas of a single table. An example would be having stores in both New York and California. We could keep the data in two tables, one for each state. The programs could be directed, at execution time, to one table or the other. This could reduce contention between the different states data as opposed to having one table with both stores. We can also have different maintenance windows for each table.

One way to do this is to have two copies of the code and hard coding different table names. Using packages and mirror tables lets us maintain only one copy of the program. Refer to 5.7, “Using collections to support access to mirror tables” on page 92 for details on the use of mirror tables. This is the preferred method for applications accessing a remote etherogenous DB2 system.

This concept can be extended to a simplified access to different environments. Packages and the ability to bind using a different qualifier allow the code to remain unchanged throughout the development, testing, and implementation phases. For example, you can use BIND PACKAGE with QUALIFIER(MTGDEV), then later, use BIND PACKAGE with QUALIFIER(MTGQA), and finally use QUALIFIER(MTG) in production, where there is a single MTG database.

Ability to support remote access with static SQL

Using packages, remote SQL statements can be executed statically if they are bound at a remote site. Using static SQL can improve your performance. If you use JDBC™ or WebSphere®, you can use Data Studio PureQuery to easily convert applications that use dynamic SQL to static SQL. See 6.9, “pureQuery” on page 129.

Ability to fall back from access path regression

Using the package stability function in DB2 9, provided by the maintenance stream (APAR PK52523, PTF UK31993), you have the ability to REBIND a package and have DB2 switch to use a previously built access path. See 10.13, “Package stability” on page 216.

Ability to use routines (stored procedures, UDFs, and triggers)

Stored procedures, User Defined Functions (UDFs), and triggers all use packages to access DB2 data.

1.4 How to identify a package

A package is identified by five different attributes:

- Location
- Collection
- Name
- Version
- Consistency token

Location

A *location* is an identifier that designates the database management system (DBMS) in which the package is stored.

By default, a blank name implies the local location name. The location name is used in distributed scenarios. A DBMS can be known by different names called location aliases.

Collection

A *collection* is an identifier that denotes a logical grouping of packages. When creating a package, you must always specify the collection. In the example in Figure 1-3, program PROGA has been associated with collection COL1, and program PROGB has been associated with collections COL1 and COL2.

A collection is also a level of authorization control. You can control who can create packages in a given collection, who can execute packages in a given collection, and who can manage the collection. Additional considerations for using collections are discussed in Chapter 5, “Application development framework” on page 69.

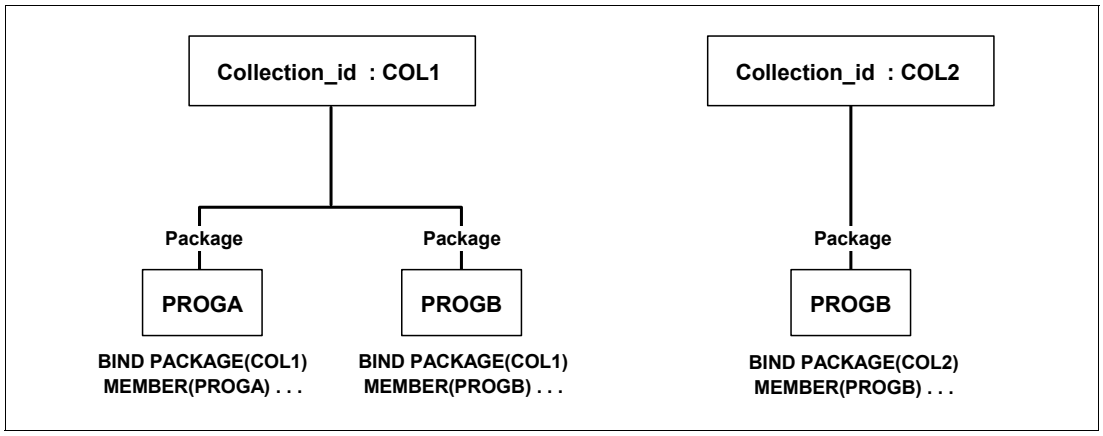

Figure 1-3 Collections and package

NAME

The NAME of the package is automatically generated the same as the DBRM during the BIND. Triggers, native SQL stored procedures, and packages created for distributed applications might not have the same name as the DBRM.

Version

A *version* is one instance of a package created from a DBRM that was generated with the VERSION keyword in either the pre-compile or the compile step of the DB2 coprocessor. One of the advantages of using packages is that you can generate multiple copies of the same package within the same collection. Using package versions in a collection can make backing out a change faster and easier.

Consistency token

Consistency tokens are used to check that the two pieces of code generated for a DB2 program can be used together at execution time. The consistency token is generated during the DB2 pre-compile, or the compile step if using the DB2 coprocessor. It is stored in both the package and the load module. When the load module executes, it uses the consistency token passed from the load module and searches the DB2 catalog for the matching package.

There is a one-to-one relationship for the version_id and consistency token for a package stored in DB2. Keep in mind that it is impossible to have duplicate version_ids or consistency tokens for a package in a single collection.

Figure 1-4 shows how the version and consistency token are stored in the package and the load module.

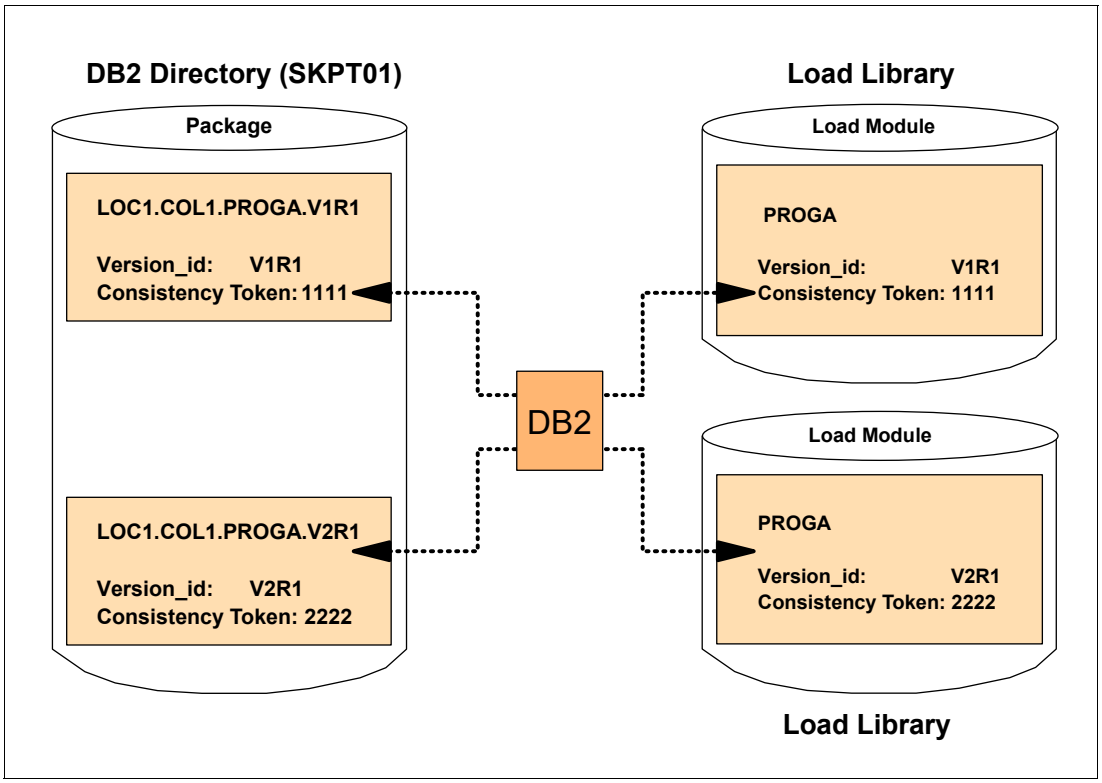

Figure 1-4 Version_id and consistency token

1.5 Program preparation process

This process is followed when coding static embedded SQL, used mainly for COBOL programs running batch or IMS and CICS® transactions. A different process is used for application environments such as JAVA or .NET.

When the source code is ready, the first step is to generate a modified source module and the database request module (DBRM). There are two different methods for creating these data sets. The first method is to use the DB2 precompiler. The second method is to use the DB2 coprocessor.

In the modified source module, the EXEC SQL statements are replaced by a call to the DB2 language interface. The DBRM contains all of the SQL statements that were extracted from the source program. Because one source program is separated into these two data sets, DB2 inserts a consistency token into both data sets. This consistency token is passed to the load module from the modified source module in the compile and link edit (Binder) process.

The BIND process transforms the DBRM into DB2 interpretable form. A consistency token is copied from the DBRM to the package at this point. At execution time, the consistency token from the load module is used to locate the package from the DB2 catalog.

The VERSION keyword is available to both the precompiler and the coprocessor. This keyword supports multiple versions of packages for one program. For a detailed description of the VERSION keyword, see 5.3, “Contrasting collection, version, and consistency CONTOKEN”.

Important: The ability to bind DBRMs directly into plans is planned to be removed in a future release of DB2. In this chapter we assume that you are working with plans that include packages or collections of packages, and not with DBRMs bound directly into plans.

1.5.1 DB2 precompiler

The DB2 precompiler processes the program source code and comments out all of the SQL statements and replaces them with calls to the language interface. It creates two outputs, the first output being the modified source code that is input to the program compiler. The second output is the DBRM that is the input into the BIND process.

Figure 1-5 shows the flow of the precompiler.

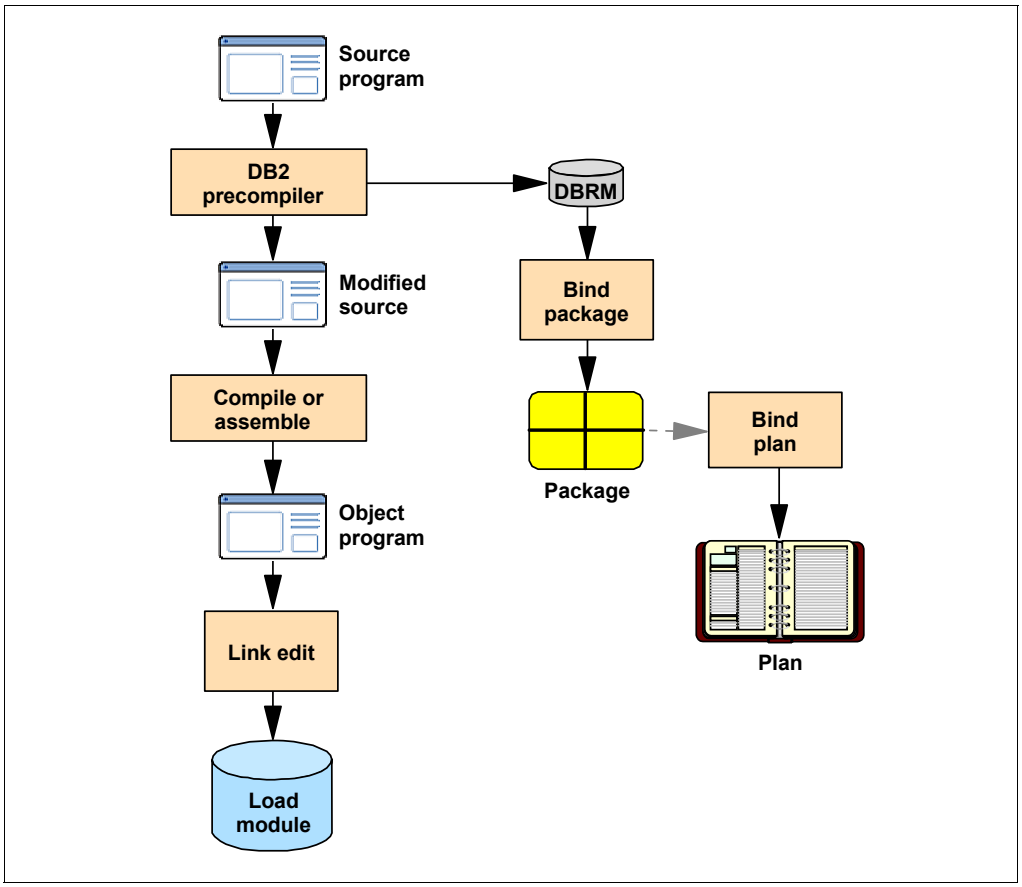

Figure 1-5 Program preparation flow for DB2 precompiler

You process the SQL statements in your source program by using the DB2 precompiler. The output is a load module, possibly one or more packages, and an application plan. Creating a load module involves compiling the modified source code that is produced by the precompiler into an object program, and link-editing the object program to create a load module. Creating a package or an application plan involves binding one or more DBRMs, which are created by the DB2 precompiler, using the BIND PACKAGE or BIND PLAN commands.The BIND PLAN can be done asynchronously with an n:m relationship implemented only via the package list.

You must include the correct interface module (DSNELI, DSNCLI, DSNRLI, DSNALI) during the link-edit step with options that support the precompiler. See the *DB2 Version 9.1 for z/OS Application Programming and SQL Guide*, SC18-9841 for details.

1.5.2 DB2 coprocessor

The DB2 coprocessor performs the functions of the DB2 precompiler and the compiler in one step. It has fewer restrictions on SQL programs than the DB2 precompiler.

Benefits

Here are some of the benefits of using the DB2 coprocessor:

- You enjoy a one-step compilation, including DB2 and CICS statements.

- You can use the Debug tool at the original source level, instead of the precompiled version.

  This enables you to see the EXEC SQL statement while debugging your program, as opposed to seeing the call to a language interface module, usually DSNHLI.
- You can use fully-qualified names for structured host variables.
- For COBOL:
  - You can include SQL statements at any level of a nested program, instead of only in the top-level source file. (Although you can include SQL statements at any level of a nested program, you must compile the entire program as one unit.)
  - REPLACE statements can apply to EXEC SQL statements.
- There is one output listing instead of two (compiler and precompiler).
- For C or C++ programs only: You can write applications with variable-length format.
- For C or C++ programs only: You can use code page-dependent characters, such as left and right brackets, without using tri-graph notation when the programs use different code pages.
- For C or C++ programs, the coprocessor is required if you use the pointer notation of the host variable.
- For COBOL: USAGE NATIONAL is only valid when using the coprocessor.

Usage considerations

These are some considerations when using the COBOL coprocessor:

- Period at the end of an SQL INCLUDE statement:

  A period is required at the end of an SQL INCLUDE statement in the coprocessor, while this is not required with the precompiler.
- Renumbered QUERYNO:

  If you are using OPTHINTs and not using the recommended QUERYNO option in your SQL for statements that need an OPTHINT, your OPTHINT might need to be updated.
- Continuation lines:

  In the precompiler, continuation lines in an SQL statement can begin in columns 8 to 72. In the coprocessor, continuation lines must begin in columns 12 to 72.
- COPY REPLACING with SQL INCLUDE:
  - In the DB2 precompiler, an EXEC SQL INCLUDE statement can reference a copybook that contains a nested COPY... REPLACING statement.
  - In the coprocessor, an EXEC SQL INCLUDE statement cannot reference a copybook that contains a nested COPY... REPLACING statement, because EXEC SQL INCLUDE is processed identically to COPY with the coprocessor, and nested COPY statements cannot use REPLACING.
- FOR BIT DATA host variables:

  A COBOL alphanumeric data item can be used as a host variable to hold DB2 character data having subtype FOR BIT DATA only if:
  - You specify the NOSQLCCSID compiler option, or
  - An explicit EXEC SQL DECLARE VARIABLE statement for the host variable is specified in the COBOL program. For example:

    ```
    EXEC SQL DECLARE :HV1 VARIABLE FOR BIT DATA END-EXEC
    ```

- Multiple definitions of a host variable:

  The precompiler does not require host variable definitions to be unique, while the coprocessor does.

- REPLACE and EXEC SQL statements:

  COBOL REPLACE statements and the REPLACING phrase of COPY statements act on the expanded source created from EXEC SQL statements with the precompiler, while they act on the original source with the coprocessor.

- Source code that follows END-EXEC:

  The precompiler ignores any code that follows the END-EXEC on the same line. The coprocessor processes the code that follows the END-EXEC on the same line. If you have code after the END_EXEC of an SQL statement, the precompiler ignores it, and the coprocessor flags this as an error.

- SQL-INIT-FLAG:

  When using the precompiler In certain instances, you needed to manually reset the SQL_INIT_FLAG using the precompiler. With the coprocessor, this reset does not need to be done. The extra resets do no harm and can be removed as time permits.

Figure 1-6 shows an overall view of the program preparation process using the DB2 coprocessor.

Note: For PL/I, the DB2 coprocessor is called from the PL/I SQL preprocessor instead of the compiler.

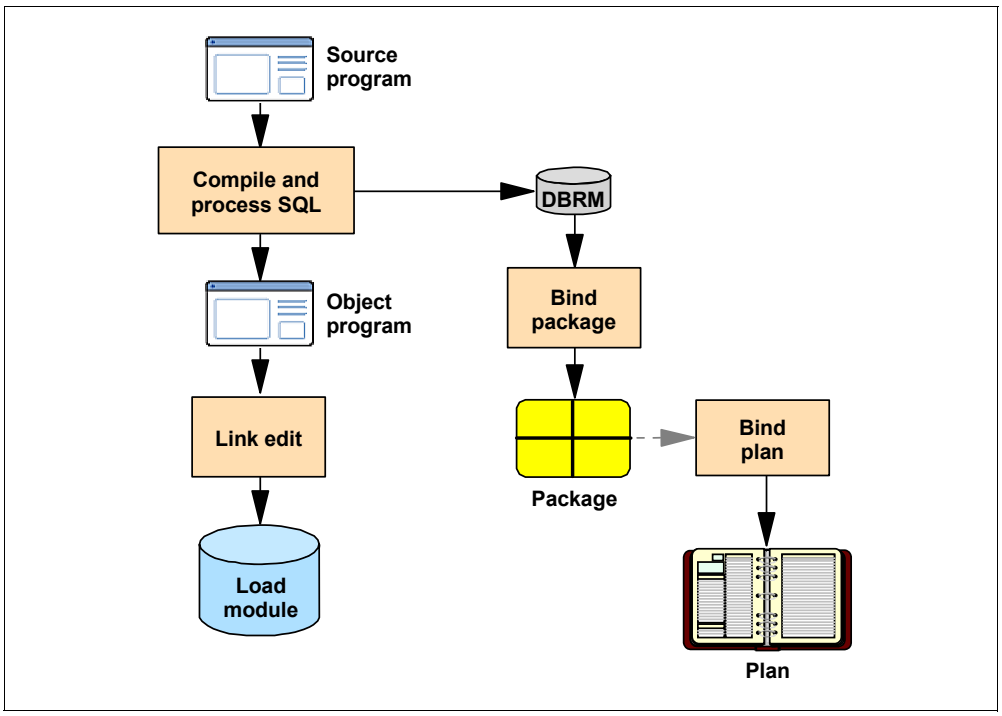

Figure 1-6 Program preparation flow for DB2 coprocessor

If you use the DB2 coprocessor, you process SQL statements as you compile your program. For programs other than C and C++ programs, you must use JCL procedures when you use the DB2 coprocessor. For C and C++ programs, you can use either JCL procedures or UNIX® System Services on z/OS to invoke the DB2 coprocessor.

For detailed information about the differences between the DB2 precompiler and the DB2 coprocessor, see Chapter 11, “Preparing an application to run on DB2 for z/OS - Differences between the DB2 precompiler and the DB2 coprocessor” in *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841, or the Chapter 18. DB2 coprocessor conversion considerations in *Enterprise COBOL for z/OS Compiler and Runtime Migration Guide Version 4 Release 1,* GC23-8527 and *IBM Enterprise PL/I for z/OS Programming Guide,*

1.5.3 SQL processing options

There are many options used by both the DB2 precompiler and the DB2 coprocessor. Some of these options only apply to either the precompiler or the coprocessor.

Next we list some options that might have different uses when using the precompiler or coprocessor.

CCSID(n)

This option specifies the numeric value *n* of the CCSID in which the source program is written. The number *n* must be an EBCDIC CCSID.

The default setting is the EBCDIC system CCSID from DSNHDECP.

SQLCODE / SQLSTATE

If you specify the STDSQL(YES) precompiler or SQL statement coprocessor option, you must declare SQLCODE and SQLSTATE (SQLCOD and SQLSTA in FORTRAN) as stand-alone host variables. These host variables receive the return codes, and you should not include an SQLCA in your program.

ONEPASS/TWOPASS precompiler option

For the DB2 coprocessor, only the ONEPASS precompiler option is allowed for C/C++ and COBOL.TWOPASS can only be used by the coprocessor for PL/I applications. To use the ONEPASS option, you must declare your variables, including the SQLDA, before using them in an SQL statement. The SQLDA must be declared before the first SQL statement when using the coprocessor when using the ONEPASS option.

PREPARE and DESCRIBE statements

In either C or C++, you can use host PREPARE or DESCRIBE statements. You can use host variables of the NUL-terminated form in PREPARE, DESCRIBE, and EXECUTE IMMEDIATE statements.You cannot do this with the precompiler.

REPLACE statement

When using the coprocessor, the REPLACE statement replaces text strings in SQL statements as well as in generated COBOL statements.

LEVEL

This option overrides the construction of the consistency token and puts the checking responsibility in the hands of the programmer.

If you plan on implementing LEVEL, read Chapter 11 in the *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841:

“Although this method is not recommended for general use and the DSNH CLIST or the DB2 Program Preparation panels do not support it, this method enables you to perform the following actions:

- Change the source code (but not the SQL statements) in the DB2 precompiler output of a bound program.
- Compile and link-edit the changed program.
- Run the application without binding a plan or package.”

Note: Use this option with care. Using this parameter makes you responsible for the data integrity checking. If you change the SQL, but not the level, and do not BIND the package, then the results are not predictable.

For a source code change, using BIND would generally be a better choice.

2

BIND options

DB2 for z/OS has several BIND PLAN and BIND PACKAGE options.

In this chapter we discuss these options, grouped as follows:

- BIND options that are usually specified or implied
- BIND options that are important for dynamic SQL
- Additional BIND options for distributed applications
- BIND options that are useful for special situations
- Hierarchy of BIND options and inheritance
- Documenting plans and packages using REMARKS

In this chapter, the most useful BIND options are listed and grouped according to their utility. Refer to the *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844 for a more complete list of options and the full range of values associated with each option.

Important: The ability to bind DBRMs directly into plans is deprecated in DB2 9 for z/OS, and might be removed in a future release of DB2.

In this chapter, we assume that you are working with plans that include packages or collections of packages, and not with DBRMs bound directly into plans. If you still have DBRMs bound directly into plans, refer to Chapter 4, “Conversion to packages” on page 49 for assistance with conversion to package-based plans.

Some BIND PLAN options are relevant only to DBRMs bound directly into plans. We have omitted the discussion of these options from this chapter.

2.1 BIND options that are usually specified or implied

For most bind operations, you specify the BIND options described here, or accept the default value after verification.

2.1.1 ACQUIRE in BIND PLAN

This option on BIND PLAN tells DB2 whether to acquire table space locks as they are needed (USE), or all at once as the plan is allocated (ALLOCATE). It is deprecated in DB2 9 for z/OS.

Packages always use ACQUIRE(USE) and obtain table space locks when they need them, regardless of the specification in BIND PLAN. As you move to packages, this option becomes obsolete.

2.1.2 ACTION (and REPLVER and RETAIN) in BIND PACKAGE and BIND PLAN

The ACTION option tells DB2 whether to add a new package or plan, or replace an existing package or plan. In most cases, allow the default of ACTION(REPLACE) to be used when BINDING both packages and plans.

The REPLVER suboption requests the replacement of a specific version of a package.

Tip: Remember to specify the RETAIN® suboption for BIND PLAN to avoid losing existing EXECUTE privileges.

Replacing a package that contains an explicit VERSION value

One input to BIND PACKAGE is a DBRM with a non-blank VERSION value, another is a package specified using the COPY option and COPYVER suboption. In either case, the source to the bind process contains an explicit VERSION value, and can replace an existing package with the same VERSION value. In this scenario:

- ACTION(REPLACE) is optional because it is the default.
- The REPLVER suboption should be omitted because the correct value of REPLVER is automatically determined from the contents of the DBRM.

Important: If REPLVER is specified, make sure that the version specified in the REPLVER suboption exactly matches the version found in the DBRM. If there is a mismatch, the version specified in REPLVER can be *removed* from the DB2 subsystem and replaced with the version in the DBRM. It is not likely that you will be pleased with this replacement, this is why our advice is to avoid the REPLVER suboption.

See the *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844 for details and an example.

2.1.3 CURRENTDATA in BIND PACKAGE

This option, which is essentially a suboption for ISOLATION(CS), tells DB2:

- How to manage lock avoidance
- Whether blocks of rows can be delivered to a distributed application, as opposed to the transmission of one row at a time

The default of CURRENTDATA(NO) should be used in most circumstances. This allows lock avoidance to take place for read-only and ambiguous cursors, and allows block fetch of data from remote locations.

CURRENTDATA(YES) requires DB2 to acquire additional page or row locks to maintain currency for read-only and ambiguous cursors. In addition, it inhibits block fetch for distributed, ambiguous cursors.

Note that ambiguous cursors are cursors where DB2 cannot determine during bind or prepare processing if the cursor will be read-only or not. The inclusion of a FOR READ ONLY or FOR UPDATE clause definitely resolves the status. A cursor is considered read-only if the result table is read-only. Refer to the “DECLARE CURSOR” statement in *DB2 Version 9.1 for z/OS SQL Reference*, SC18-9854 for details on the range of conditions that force a read-only result table. If there is no UPDATE or DELETE statement using the WHERE CURRENT OF clause, and no FOR READ ONLY or FOR UPDATE clause, and the result table does not need to be read-only, the cursor is ambiguous.

Note that earlier releases of DB2 used CURRENTDATA(YES) as a default, while it is NO for DB2 9 for z/OS.

2.1.4 DEGREE in BIND PACKAGE

This option only applies to static SQL and requests DB2 to consider parallelism when optimizing the SQL for the package. DEGREE(1) prevents parallelism from being considered.

Note that using DEGREE(ANY) does not *guarantee* that parallelism will be used. At bind or prepare time, optimizer calculations need to conclude that the additional cost of executing the query in parallel will result in improved overall performance. At runtime, a combination of DSNZPARM and buffer pool parameters need these supporting values:

- PARAMDEG in the DSNZPARM macro DSN6SPRM sets the maximum degree of parallelism locally.
- DSN6SPRM CDSSRDEF sets the default value for the CURRENT DEGREE special register. The recommendation is 1.
- ASSIST and COORDNTR in the DSNZPARM macro DSN6GRP allows for participation in Sysplex query parallelism.
- These buffer pool thresholds are set via the ALTER BUFFERPOOL command:
  - A non-zero *virtual buffer pool parallel sequential threshold* (VPPSEQT) value for local parallelism.
  - A non-zero *virtual buffer pool assisting parallel sequential threshold* (VPXPSEQT) value for assisting with Sysplex query parallelism.

Even with all of these values in place, DB2 can still determine at runtime that the operating conditions are not suited to parallelism and reduce the degree of parallelism requested during bind or prepare processing.

For more details, refer to “Enabling parallel processing” in *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851.

2.1.5 ISOLATION in BIND PACKAGE

This option determines how a package manages locks in conjunction with other concurrently executing packages. For most scenarios, CS (Cursor Stability) is the normal and default setting.

For read-only SQL that can tolerate reading uncommitted data, a value of UR (Uncommitted Read) can be used. The SQL will avoid locks (with the exception of LOB data), and rely on latches for intra-page integrity during read operations. If a package is bound with UR that contains SQL that performs anything other than read-only operations, those operations execute with ISOLATION(CS).

Additional ISOLATION options have specialized applicability, and can be found in the *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844.

2.1.6 LIBRARY in BIND PACKAGE

This option tells DB2 which library to open to find the DBRM that is being bound into a package.

We recommend that you do *not* use this option. Allocate your DBRMs via the //DBRMLIB DD name in your JCL, or allocate DBRMLIB dynamically if you are using CLIST/REXX processing.

Bypassing the LIBRARY option makes it easier to manage JCL using Procedures, and avoids “hidden” references to environments that can be accidentally ignored when code is promoted.

2.1.7 MEMBER in BIND PACKAGE

This option names the program being bound into a package.

In the BIND PACKAGE command, the program name is referred to as MEMBER.

The program, bound as a package, is known as NAME in the SYSPACKAGE, SYSROUTINES, and SYSTRIGGERS catalog tables.

2.1.8 OWNER in BIND PACKAGE and BIND PLAN

For a BIND PACKAGE command, the authorization ID or role identified by OWNER must have the privileges required to execute the SQL statements contained in the package.

For a BIND PLAN command, without DBRMs bound into a plan, the authorization ID/role of the OWNER must have authority to issue the BIND and authorization to execute packages in the PKLIST.

When OWNER is omitted, the primary authorization ID of the agent running the bind process becomes the owner.

OWNER for BIND and REBIND in trusted context

When BIND and REBIND commands are issued in a trusted context that has the ROLE AS OBJECT OWNER clause, the owner is determined as follows:

- If the OWNER option is not specified, the role associated with the binder becomes the owner.
- If the OWNER option is specified, the role specified in the OWNER option becomes the owner. In a trusted context, the OWNER specified must be a role.

  For the bind to succeed, the binder needs BINDAGENT privilege from the role specified in the OWNER option. The binder also receives BINDAGENT privilege, if the role associated with the binder has BINDAGENT privilege.

If the ROLE AS OBJECT OWNER clause is not in effect for the trusted context, then the current rules for BIND and REBIND ownership apply. If a role is associated in a trusted context, then the role privileges are included in the binder's privilege set to determine if the binder is allowed to perform the bind.

For an existing package (for example, bound with OWNER=TEST1), a change to the OWNER value (for example, binding with OWNER=TEST4) introduces complications and should generally be avoided. When the owner of an existing package is changed, the new owner gains the BIND and EXECUTE privileges, and the authorization of the previous owner is removed and *automatically delegated back by the new owner*.

It can be easier to FREE and BIND a package than to go back and correct and remove the authorizations generated by changing package ownership.

In an environment where VERSION(AUTO) is used, the issue associated with change of ownership can be avoided by simply changing the OWNER value in the JCL for binding new package versions after compilation or migration through development environments. For VERSION(AUTO), the issue persists only where it is necessary to make changes to the existing environment without the roll-out of newly-compiled code.

2.1.9 PKLIST in BIND PLAN

Plans provide an entry point into DB2, and an anchor for performance measurement, but the true execution of SQL takes place via the invocation of packages. PKLIST specifies the list of packages to be made available to a plan.

Note that packages are not *bound into* a plan, they are *made available to* a plan. The same package or set of packages can be used in multiple plans, as scoped by the PKLIST option. Packages available to a plan can be on the local DB2 subsystem or on a remote database server.

See Figure 2-1 for a general example of how packages can be organized into collections, and included within plans. Note that "package" is more complicated than it might first seem. In this diagram, a "package" is assumed to be a single BIND of a package (within a specific collection, and with a specific version). In reality, a program could exist with multiple versions (and, therefore, be bound to as many packages), or could be bound to many collections (perhaps with different OWNER/QUALIFER values, but the same consistency token).

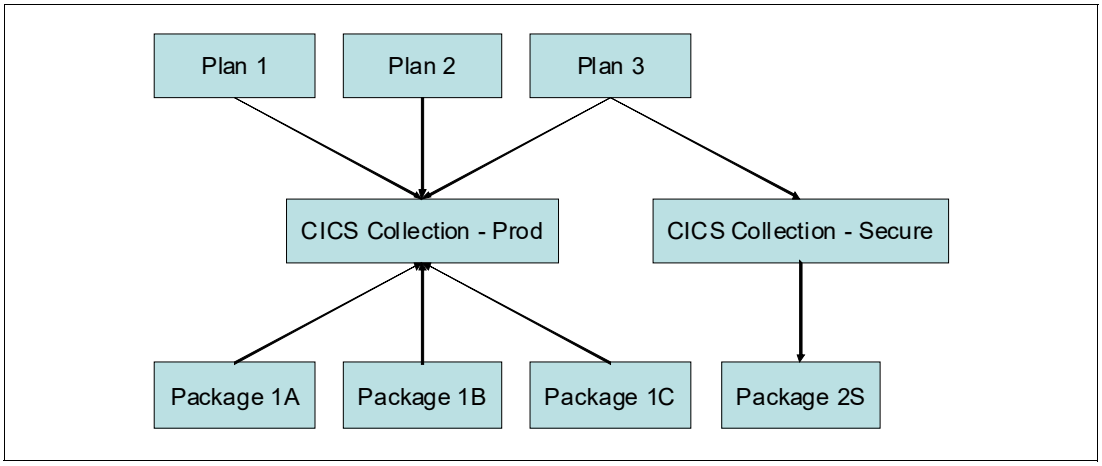

Figure 2-1 An example relationship between Plans, Collections, and Packages

For more information about the use of COLLID and VERSION values to manage packages in development environments, refer to Chapter 5, “Application development framework” on page 69.

Here is an example of a BIND PLAN subcommand that scopes sets of packages, which are available when the plan is executed:

```
BIND PLAN(planname) PKLIST(loc4.coll3.pack2, coll6.*, *.col5.*) ...
```

An asterisk can be used in place of the location name, collection ID, or package name. Note that the location name is optional, and packages are matched against the local subsystem if the location name is not specified. Table 2-1 explains the meaning of the asterisk in each field.

Table 2-1 Using an asterisk in PKLIST

*** use**	**Meaning**
* for package name: PKLIST(col6.*) -- or -- PKLIST(col6.*, col8.*, col2.*)	Any package (program) within the specified collection can be executed by the plan. This is the most common use of a wildcard in PKLIST. Often multiple collections are included in PKLIST for greater flexibility. An additional benefit is that new packages do not require any changes to the existing plan definition.

* use	Meaning
* for location name: PKLIST(*.col5.*)	The package can be run on any location to which the package is bound. Typically, the package name is also set to asterisk, so that any package in the collection can be run at any location to which it is bound. By default a package runs locally, unless one of the following is true: ► A successful SQL CONNECT statement in the program has caused the program to connect to a remote location ► The CURRENTSERVER option on BIND PLAN was used to successfully connect the program to a remote location. The current location name can be determined by interrogating the CURRENT SERVER special register.
* for collection ID: PKLIST(*.pack9)	In this uncommon scenario, the package can be retrieved from any valid collection. The collection ID must be supplied at runtime using by the SET CURRENT PACKAGESET statement. (If this is not executed, an error message is generated at runtime.)

2.1.10 QUALIFIER in BIND PACKAGE and BIND PLAN

This option is used to provide a qualifier for any unqualified names of tables, views, indexes, and aliases contained in the plan or package.

Tables and views (and their companions, synonyms and aliases) use two-part names: a CREATOR and a NAME. A fully-qualified name includes both the CREATOR and the NAME values, but allows a bind to target a single development environment. The use of QUALIFIER in the bind process allows the use of unqualified SQL in programs. As code is staged and migrated, packages can be bound to each of the development environments (with different QUALIFIER values) without the need for change or re-compilation. See Example 2-1 for an illustration.

Example 2-1 Unqualified table/view names in package

```
Sample Table = EMPLOYEE
Sample Environments: DEVT, TEST, PROD

Sample SQL:
SELECT NAME, SSN
FROM EMPLOYEE
WHERE DATE_OF_BIRTH = '01-01-1963'

Sample BIND to the TEST environment:
BIND PACKAGE (FIN_TEST_BATCH) MEMBER(PGM01) OWNER(FINTEST) QUALIFIER(FINTEST)...

All SQL is qualified with 'FINTEST' and the SQL is processed as:
SELECT NAME, SSN
```

```
FROM FINTEST.EMPLOYEE
WHERE DATE_OF_BIRTH = '01-01-1963'
```

Note that where QUALIFIER is not specified on the BIND command, the value of OWNER becomes the qualifier. If OWNER is also omitted, the default value for OWNER is used as the value for QUALIFIER. See 2.1.8, "OWNER in BIND PACKAGE and BIND PLAN" on page 20 for details on how OWNER is determined.

2.1.11 VALIDATE and SQLERROR in BIND PACKAGE

This option tells DB2 whether to validate the SQL in the package during the bind process, or when the package is executed.

For dynamic SQL, the VALIDATE(RUN) option is normally used. Any DB2 objects or privileges that are not resolved at bind time are re-validated during execution. The authorization ID used for validation at runtime is determined by the value of the DYNAMICRULES option, which is described in more detail in 2.2.1, "DYNAMICRULES in BIND PACKAGE" on page 26.

For static SQL, the VALIDATE(BIND) option is normally used. If the DB2 objects and privileges do not exist at bind time, and the SQLERROR option has been allowed to default to NOPACKAGE, the package bind process fails with an error message.

With either VALIDATE value, SQLERROR(CONTINUE) forces successful bind of a package. It is primarily used for distributed applications that are designed to access multiple remote sites. Such programs include syntax that is unique to each environment, and could generate an error when bound at some DB2 sites. With SQLERROR(CONTINUE) the bind process completes successfully. However, the SQL statements that failed in the bind process at any given site generates an error at runtime if execution is attempted at a site where the SQL is invalid. Table 2-2 shows our recommended settings for these values based on the kind of processing and SQL being used.

Table 2-2 Suggested combination of VALIDATE and SQLERROR options

Local or Remote bind processing	Static or Dynamic SQL in package	Suggested VALIDATE option	Suggested SQLERROR option
Local	Static only	BIND	NOPACKAGE
Local	Dynamic, or mix of static and dynamic	RUN	NOPACKAGE
Remote	Static only	BIND	NOPACKAGE if all locations are expected to complete the bind processing because of identical databases. CONTINUE if different database structures exist at each location, and the program is cognizant of location-specific configuration.
Remote	Dynamic, or mix of static and dynamic	RUN	CONTINUE

You should check for incremental binds that are not intended, as they can cause additional CPU costs. Monitor the accounting reports for incremental binds to see if they can be avoided in the following cases:

- The program is written using static SQL and bound with VALIDATE(RUN) and the object does not exist at BIND time
- The program is written using static SQL and bound with REOPT(ALWAYS)
- The program references a declared temporary table.
- The program uses distributed private protocol.

Figure 2-2 provides a graphical representation of the interaction between VALIDATE and SQLERROR during the bind process.

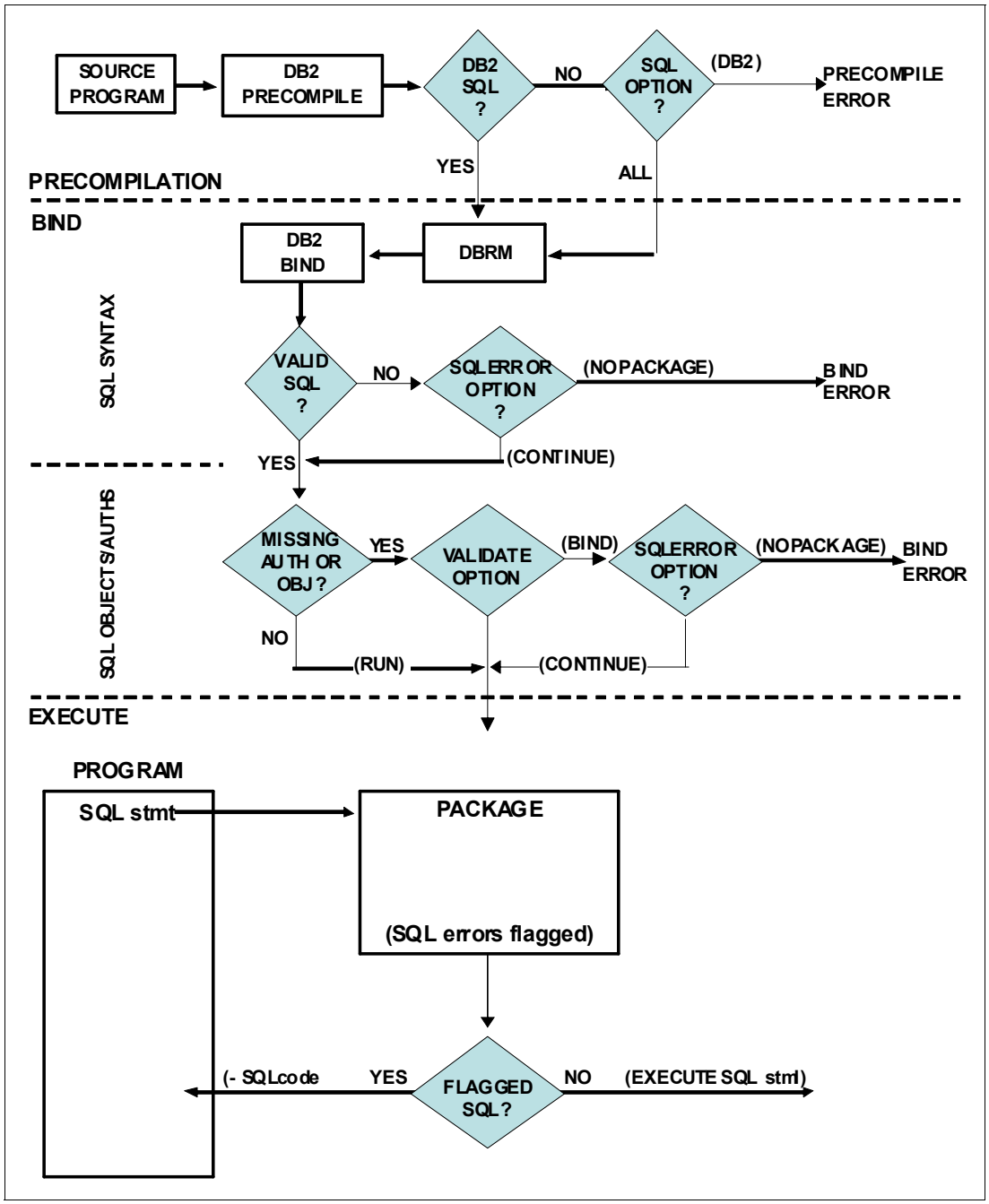

Figure 2-2 Effect of VALIDATE and SQLERROR on BIND PACKAGE

2.2 BIND options that are important for dynamic SQL

Some BIND options apply only to dynamic SQL. In this section we cover the key options.

2.2.1 DYNAMICRULES in BIND PACKAGE

This option applies to the dynamic SQL in a package and:

- Determines the authorization ID that is used to check for authorization
- Determines the qualifier that is used for unqualified objects
- Defines whether *define behavior* or *bind behavior* is used for evaluating the dynamic SQL
- Defines what kinds of SQL statements are allowed to be executed

This is explained in more detail in 9.3.2, “Using DYNAMICRULES(BIND)” on page 181.

2.2.2 KEEPDYNAMIC in BIND PACKAGE

This option allows dynamic SQL statements to be kept past a commit point, thereby avoiding the need for the statement to be prepared again.

The actions taken depend on whether or not the prepared statement cache is active, and whether or not the package is being invoked from a remote location.

Note that REOPT(ALWAYS) is inconsistent with KEEPDYNAMIC(YES), and is not allowed.

The use of statement caching is beyond the scope of this chapter. See 6.7.4, “Dynamic SQL statement caching” for more information.

More details on the DB2 subsystem definition requirements can be found in “Thread management panel: “DSNTIPE and CLIST calculations panel 1: DSNTIPC” in *DB2 Version 9.1 for z/OS Installation Guide,* SC18-9846.

More details on the application programming requirements can be found in “Keeping prepared statements after commit points” in *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9846.

More details on the performance implications can be found in the*DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851.

2.3 Additional BIND options for distributed applications

Some BIND options are especially important for distributed applications, both for dynamic or static SQL. Key options are covered here.

2.3.1 CURRENTSERVER in BIND PLAN

Ordinarily a plan executes local packages unless the program issues a CONNECT statement to cause connection to a remote DB2 for z/OS subsystem, or other DBMS location.

The CURRENTSERVER option causes a connection to the remote location before the program starts to execute. This is executed as a Type 1 CONNECT. Performance is poorer, and options are more limited than for Type 2 CONNECT statements. Where access to the code is possible, it is recommended that explicit CONNECT statements are issued.

See “Program preparation options for remote packages” on *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841-01 for details on CONNECT(1) and CONNECT(2) precompiler and coprocessor options.

Note: Starting with DB2 V8 for z/OS, the DDF install panels (DSNTIPR) have new terminology, they use:

- “Inactive DBAT” instead of “Type 1 Inactive Thread”
- “Inactive Connection” instead of “Type 2 Inactive Thread”

  This is the recommended type. Type 1 is only provided for compatibility.

DBAT is a database access thread.

2.3.2 DBPROTOCOL in BIND PACKAGE

DBPROTOCOL(DRDA) is the only useful option. Use of the DB2 private protocol might be removed in a future release of DB2, and any code written to use DB2 private protocol should be updated to use DRDA. For assistance with the conversion, refer to 6.10, “Private protocol” on page 140.

2.3.3 DEFER and NODEFER in BIND PACKAGE

This option tells DB2 whether or not to prepare dynamic SQL statements immediately, or wait until they are executed. Waiting until they are executed (DEFER) allows the PREPARE and EXECUTE to be sent across the network as a package, and reduces network overhead.

In general, DEFER(PREPARE) is the best option to be used. The default value depends on a number of factors, including:

- The value of REOPT
- The SQL statement being used
- The use of the INTO clause
- Whether it is a local or remote bind
- What value is specified at the plan level

See “Bind package options” on page 107, and the *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844 for more details.

2.3.4 DISCONNECT in BIND PLAN

This option is used to manage when connections to remote locations are released during COMMIT operations, and is discussed in Chapter 6, “Developing distributed applications” on page 101.

DISCONNECT(EXPLICIT) can be combined with the RELEASE statement to retain status information where multiple connections are being maintained. It is discussed in Chapter 6, “Developing distributed applications” on page 101.

2.3.5 OPTIONS in BIND PACKAGE

When a package is bound to a remote server, this option determines what actions to take when the remote server is not a DB2 for z/OS server, or is at a different version level to the current DB2 subsystem.

It is typically used when packages are expected to be available on all DB2 subsystems, and the initial roll-out starts at the newest version. The BIND...COPY process can require OPTIONS(COMMAND) to achieve a successful bind at the down-level DB2 subsystems.

Refer to the *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844 for details.

2.4 BIND options that are useful for special situations

Additional BIND options are covered because there are special situations where it can be helpful to know how to use them.

2.4.1 COPY and COPYVER in BIND PACKAGE

This option allows you to copy an existing package. The process does not read a DBRM, but uses catalog package information as the source for the copy. In other respects, it goes through the full bind process. COPYVER defaults to blank, so it must be specified if the source package has a non-blank value for VERSION.

Because the copy function is still a complete bind process, the access path is optimized for the copied package at bind time (in accordance with relevant bind options, like VALIDATE).

A package can be copied to a local or remote system, but the original package must reside in the local DB2. To put it another way, a BIND...COPY operation copies *from* a local package either to another collection on the same DB2 subsystem, or to a collection on a remote server.

If you want to copy a package within the same DB2 subsystem, the new package must be bound to a different collection because both copies of the package have identical location names, package names, version names, and consistency tokens.

If an application package is being copied to a remote location, or to a different collection in the same DB2 subsystem, it is recommended that you use EXPLAIN(YES) to be able to compare the access paths of the original and new package.

If system software is being copied to remote DB2 subsystems, it might not be necessary to examine access paths as rigorously as for application code.

Tip: A copied package generally has the same bind options as the original package, unless bind options are overridden. However, the following parameters are not inherited from the original package, and you must specify them if the system default is undesirable:

- OWNER
- QUALIFIER
- ENABLE / DISABLE

Eliminating your DRBM library

Depending on the details of your source management strategy, the COPY option can be used to eliminate the use of DBRM libraries, provided that the precompile/coprocessor step and BIND PACKAGE process occur in the same job stream.

If a temporary data set is used for the DBRM and passed to the BIND PACKAGE process, the COPY option can be used to migrate the package to other development environments without further need for the DBRM.

Note that we do not generally recommend this DBRM library elimination, but it can be used in some environments.

2.4.2 ENABLE and DISABLE in BIND PACKAGE and BIND PLAN

In theory, a package created for a secure CICS transaction could be run from another environment, such as a batch job. To mitigate this risk, the execution of packages and plans can be restricted to specific system connection types.

The ENABLE option allows a package or plan to be executed *only* using the specified connection type and DISABLE prohibits a package or plan from being executed anywhere except under the specified system connection type. See Example 2-2 for different approaches to the use of ENABLE and DISABLE.

Example 2-2 Using ENABLE and DISABLE

```
BIND PLAN(plana) PKLIST(col4.*) ENABLE(CICS) ...

BIND PACKAGE (secure-col5) MEMBER(prog7) ENABLE(CICS) CICS(prodcics) ...

BIND PLAN(plana) PKLIST(col4.*) DISABLE(BATCH,DB2CALL,RRSAF,REMOTE) ...
```

Note that the ENABLE and DISABLE keywords are valid only for *local* bind operations.

A useful scenario for DISABLE is to provide a staging area before an obsolete package is removed from the subsystem. By disabling access to all environments, a package can no longer be executed, but its information is retained in the DB2 Catalog and Directory. The process to manage obsolete packages is described in detail in 8.3, “Deprecating packages” on page 165.

The ENABLE and DISABLE keywords operate at both the plan *and* package level. Figure 2-3 shows how these options interact.

		PLAN	
		Enabled	**Disabled**
PACKAGE	**Enabled**	Application is executable in the specified system connection type.	Application is NOT executable in the specified system connection type.
	Disabled	Application is NOT executable in the specified system connection Type.	Application is NOT executable in the specified system connection type.

Figure 2-3 Relationship between ENABLE and DISABLE

2.4.3 DEPLOY in BIND PACKAGE

This option is used only to copy a native SQL procedure. It is similar in functionality and syntax to the COPY option, except that it applies specifically to native SQL stored procedures. The target of the command is a package and it also copies the stored procedure.

When using the DEPLOY option, only the collection ID, QUALIFIER, ACTION, and OWNER options are allowed. (OWNER and QUALIFER defaults to their existing values if not specified). This allows a functionally-equivalent procedure to be copied to another development environment in a local or remote DB2 subsystem, but does not allow other changes.

Note that a native SQL procedure, including the bind of the package that supports it, is initially defined to DB2 using the CREATE PROCEDURE statement. Like other procedures and programs, a native SQL procedure can have multiple versions.

Refer to *DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7604, for more details on implementation and management of stored procedures.

2.4.4 ENCODING in BIND PACKAGE

CCSID conversions can be required for environments where different languages are used, and where internationalization of the application is required to allow an application to run unchanged in different countries with different languages.

The ENCODING overrides the default CCSID found in DSNHDECP and applies to host variables used with static SQL statements.

2.4.5 IMMEDWRITE in BIND PACKAGE

In a situation where two independent database transactions (units of work) form a complete business process, the set of transactions is referred to as a *global transaction*.

The IMMEDWRITE parameter, and a subsystem parameter of the same name, work together in a data sharing environment to allow the second transaction (the *dependent transaction*) to access data updated by the first transaction.

For details, refer to the *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844, and to “Programming your applications for concurrency” in the *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851.

2.4.6 OPTHINT in BIND PACKAGE

When optimization hints are enabled for the DB2 subsystem via the OPTHINTS parameter in DSNZPARM, the OPTHINT bind option requests the bind process to use predefined hints for access path selection. For details, refer to 10.9, “Optimization hints” on page 209.

2.4.7 PATH in BIND PACKAGE and PATHDEFAULT on REBIND PACKAGE

The PATH and PATHDEFAULT options determine how DB2 resolves unqualified names for user-defined distinct types, functions, and stored procedure names (in CALL statements). This is different and distinct from QUALIFIER, which covers tables, views, indexes, and aliases.

Simply stated:

- QUALIFIER deals with database objects
- PATH deals with executable code

Unlike schema names in SQL statements, the schema names included a PATH are not folded to upper-case, so mixed-case or lower-case schema names do not find a match during the bind process.

Specifying PATHDEFAULT on a REBIND resets the PATH value to the default IBM path:

- “SYSIBM”
- “SYSFUN”
- “SYSPROC”

If neither PATH nor PATHDEFAULT are specified on a REBIND, the existing PATH definition is used.

For BIND, the IBM defaults are (in sequence): “SYSIBM,” “SYSFUN,” and “SYSPROC.” When PATH is specified, any schema you specify is *appended* to this sequence.

An example of how this can be used is where an organization wants to provide corporate procedures for general use, departmental procedures that might replace a few of the corporate procedures, and application-level procedures that might replace either the corporate or departmental procedures. Conceptually, this is similar to STEPLIB processing and is illustrated in Example 2-3.

Example 2-3 Hierarchy of schemas in a PATH

```
Default IBM hierarchy:
   SYSIBM, SYSFUN, SYSPROC

Application-level procedures schema: APPL_STUFF
Departmental procedures schema: RESEARCH_PROCS
Corporate procedures schema: STUFFY_INC_PROCS

You specify:
   PATH(APPL_STUFF, RESEARCH_PROCS, STUFFY_INC_PROCS)

Actual PATH used by DB2, including the IBM defaults:
   SYSIBM, SYSFUN, SYSPROC, APPL_STUFF, RESEARCH_PROCS, STUFFY_INC_PROCS
```

2.4.8 PLANMGMT and SWITCH in REBIND PACKAGE

The PLANMGMT option provides support for *package stability* and requests that DB2 maintain old copies of the details for packages. This includes metadata, query text, dependencies, access paths, and so on.

New options for REBIND PACKAGE and REBIND TRIGGER PACKAGE allow a user to preserve multiple package copies and switch back to a previous copy of the bound package if needed.

If REBIND causes undesirable changes to an access path, a REBIND with the SWITCH option can be used to revert to the original access path.

For information about usage, refer to 10.13, “Package stability” on page 216, and also the *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851.

2.4.9 RELEASE in BIND PACKAGE

This option tells DB2 whether to release resources at each COMMIT, or when the plan is deallocated.

“Resources” are generally considered to be locks, but in fact some transient DB2 structures used to improve performance are also released (destroyed).

RELEASE(COMMIT) releases resources at each COMMIT. It also causes structures to be destroyed. This switches off dynamic prefetch, and eliminates Insert procedures (IPROCs), Update procedures (UPROCs) and Select procedures (SPROCs). If processing continues, DB2:

- Reacquires database object locks as they are needed
- Uses sequential detection to restart dynamic prefetch if it is needed
- Rebuilds IPROCs, UPROCs, and/or SPROCs if they are needed

For this reason, issuing COMMIT too frequently causes overhead beyond additional CPU for the COMMIT processing itself. It is recommended that online applications generally use RELEASE(COMMIT) because there is usually only a single COMMIT.

For batch applications that process large data volumes, RELEASE(DEALLOCATE) can be appropriate, although the retained table space locks could prevent a online utility from completing successfully.

2.4.10 REOPT in BIND PACKAGE

This option applies to both static and dynamic SQL, and tells DB2 whether or not to reconsider an access path at runtime. The values range from REOPT(NONE) where DB2 does not reconsider an access path at runtime, to REOPT(ALWAYS) where DB2 always reconsiders the access path.

See 10.8, “REOPT (NONE / ALWAYS / ONCE / AUTO)” on page 206 for details and the differences in impact to static and dynamic SQL statements.

2.4.11 ROUNDING in BIND PACKAGE

The ROUNDING option takes the DSNHDECP default unless it is overridden, and the DSNHDECP default is normally sufficient.

The conventional approach to rounding is that .5 or higher is rounded up, and anything else is rounded down. This is represented by the HALFUP option in DSNHDECP or bind processing. Your friendly DB2 SYSADM can set this default to be used in your DB2 subsystems.

2.5 Hierarchy of BIND options and inheritance

When plans and packages are bound, the bind options can conflict. Most of the time this can occur because many BIND PLAN options apply only to DBRMs bound directly into the plan. When this is not the case, the hierarchy is defined to allow statement-level processing to override package-level options, which override plan-level options:

- Statement-level option
- Package-level option
- Plan-level option

In fact, it is rare for plan options to be inherited by packages. Part of the reason for this is that a given package can appear in many collections and, therefore, many plans. When the same package appears in many plans, which of the plan options would apply to the package? The answer is that where this situation does occur, for example with the ISOLATION option, the package inherits the plan option at *runtime*. This makes the resolution unambiguous, and not dependent on the sequence in which binds were executed.

For some key BIND options, details on the default value, and whether inheritance applies can be found in Table 2-3. We have omitted the BIND options where the plan and package values are independent, and the option is only relevant to BIND PLAN when DBRMs are bound directly into a plan.

Table 2-3 Hierarchy and inheritance of BIND options

BIND Option	Relevance to BIND PLAN	Package default, and inheritance of plan value
ENABLE / DISABLE	► Relevant to BIND PLAN when DBRMs are bound into the plan ► Interacts with package-level ENABLE / DISABLE values: if specified at plan-level and package-level, *both* need to be set to ENABLE for a package to execute	► No inheritance of plan value, but both are checked at execution time before execution is allowed (see across)
ENCODING	► Only relevant to BIND PLAN when DBRMs are bound into the plan.	► Does not default to a specific value. ► The encoding scheme for the plan is *not* inherited by the package. ► When a package is run remotely, the ENCODING option is ignored, and the remote server's encoding scheme is used.
ISOLATION		► Does not default to a specific value. ► When omitted from BIND PACKAGE, it defaults *at runtime* to the plan value for a local server, and CS for a remote server.
OWNER	► Still relevant to BIND PLAN even when no DBRMs are bound into the plan because it is used to determine access to execute the associated packages.	► No inheritance of plan value
RELEASE	► A value does not need to be specified for the RELEASE option. ► For the plan, if omitted, this option defaults to RELEASE(COMMIT)	► For the package, if omitted, this option defaults to the plan value *at runtime*.

BIND Option	Relevance to BIND PLAN	Package default, and inheritance of plan value
ROUNDING	► Only relevant to BIND PLAN when DBRMs are bound into the plan.	► Takes the default from DSNHDECP when omitted.

2.6 Documenting plans and packages using REMARKS

The SQL COMMENT statement allows documentation of a plan or package. REMARKS cannot be created for a trigger package or a package that is associated with an SQL routine.

Up to 550 bytes can be entered for plans or packages.

REBIND PLAN or REBIND PACKAGE does not remove documentation.

BIND PACKAGE... COPY does not copy REMARKS. BIND PACKAGE... ACTION(REPLACE) removes REMARKS. This provides a maintenance challenge for package documentation, so be aware that that configuring REMARKS for packages can be time-consuming, error-prone, and susceptible to loss, depending on the precision of the source management process.

BIND PLAN is often executed infrequently, and many sites use a relatively small number of plans that are persistent and stable. We recommend, therefore, that setting up REMARKS for plans can provide helpful metadata for developers, performance analysts, and others.

3

Execution flow of packages

Packages must exist in collections. During execution, DB2 always requires a plan that has established a connection to DB2. Except for packages such as triggers, stored procedures, and distributed applications, packages must belong to a plan. The packages that can be executed under a specific plan are defined in the PKLIST option, and before using packages, it is essential that you clearly understand the way in which this list is used by DB2 in searching for a package.

In this chapter we explain the processing of packages in the EDM pool and the search algorithm that is used to locate the packages at execution time.

We cover the following topics:

- EDM pool processing
- Package list search

Other exception situations, such as those for Java, JDBC, ODBC, and CURRENT PACKAGE PATH, are discussed in Chapter 5, “Application development framework” on page 69.

3.1 EDM pool processing

In this section we discuss the plan header entries related to packages and the processing of packages in the EDM pool. The differences in EDM pool processing between DBRMs and packages are highlighted, particularly with respect to skeleton cursors.

As you migrate away from plans containing DBRMs to plans containing packages only, portions of this section become not applicable.

3.1.1 Overview of plan structure

Let us quickly review the plan structure. Figure 3-1 gives an overview of the possible components of a plan showing DBRMs and packages (MEMBER and PKLIST entries).

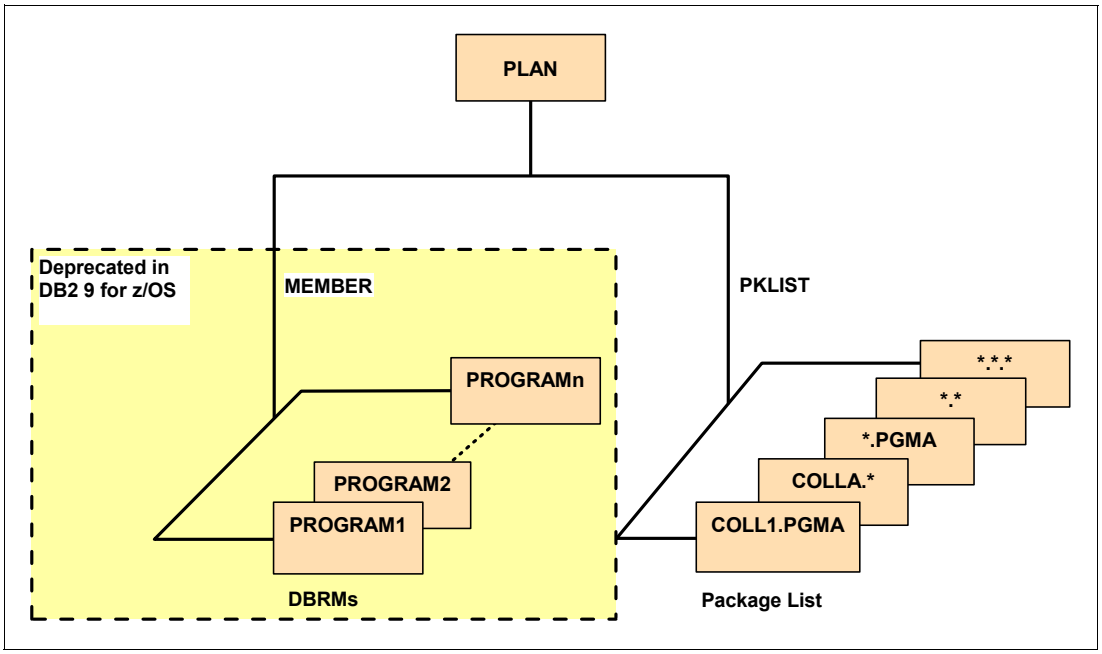

Figure 3-1 Overview of plan structure

We now discuss each of these sub-components of the plan.

DBRMs

DBRMs directly bound into plans are still supported but their use has been deprecated in DB2 9. At runtime, if the CURRENT PACKAGESET register is set by the application program, DB2 skips the DBRM checking. However, if the CURRENT PACKAGESET register is NOT set, DB2 checks the plan header first to see whether DBRMs are present. If they are, DB2 scans the member list looking for a DBRM with the same name and consistency token as those passed from the program.

Note that the search need only use the DBRM name and consistency token because DBRMs can only be used locally and collections do not apply. If a match is found, the DBRM is allocated and the SQL request is processed. If there is no DBRM list in the plan header, or if no match is found in the scan of the DBRM list, DB2 begins a search for a package to satisfy the SQL request.

Collection ID/PKLIST entries

The following entry types can be included in the plan definition using the PKLIST parameter and are commonly used:

Entry Types	Description
COLL1.PGMA	Specific package (in a location or in all locations)
or	
***.COLL1.PGMA**	Specific packages, identified by collection ID and package name (DBRM name), can be included in the PKLIST parameter in the BIND PLAN command. Adding new specific packages requires a rebind of the plan.
COLL1.*	**Collections**
or	
.COLL1.	This specification allows all packages bound into the specified collection (local or remote) to be executed under the plan and eliminates the need for a BIND PLAN when new packages are added to the same collection.

The following entry types can also be included in the plan definition using the PKLIST parameter, but will generally not be used:

***.PGMA**	This entry indicates that all collections that contain a package with a name of PGMA are valid for this plan. DB2 returns a -812 SQL code if this entry is used without a value being set in the CURRENT PACKAGESET register (which can be overridden by the CURRENT PACKAGE PATH register). Privileges are checked again at runtime if an asterisk is used for the collection name.
.	This entry indicates that all packages in all collections are valid for this plan at this location. Entries such as these also require that the CURRENT PACKAGESET register be set in the program (which can be overridden by the CURRENT PACKAGE PATH register).
.*.	This entry indicates that all packages in all collections for all locations are valid for this plan. This entry requires that the CURRENT PACKAGESET register be set in the program (which can be overridden by the CURRENT PACKAGE PATH register). The location name defaults to the local location if the CURRENT SERVER register is not set through either a CONNECT statement in the program or the use of the CURRENTSERVER parameter at package bind time.

Plan header contents

When a plan contains DBRMs and packages, the contents of the plan header are as shown in Figure 3-2. The contents in the header related to DBRMs are unchanged, with the package list specified at BIND PLAN time added at the end. Some key things to remember are:

- The DBRM directory contains a sorted list of the DBRMs bound into the plan.
- The PKLIST entries are built from the package list content, they are stored in the sequence in which they were specified at the time the plan was bound.

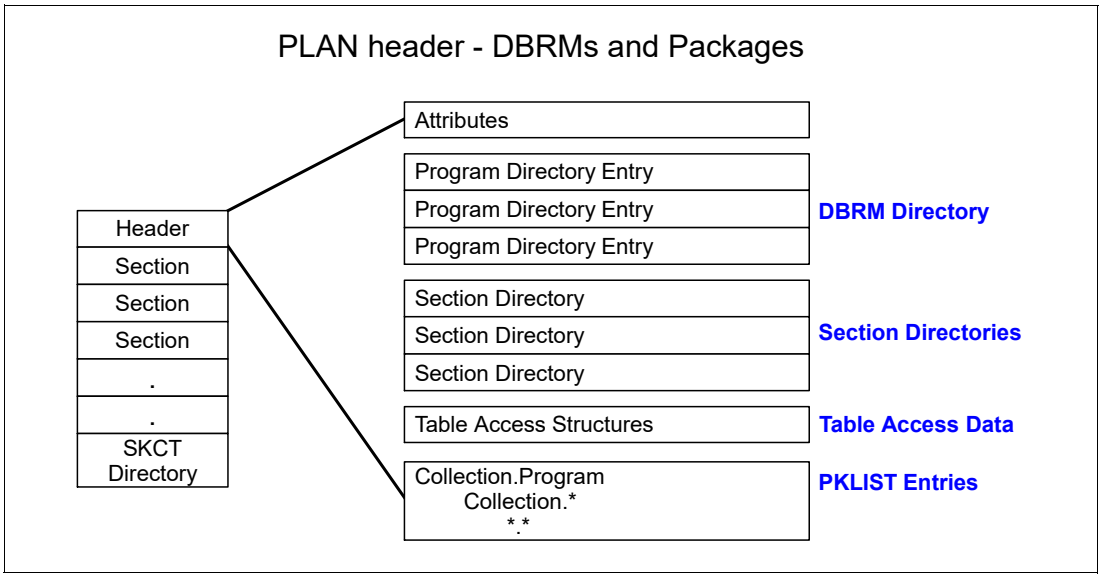

Figure 3-2 Plan header: When both DBRM and packages are present

When there are no DBRMs associated with a plan, section information is not stored in the header. The format of the plan header when there are no DBRMs associated with the plan is shown in Figure 3-3.

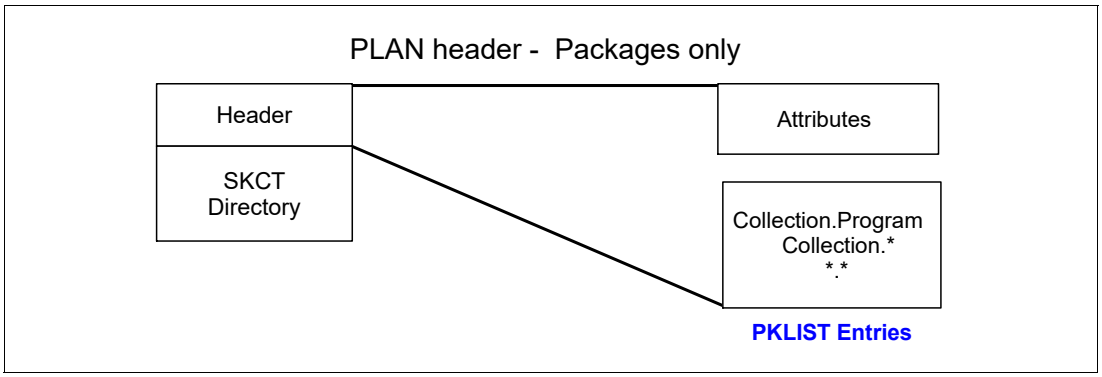

Figure 3-3 Plan header when packages only are present

3.1.2 Package execution flow

In this section, we describe the steps involved in the execution of a package. Figure 3-4 gives an overview of the processing of packages in the EDM pool. We assume that there are no DBRMs associated with the plan and that it contains packages only.

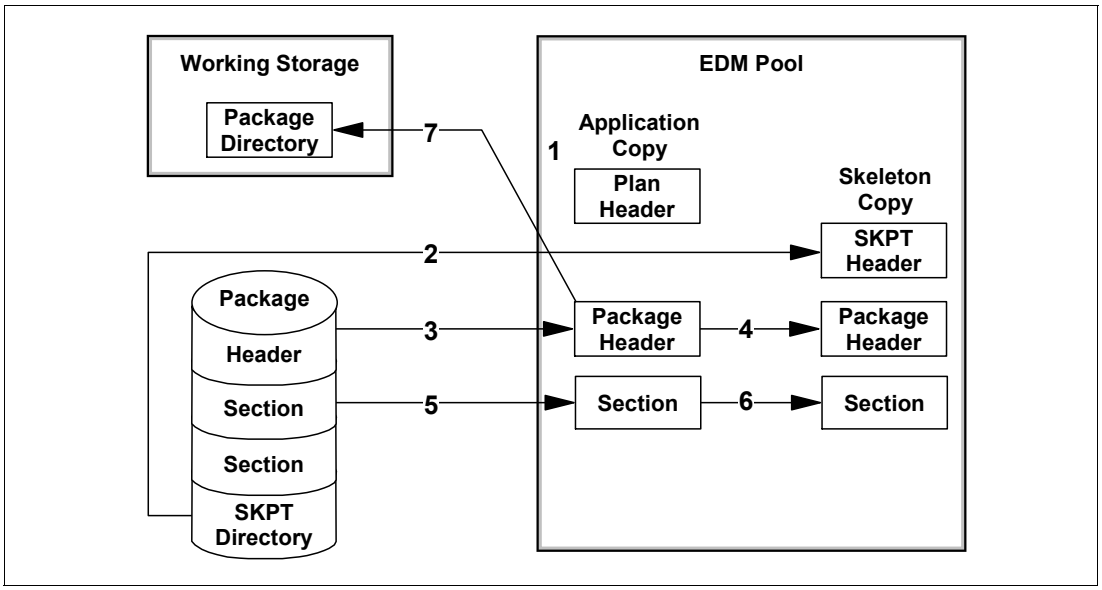

Figure 3-4 Package execution flow overview

The steps involved in the execution of a package are as follows:

1. The plan header is loaded into the EDM pool at plan allocation and forms part of the control block structure that supports the application. The plan header contains the package list that is used to locate the packages involved in the program execution.
2. The program issues an SQL request, and the required package is located. The search process used to locate this package is discussed later in 3.2, “Package list search ” on page 41. The SKPT header for this package is loaded from the Skeleton Package Table space SPT01) and forms part of the skeleton copy.
3. The package header is read from the SPT01.
4. A copy of the package header is made for the skeleton copy.
5. The relevant section is read from the SPT01.
6. A copy of the section is made for the skeleton copy.
7. A package directory is created for each agent in working storage at the time of the first package allocated to the plan in the EDM pool. A DBRM can be bound only once in a plan. However, the same package name, through the use of collections and versioning, can occur many times in a plan. The purpose of this directory is to keep track of all packages that have been allocated and not released since the plan was first allocated. The directory is used during package search to determine whether there is a package in the EDM pool that meets the search criteria.

The package directory is updated during package allocation with the full package name; that is, location name, collection ID, package name, and consistency token.

Deletion of the entries from the package directory is governed by the RELEASE option specified at bind time. If RELEASE(COMMIT) was specified at bind time, the entry in the package directory for the package is deleted at COMMIT time. Note that the skeleton copy of the package remains in the EDM pool and can still be used to create application copies until it is deleted to make space in the EDM pool. If RELEASE(DEALLOCATE) was specified at bind time, the entry for the package remains in the directory until the plan is deallocated.

3.1.3 Package REBIND and the EDM pool

An Internal Resource Lock Manager (IRLM) lock is taken on a package when it is allocated to a plan in the EDM pool. Any attempt to rebind a package (for example, to change the access path being used) while it remains allocated results in a resource unavailable message.

Note: This restriction applies to a specific version of a package. That is, while a version is allocated to a plan, another version of a package could be rebound with no impact on availability. See Chapter 5., “Application development framework” on page 69 for details on versioning.

If the package was bound with RELEASE(COMMIT), the package cursor is released at the next commit, and the rebind can be achieved. If frequent commits are issued by the program, the rebind would occur before the resource unavailable message is received. If there is a skeleton copy of the package in the EDM pool at the time of the rebind, it is marked invalid, and the next request for this package requires a read from SPT01 in the DB2 Directory.

If the package has been bound with RELEASE(DEALLOCATE) and is allocated to a plan on a long-running thread, the process must wait and some special action might be necessary. This could include stopping one or more threads to enable the plan to be deallocated. When the package is bound or rebound, the skeleton is marked invalid, and a request for it requires a read from the DB2 directory as before.

3.1.4 Reuse of packages in the EDM pool

When another application using a different plan (or a different thread using the same plan) requires the same package, and the skeleton copy is still in the EDM pool, a copy of the SKPT is made to satisfy the request. Figure 3-5 shows the processing involved in this reuse.

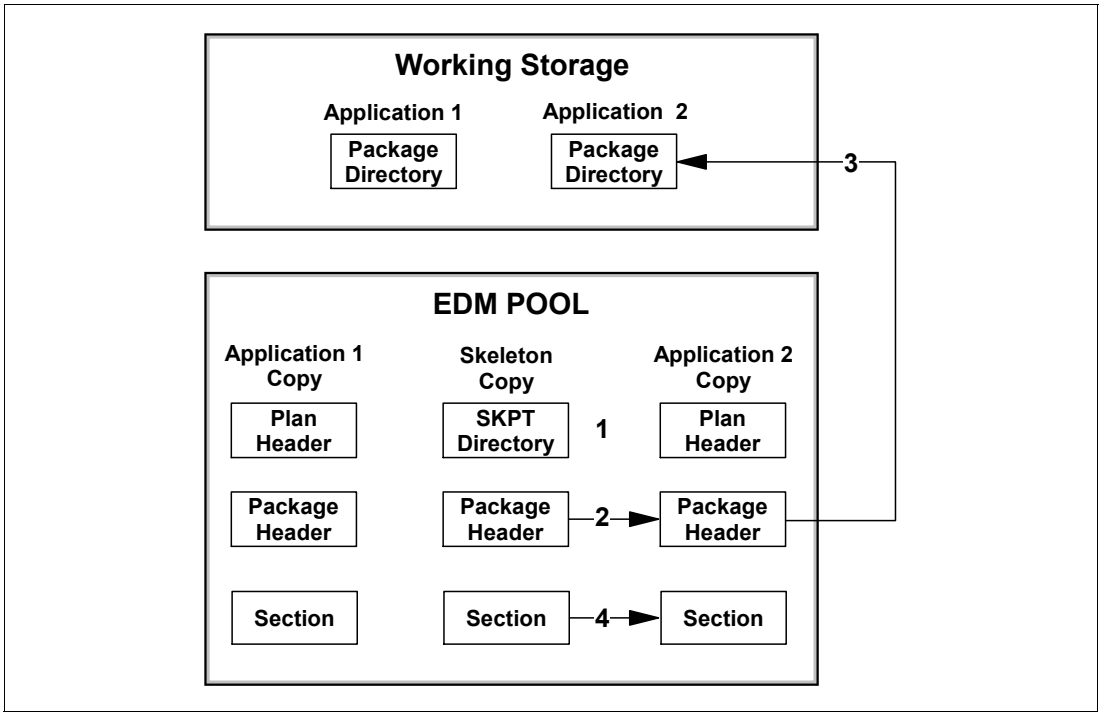

Figure 3-5 Package replication in the EDM pool

Figure 3-5 shows the processing that occurs in the EDM pool when a package that Application 1 is using has been identified during package search processing as one that Application 2 also requires. Before this request, the EDM pool would have contained the control block structure that Application 1 is using and the skeleton copy for the package. The application copy consists of the Plan 1 header and the header and a section of the package being used. There would also be an entry for this package in the package directory for Application 1 in working storage.

Application 2 is now started. The major activities shown are as follows:

1. The plan header is loaded from the skeleton cursor table space (SCT02) space and a package directory for this plan is allocated in working storage.
2. The SQL statement passed from the program requires the package that is already in the EDM pool. The assumption here is that the package search process has established the identity of the package required and passed the request to the Data Manager. The Data Manager always checks the packages already allocated in the EDM pool before initiating a read from SPT01. Because the required package is already in the pool, a copy is made. Note that this is not detected by reference to the Application 1 package directory, which is only used to check whether a package required for Application 1 is already in the EDM pool. The package header is copied from the skeleton copy.
3. An entry for this package is added to the Application 2 package directory.
4. Application 2 requires the same section currently in the EDM pool, and that section is copied as well.

A similar situation involving DBRMs would require two skeleton copies of the DBRM, one for each of the applications executing concurrently.

3.2 Package list search

While a plan can include DBRMs, this feature has been deprecated in DB2.

The flexibility comes from the package list specified at bind time. Packages executed at runtime need not have existed at the time the plan was bound. In addition, while a DBRM can exist only once in a plan, the same package name can exist many times in a plan either as versions within a collection or in different collections. Because of this flexibility, it is important that the run-time search algorithm used to locate a package be understood.

Package list entries can include specific packages, where location name, collection ID, and package name are specified, or specific collections, where location name and collection ID are specified. Or, the specification of the collection can be deferred until runtime using an asterisk for either the location or collection qualifiers, or both.

3.2.1 Consistency token

Let us quickly review the concept of the consistency token, as DB2 uses the token in conjunction with the location name, collection ID, and package name when searching the package list in the plan to satisfy a program request.

Two unique indexes are used to enforce package naming constraints on the SYSIBM.SYSPACKAGE table. One index is based on location name, collection ID, package name, and version_id. This index associates each version of a package within a collection to a unique version identifier.

The other index is based on location name, collection ID, package name, and consistency token. This index associates each version of a package within a collection to a unique consistency token.

The important point to note is that there is always a one-to-one relationship between the version_id and the consistency token within one collection. However, it is the consistency token and not the version_id that DB2 uses when searching for the correct version of the package to be executed.

3.2.2 Plan processing: The “big picture”

Note: In this section, we discuss the program search and allocation process which applies to DBRM-based plans as well as for plans containing packages (or both).

If the plan contains multiple package lists, the PKLIST needs to be searched as well. This is discussed in the 3.2.3, “Package list search sequence” on page 45.

When an SQL statement is issued in a program, three parameters important for the search are passed to DB2 to identify the appropriate DBRM or package to be used. These are:

- The DBRM name
- The section number
- The consistency token

When you are not using packages, a DBRM can occur only once in a plan, so that the DBRM name alone uniquely identifies the DBRM to be used. The consistency token is used to check that the load module being used and the DBRM in the plan were both generated in the same run of the precompiler.

When using packages, the DBRM name alone does not uniquely identify a package, however. The same DBRM can be bound many times into packages, for example, in a number of different collections. These collections could then be bound into the same plan, which would then contain many packages with the same DBRM name and consistency token.

Which one of these packages is chosen at runtime depends on four factors:

- The value in the CURRENT SERVER special register.
- The value in the CURRENT PACKAGESET special register.
- The value in the CURRENT PACKAGE PATH special register. This value overrides the value in the CURRENT PACKAGESET when determining the collection to search.
- The sequence of entries in the package list.

Figure 3-6 gives an overview of the plan-level processing and allocation process.

Note: In order to simplify the processing flow, we have not shown the dependency on CURRENT PACKAGE PATH, which overrides the value of CURRENT PACKAGESET.

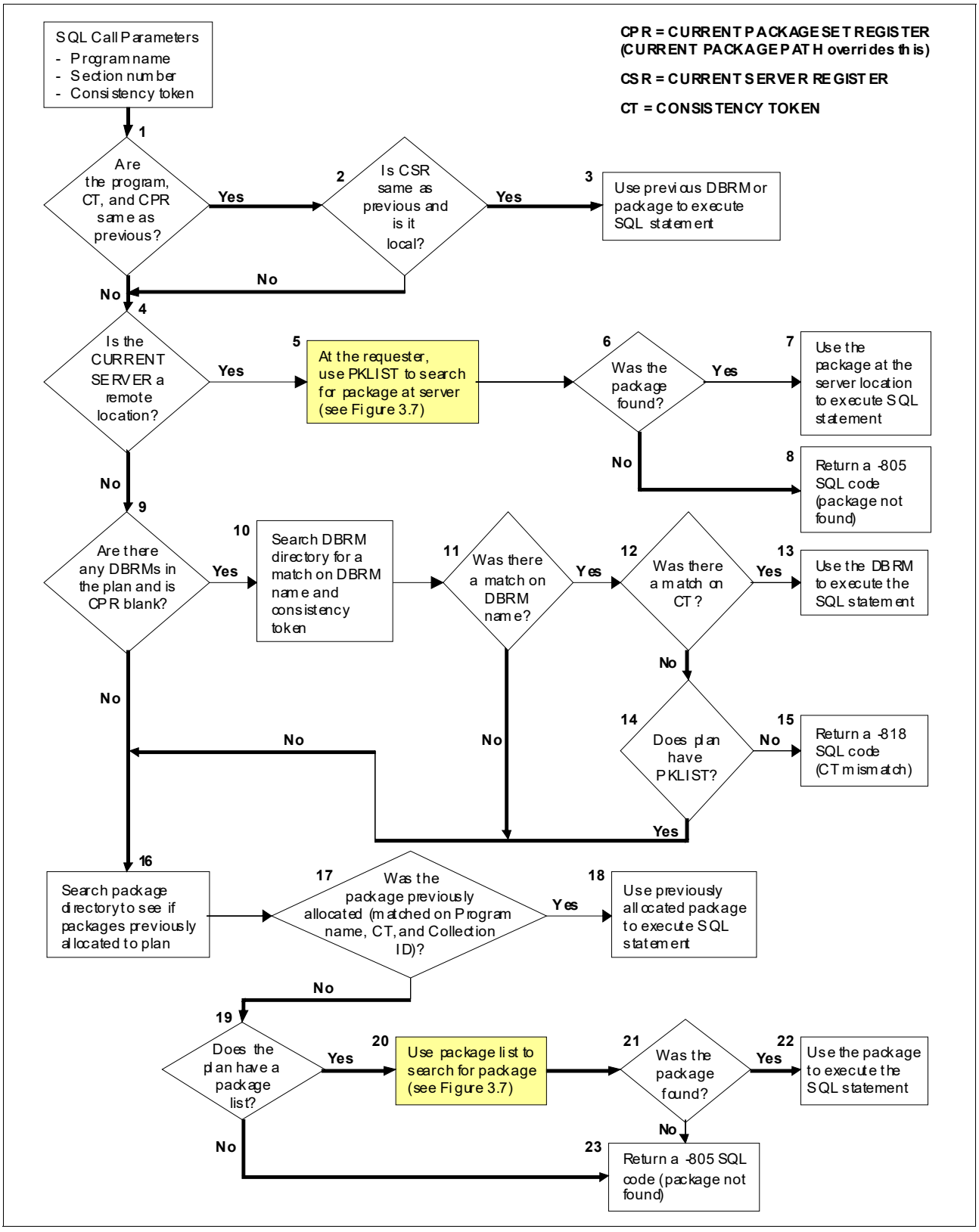

Figure 3-6 Plan processing - the "big picture"

The numbers associated with these descriptions match the numbers in Figure 3-6:

1. The program name and consistency token passed from the program and the contents of the CURRENT PACKAGESET register (or CURRENT PACKAGE PATH register, which overrides this) are compared to the values for these variables stored from the last SQL request received for the plan. The check is basically to see whether this SQL statement came from the same module as the previous SQL statement because, if it did, search processing is simplified.
2. The contents of the CURRENT SERVER register are compared to the value stored from the previous request. If they are equal, the value of this register is checked to see whether it references the local location name. This name is blank if there is no distributed processing, or it is the name in the SYSIBM.LOCATIONS table for this location if distributed processing is supported.
3. Steps 1 and 2 have established that this request has come from the same program module as the previous request, so the DBRM or package for the program that was used for the last request is used for this request. Steps 1, 2, and 3 constitute the most common path in this process.
4. Steps 1, 2, and 3 indicate that the value for at least one of four parameters checked—DBRM name, consistency token, CURRENT PACKAGESET special register (or CURRENT PACKAGE PATH register, which overrides this), and CURRENT SERVER special register—has changed. Here, the value of the CURRENT SERVER special register is checked to see whether it identifies a remote location.
5. If the request is for a remote location, the PKLIST for the plan at the requester site is searched to locate a package bound at the server site that can process the request. The processing involved in this search is shown in Figure 3-7 on page 46.
6. A number of packages could be found in the PKLIST search to satisfy the search criteria. At this stage, the location name, DBRM name, and consistency token are known. If the CURRENT PACKAGESET special register (or CURRENT PACKAGE PATH register, which overrides this) is set, the name of the package required is fully qualified.

   If it is not set, a number of packages can be identified as possible candidates to meet the requirements. For example, if the contents of the CURRENT SERVER special register is LOCATION1 and the DBRM name is PGMA, the package list entries and the candidate packages would be:

   ```
   PKLIST Entry                  Type Candidate Package
   LOCATION1.COLL1.PGMA           LOCATION1.COLL1.PGMA
   *.COLL2.PGMA                   LOCATION1.COLL2.PGMA
   *.COLL3.*                      LOCATION1.COLL3.PGMA
   ```

   Notes:

   - An entry type of *.*.* is not valid because the CURRENT PACKAGESET register (or CURRENT PACKAGE PATH register, which overrides this) has not been set.
   - COLL1, COLL2, and COLL3 represent nonblank collection IDs.

   The candidate packages are passed to the server for verification in the sequence in which the entries on which they are based are stored in the package list.
7. If a package is found at the server site that matches the location name, collection ID, package name, and consistency token, it is allocated and used to process the SQL statement.
8. If a match is not found on any of the package names passed to the server, the requester returns a “package not found” (-805 SQL code and reason code) to the program.

9. For the local location, a check is made on the plan header for the presence of DBRMs in the plan, and the value in the CURRENT PACKAGESET special register (or CURRENT PACKAGE PATH register, which overrides this) is compared to blanks. Where both conditions are true, processing continues at Step 10, otherwise at Step 16.
10. A search is made of the DBRM directory in the plan header for an entry that matches the DBRM name and consistency token passed from the program.
11. If a match is found on the DBRM name, processing continues to check the consistency tokens. Otherwise, processing continues at Step 16.
12. A check is made for a match in consistency tokens between the value passed from the program and the value stored in the DBRM directory.
13. Because both the DBRM name and consistency token match, the required DBRM has been identified, and it is allocated to the plan to process the SQL statement.
14. If the consistency tokens do not match, a check is made of the plan header for the presence of a package list. If a package list exists, processing continues at Step 16 to try to locate a package that meets the search criteria. If a package list is not associated with this plan, the DBRM bound into the plan was not generated in the same precompiler run as the modified program source.
15. As a result of the consistency token mismatch, a -818 SQL code is returned to the program and processing ends.
16. A check is now made in the package directory for the plan to determine whether a package previously allocated to this plan meets the search criteria.
17. Because it is known that the location name refers to the local location, this check requires matching on three variables, namely collection ID, package name (program name), and consistency token.

    When a package is allocated to a plan, an entry is placed in the package directory for the plan, with information related to the contents of the CURRENT PACKAGESET register (or CURRENT PACKAGE PATH register, which overrides this) appended. This information and the DBRM name and consistency token passed from the program are used in the search for a matching package in the package directory.
18. If a match is found in the package directory, the previously allocated package is used to execute the SQL statement.
19. If a match is not found in the package directory, a check is made of the plan header for the presence of a package list.
20. If the plan contains a package list, this list is searched to locate a package to process the SQL statement. This search is shown in Figure 3-7.
21. This step checks the result of the package search process.
22. If a package is found, it is allocated to the plan and used to execute the SQL statement passed from the program.
23. If a package is not found, a -805 SQL code is returned to the program indicating that a package could not be found in this plan to process the SQL request.

3.2.3 Package list search sequence

DB2 uses package list search (see Figure 3-7) to process the entries in the sequence specified in the PKLIST for the plan to identify the package required to process the SQL request. This processing is described next. The numbers in the description match the numbers in the figure.

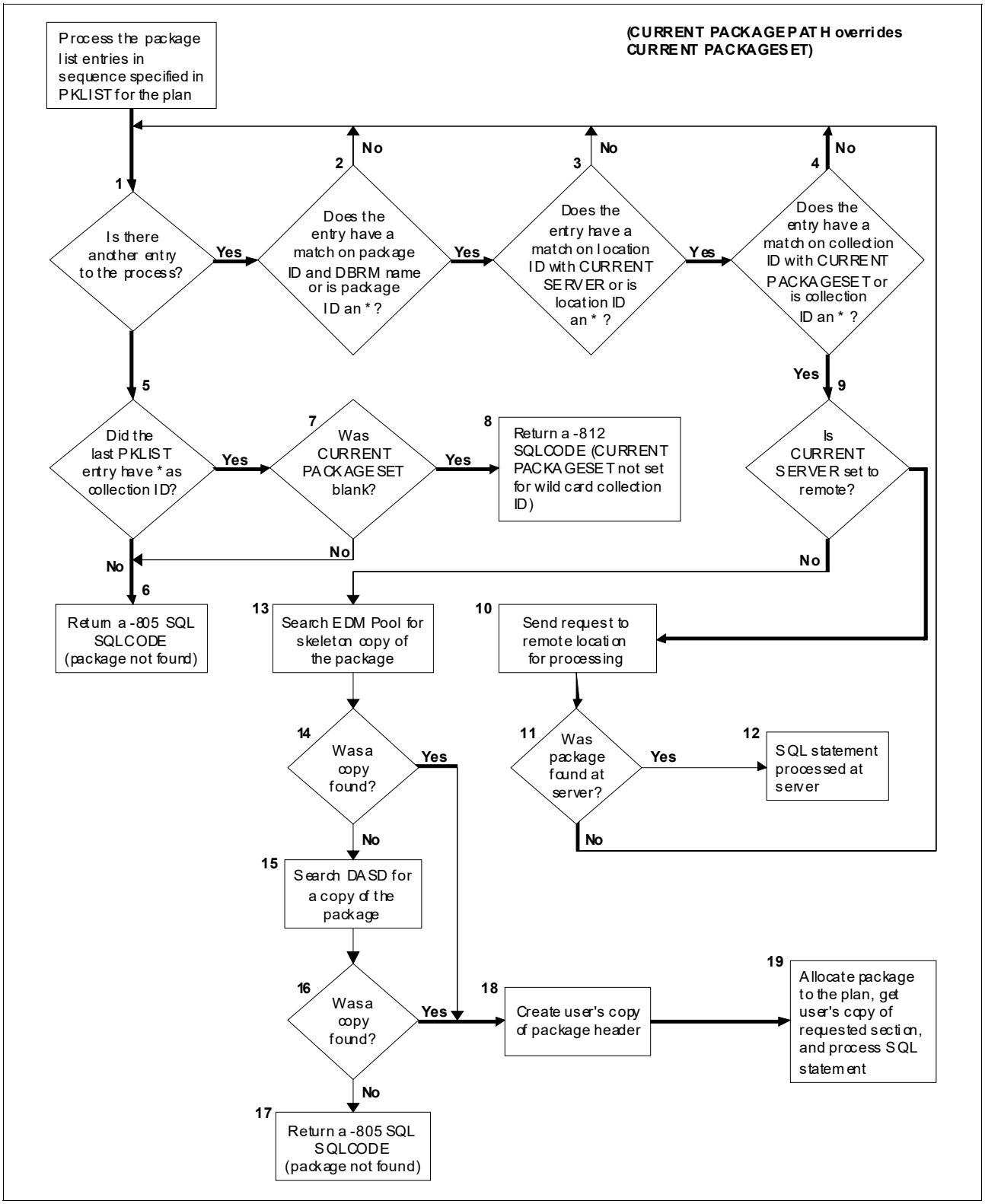

Figure 3-7 Package list search

1. The entries in the package list are processed in the sequence specified in the PKLIST until either a package is located that meets the search criteria or the list is exhausted.

2. A comparison is made between the DBRM name passed with the SQL request and the package name for the package list entry. If they match or if the package name is a wild card (that is, an asterisk), processing continues at Step 3. Otherwise, processing returns to Step 1 to obtain another package list entry.

   If the DBRM name passed is PGMA, the types of package list entries that would match in this step include:

   ```
   LOCNn.COLLn.PGMA                    *.COLLn.PGMA
   LOCNn.COLLn.*                             *.COLLn.*
   LOCNn.*.PGMA                              *.*.PGMA
   LOCNn.*.*                                       *.*.*
   ```

   Note that LOCNn and COLLn represent variable non-blank values.

3. A comparison is made between the location name in the package list entry and the value in the CURRENT SERVER register. If they match or if the location name for the entry is a wild card, processing continues at Step 4. Otherwise, processing returns to Step 1 to obtain the next entry in the package list.

   The entries that matched in Step 2 are processed as follows:

   – If the content of the CURRENT SERVER register is LOCATION1:
     - Entries with a location name of LOCNn are passed where LOCNn is equal to LOCATION1.
     - Entries with a wild card location name are passed with a location name of LOCATION1.

   – If the content of the CURRENT SERVER register is blank:
     - Entries with a location name of LOCNn are passed where LOCNn is equal to the local location.
     - Entries with a wild card location name are passed with the local location as the location name.

4. Processing similar to that in Step 3 for location name is done in this step for collection ID.

   The processing of the different types of entries, for which only collection ID and package name are shown, is summarized as follows:

   – If the content of the CURRENT PACKAGESET register (or CURRENT PACKAGE PATH register, which overrides this) is COLL1:
     - Entries with a collection ID of COLLn are passed where COLLn is equal to COLL1.
     - Entries with a wild card collection ID are passed with a collection ID of COLL1.

   – If the content of the CURRENT PACKAGESET register (and CURRENT PACKAGE PATH register, which overrides this) is blank:
     - Entries with a collection ID of COLLn are passed unchanged.
     - Entries with a wild card collection ID are skipped, and processing returns to Step 1 to obtain a new entry.

5. If the end of the package list has been reached and the package not found, the last entry processed is checked for a wild card collection ID.
6. If the last package list entry did not have a wild card collection ID, a -805 SQL code is returned to the program to indicate that the package was not found.
7. If the last package list entry had a wild card collection ID, a check is made on the contents of the CURRENT PACKAGESET register (or CURRENT PACKAGE PATH register, which overrides this). If the register had a nonblank value, processing continues at Step 6, where a -805 SQL code is returned to the program.

8. If the last entry in the package list contained a wild card collection ID and the CURRENT PACKAGESET register (and CURRENT PACKAGE PATH register, which overrides this) is blank, a -812 SQL code is returned to the program.
9. At this point, the package required can be fully qualified because the location name, collection ID, package name, and consistency token are known. A check is now made on the value in the CURRENT SERVER register to determine whether the request should be shipped to a remote location.
10. The package identified by the package search and the SQL request is shipped to the remote server identified by the location name.
11. If the package identified by the search is not found at the server, processing transfers to Step 1 to obtain another package list entry.
12. If the package is found at the server, it is allocated to a common plan DISTSERV, which is used to process the remote SQL request.
13. If the request is for the local DB2 subsystem, a search of the EDM pool is made first to determine whether the skeleton copy of the package, SKPT, is already in the pool.
14. If the SKPT is found in the EDM pool, processing transfers to Step 18.
15. If the SKPT is not found in the EDM pool, a search for the package is made in the SPT01.
16. If the package is found in SPT01, processing transfers to Step 18.
17. If the package is not found in SPT01, a -805 SQL code is returned to the program, indicating that the package has not been found.
18. If the package is located in either the EDM pool or SPT01, a copy of the skeleton package header is made for the user.
19. The package is then allocated to the plan. The section of the package requested in the SQL statement is allocated, and the SQL request is processed.

3.2.4 Implications of the search process

Two conclusions can be drawn from the package search process:

- DB2 does not return a -818 SQL code when plan contains only packages (no DBRMs). The reason for this is that a DBRM can be bound only once into a plan through the MEMBER parameter, but it can be included many times as a package through the use of collections and/or versioning.

  Therefore, when there is a consistency token mismatch for a DBRM and the plan contains packages, the assumption is that a package is required. If a package is not found that matches, a -805 SQL code is returned. See 7.1, “-805 errors” on page 150 for details on how to diagnose and correct this type of error.
- When the CURRENT PACKAGESET register (or CURRENT PACKAGE PATH register which overrides this) is not set in a program, the use of multiple wild card entries such as COLL1.* in the package list can degrade performance. For example, consider the use of a package list such as:

  ```
  PKLIST=(COLL1.*, COLL2.*, COLL3.*, .... COLL100.*)
  ```

  In the worst case, this would result in 100 scans to SPT01 if the required package is in COLL100.

  See 11.1, “Binding parameters that impact performance” on page 224 for details.

4

Conversion to packages

In this chapter we provide considerations related to moving to an application environment that uses packages. We discuss conversion planning, how to modify existing plans to use packages, and the use of the new REBIND command, which converts plans with DBRMs to plans with packages.

We cover the following topics:

- Conversion planning
- CICS dynamic plan switching
- Modifying existing plans to use packages
- Impact on DB2 catalog and REBIND panel
- Conversion examples

4.1 Conversion planning

DBRM-based plans are deprecated in DB2 9 for z/OS. That is, no enhancements are expected to be implemented and, in a future release, these types of plans might no longer be supported. In preparation for this planned change, you should stop defining the DBRM-based plans starting now and convert as soon as PTF UK43291 for APAR PK62876 is available and applied to your DB2 9 for z/OS.

4.1.1 Planning checklist

Here is a planning checklist to assist you in preparing for your conversion effort:

- Determine the number of plans affected
- List the plans and the number of DBRMs
- Set standards for your package objects: plans, collections, versions
- Determine if you have situations where you have DBRMs in multiple plans
- Determine if you have DBRMs within like-named plans
- Determine if these like-named DBRMs are different, have different consistency tokens or different bind options
- Create a strategy to convert and test the plans
- Run the conversions at a pace you can control

The first item to complete when planning your migration from DBRM-based plans to packaged-based plans is to determine the scope of your project. How many plans do you need to convert? You need to decide on strategies about how you divide up your collections. Are you using CICS dynamic plan switching?

You need to decide on naming standards for your collections. Determine if you will use VERSIONs and if so, how you will use them. See “Contrasting collection, version, and consistency CONTOKEN” on page 76 for more information on developing with VERSIONs. You need to decide on procedures to back out changes using packages. You need to benchmark the performance of your plans, or at least have a copy of your PLAN_TABLE so you can determine if the access paths have changed. What kind of testing will you do to show that everything works? We focus on these issues in this section.

4.1.2 Determining the scope of your project

Your first task is to determine how many DBRM-based plans you have in your system. The query in Example 4-1 shows you the number of DBRM-based plans in your system.

Example 4-1 Count of plans with DBRMs

```
SELECT COUNT(DISTINCT PLNAME) AS NUMBER_OF_PLANS FROM SYSIBM.SYSDBRM
```

If the count is zero, you are done.

Your second task is to list all of these plans, so you can understand how critical these plans are to your enterprise.

The query in Example 4-2 lists out the plans that you need to convert to packages.

Example 4-2 List of plans and how many DBRMs are in each plan

```
SELECT PLNAME AS PLAN_NAME,COUNT(*) AS NUMBER_OF_DBRMS
FROM SYSIBM.SYSDBRM
```

```
GROUP BY PLNAME
ORDER BY PLAN_NAME
```

The query in Figure 4-3 lists all of the DBRMs that are directly bound within plans.

Example 4-3 List of DBRMS within PLAN

```
SELECT SUBSTR(PLNAME,1,8) AS PLAN_NAME
      ,SUBSTR(NAME,1,8)   AS DBRM_NAME
FROM SYSIBM.SYSDBRM
ORDER BY PLAN_NAME,DBRM_NAME
```

4.2 CICS dynamic plan switching

One of the biggest problems with DBRM-based plans is that in order to make a change to one program, you needed to BIND the entire plan, which included all DBRMs associated with that plan. One of the early ways to reduce the number of DBRMs in a CICS plan was to code your programs with a dynamic plan selection algorithm, which would switch the plan based upon application parameters. The Resource Definition Online (RDO) would be set to use dynamic plan switching. One benefit of using packages is that you can eliminate the dynamic plan selection exit and have your CICS RDO use a specific PLAN.

An example would be if you had a financial application which used 40 different plans to do forty different functions with each function using a separate plan. CICS was set up to use a dynamic plan selection exit, so that each time a new transaction was started, a new plan would be used. There was overhead in CICS to accomplish this as well as administrative tasks that needed to be done in DB2 and possibly application code that needed to be done to start the new transaction.

By switching to packages, the CICS exit is not needed. DB2 can be set up with one plan. The plan could include one collection (such as FIN_PROD_CICS), and all of the financial application packages can be included in this collection. The application code that was handling starting the new transactions can be eliminated.You can now take advantage of thread reuse, because you are not changing the PLAN name throughout the execution of the application.

Recommendation: Use DB2 package lists within your plans instead of a CICS dynamic plan selection routine. This allows you to bind one program at a time in a CICS environment and benefit from thread reuse.

Keep in mind that running a large plan with many packages needs a change in monitoring strategy to packages from plans.

Also consider that thread reuse is very effective for very small, light transactions with a high transaction rate. Thread reuse is important for a case with one to three SQL statements per transaction running 100 tps. When the transaction has 10, 20, or 100 SQL statements, the value is diminished sharply. If the transaction rate is less than 10 per second or the number of SQL statements per transaction is more than 20, then the value of thread reuse is minimal. Overheads in storage and memory management are larger for the complex transactions.

When the value of thread reuse is minimal, then the costs of keeping the threads, in memory and CPU, generally exceeds the value.

For more information, see *CICS Transaction Server for z/OS V3.1 CICS DB2 Guide,* SC34-6457.

4.3 Modifying existing plans to use packages

After you have determined which plans and DBRMs you need to convert, you bind each of the DBRMs into a collection or collections.This can be done before you make any changes to the plan. After all of your DBRMs have been bound into packages, you need to BIND your plan to include the packages. The change you need to do this is to remove the MEMBER parameter in your BIND control cards and replace them with a PKLIST parameter. See Example 4-4 to see the difference between the two methods.

Example 4-4 Comparison of BIND command using DBRMs and packages

```
Basic command to BIND PLAN with DBRMs
   BIND PLAN FINPLAN MEMBER(PGM01, PGM02)

Basic command to BIND PLAN with packages from collection FINTEST
   BIND PLAN FINPLAN PKLIST(FINTEST.PGM01,FINTEST.PGM02)
```

You can list each of the packages individually or use “collection-id.*” (package list) to include all of the packages in the collection. You can add multiple collection lists into the plan as well. See 5.3, “Contrasting collection, version, and consistency CONTOKEN” on page 76 for information on setting up your collections.

Tip: Before binding the new packages into the plan, compare the new access paths to the current access paths and investigate any differences.

4.3.1 REBIND command to convert to package-based plans

Included via APAR PK62876 in DB2 Version 8 for z/OS and DB2 9 for z/OS is a new option of the REBIND PLAN command that converts DBRM-based plans to packaged-based plans.

There are choices you can make when determining your conversion strategy. You need to be aware of their consequences.

COLLID option

To convert DBRMs bound within plans into packages, a new REBIND PLAN option COLLID (collection name or *) is introduced to perform the DBRM to package conversion. During the REBIND PLAN process, DB2 reads the SYSDBRM table to retrieve the DBRMs bound with the plan, binds them into packages and binds the packages into the plans. Each package is listed separately in the PKLIST, not as a *collection*.*. Finally, the REBIND deletes the DBRMs bound into the plans. After the rebind, there are no longer any SYSDBRM or SYSSTMT records associated with the plan. The SYSPACKAGE and SYSPACKSTMT tables are populated with new package statement data.

A description of the COLLID options are listed in Table 4-1.

Table 4-1 Options of the COLLID parameter

PLAN	COLLID	Action
Single plan	Explicitly named	Packages are bound into the named COLLID
	*	Packages are bound into the DSN_DEFAULT_COLLID_*planname* collection
Individually listed	Explicitly named	Packages are bound into the named COLLID
	*	Packages are bound into the DSN_DEFAULT_COLLID_*planname* collection
All plans	Explicitly named	Packages are bound into the named COLLID
	*	Packages are bound into the DSN_DEFAULT_COLLID_*planname* collection

If you specify COLLID(*), DB2 uses a default collection of “DSN_DEFAULT_COLLID_xxxxx” where xxxxx represents the plan being rebound.

If you wish to specify a specific collection for the packages to be bound into, you specify the collection name in the COLLID parameter. See Example 4-5 for the collection names generated by the REBIND COLLID command.

Example 4-5 Collection names generated with COLLID(option) for plan FINTEST1

```
REBIND PLAN(FINTEST1) COLLID(*)
generates a collection name of DSN_DEFAULT_COLLID_FINTEST1

REBIND PLAN(FINTEST1) COLLID(FINTEST)
generates a collection name of FINTEST
```

4.3.2 Package order in the converted package list

The order of the converted package list is important during the execution of a plan. Currently, if a plan has both DBRMs and packages, the search order is to look for the DBRMs first and, if found, use them. If the DBRM is not found, then the package list is searched. The conversion honors the current search order by creating the package list with the old DBRMs listed *before* any other package lists. This makes a converted plan behave the same as a DBRM-based plan.

4.3.3 Conditions where you could lose a DBRM

There are instances where you can overlay either existing packages or newly created packages during the conversion, thus losing a DBRM permanently. This can happen when:

You use the COLLID(collection ID) option of the REBIND command and either:

- One DBRM is listed in multiple plans and is somehow different in the plans and you use the same collection ID for both.

  or

- A DBRM is also a package in a collection that you specify in the COLLID option.

In the following three situations, you can overlay a package and lose the existing DBRM:

- PGM01 is precompiled into a DBRM at time 1 and directly bound into PLAN FINTEST1 and linked into load module 1. A change is made to PGM01 that you did not want to make, or you forgot to incorporate into PLAN FINTEST1, but you kept the same name. PGM01 is directly bound into PLAN FINTEST2 and linked into a different load module.
- PGM01 has been bound directly into a PLAN. PGM01 has also been bound into collection FIN_TEST_BATCH as a package.
- The BIND options can be different between the DBRM and package, or between two copies of the DBRM bound into different plans.

If you do not use the default collection, or if you create a *new* collection to put your converted DBRMs in without including the PLAN in the name of the collection, you can overlay packages in existing or new collections. Then, you might not be able to recreate these packages if you do not have the existing DBRM in a data set.

Recommendation: Use the default collection to perform all of your conversions, or use the COLLID option to create new collections and include the plan name in the collection name. Use the COPY option of the BIND PACKAGE command to move the new packages into the proper collections.

4.3.4 Identically named DBRMs in different plans with different consistency tokens

By using the COLLID(collection ID) option on the REBIND PLAN command, if you have the same DBRM in multiple plans, you overlay the newly created package in the collection each time you run the REBIND command for individual plans, or for each plan in a list if you specify multiple plans on the REBIND PLAN command. To list the DBRMs that are in multiple PLANs, see Example 4-6.

Example 4-6 Listing DBRMs in multiple plans

```
SELECT
 DBRM.NAME   AS DBRM_NAME
,DBRM.PLNAME AS PLAN_NAME
,COUNT(DISTINCT HEX(DBRM.TIMESTAMP))
FROM SYSIBM.SYSDBRM DBRM,
    (SELECT SUBSTR(DBRM1.NAME,1,8)       AS DBRM_NAME,
            COUNT(DISTINCT DBRM1.PLNAME) AS NUMBER_OF_PLANS
     FROM SYSIBM.SYSDBRM DBRM1
     GROUP BY SUBSTR(DBRM1.NAME,1,8)
     HAVING COUNT(DISTINCT DBRM1.PLNAME) > 1) AS MULT_DBRM
WHERE DBRM.NAME = MULT_DBRM.DBRM_NAME
GROUP BY DBRM.NAME,DBRM.PLNAME
HAVING COUNT(DISTINCT HEX(DBRM.TIMESTAMP)) > 1
ORDER BY DBRM_NAME,PLAN_NAME
```

4.3.5 Identically named DBRMs in different plans with different options

By using the COLLID(collection ID) option on the REBIND PLAN command, if you have DBRMs bound into multiple plans with different options, such as CHARSET, MIXED, COMMA, QUOTE or from a different PDS, you can overlay one package with the options from another. To see if you have DBRMs in multiple plans with different characteristics, see Example 4-7.

Example 4-7 Checking for multiple DBRMs with differences that you need to account for when using the COLLID option of REBIND PLAN

```
   SELECT DISTINCT DBRM1.PLNAME,DBRM1.NAME
   FROM SYSIBM.SYSDBRM DBRM1,SYSIBM.SYSDBRM DBRM2
   WHERE (DBRM1.NAME = DBRM2.NAME
     AND  DBRM1.PLNAME        <>DBRM2.PLNAME)

     AND (DBRM1.TIMESTAMP     <> DBRM2.TIMESTAMP
      OR  DBRM1.PDSNAME       <> DBRM2.PDSNAME
      OR  DBRM1.PLCREATOR     <> DBRM2.PLCREATOR
      OR  DBRM1.PRECOMPTIME   <> DBRM2.PRECOMPTIME
      OR  DBRM1.PRECOMPDATE   <> DBRM2.PRECOMPDATE
      OR  DBRM1.QUOTE         <> DBRM2.QUOTE
      OR  DBRM1.COMMA         <> DBRM2.COMMA
      OR  DBRM1.HOSTLANG      <> DBRM2.HOSTLANG
      OR  DBRM1.IBMREQD       <> DBRM2.IBMREQD
      OR  DBRM1.CHARSET       <> DBRM2.CHARSET
      OR  DBRM1.MIXED         <> DBRM2.MIXED
      OR  DBRM1.DEC31         <> DBRM2.DEC31
      OR  DBRM1.VERSION       <> DBRM2.VERSION
      OR  DBRM1.PRECOMPTS     <> DBRM2.PRECOMPTS
      OR  DBRM1.PLCREATORTYPE <> DBRM2.PLCREATORTYPE
      OR  DBRM1.RELCREATED    <> DBRM2.RELCREATED)
   ORDER BY DBRM1.NAME,DBRM1.PLNAME
WITH UR
```

4.3.6 DBRMs that are also bound as packages

By using the COLLID(collection ID) option on the REBIND PLAN command, if you have DBRMs with the same name as packages that already exist in the specified collection, you can overlay them with the converted package from the DBRM. To determine if you have queries that cause you to bind the same package into the same collection and overlay a different copy of the package, see Example 4-8.

Example 4-8 Query to determine if there are DBRMs and packages with the same name

```
SELECT
 SUBSTR(DBRM.NAME,1,18)    AS DBRM_NAME
,SUBSTR(PACK.LOCATION,1,8) AS LOCATION
,SUBSTR(PACK.COLLID,1,18)  AS COLLID
,SUBSTR(PACK.NAME,1,18)    AS PACKAGE
FROM SYSIBM.SYSDBRM    DBRM
    ,SYSIBM.SYSPACKAGE PACK
WHERE DBRM.NAME = PACK.NAME
ORDER BY PACK.COLLID,PACK.NAME
WITH UR
```

By using the COLLID(*) option on the REBIND PLAN command, you can create many collections that are incorporated into the plan. This is the safest method of performing this conversion. By using this method, you always have a copy of the original DBRM in the DB2 catalog in the SYSPACKAGE table with the default collection ID. You can use the COPY option of the BIND PACKAGE command to copy the package into collections that conform to your naming standards after the conversion has completed.

However, if you have plans with large numbers of packages, there are performance overheads. The overhead is worse in case of thread reuse with low volume transactions.

4.3.7 Rebinding a single PLAN

In the next section we discuss in detail the various ways that you can convert your DBRM-based plans.

In this section we discuss binding a single plan with either the default collection or a named collection. We describe the use of the PKLIST option in 4.3.8, “Rebinding multiple plans” on page 57. We explore each option and provide reasons to use or not use it.

REBIND PLAN(FINTEST1) COLLID(*)

This command does a REBIND PLAN FINTEST1 and converts any DBRMs directly bound into plan FINTEST1 into packages into collection: DSN_DEFAULT_COLLID_*FINTEST1*. Using this option avoids having packages overlaid, because each DBRM has the plan as part of the collection ID. If there are both DBRMs and packages in this plan, entries from the newly converted DBRMs are inserted in front of the existing package list. The packages in the new collection DSN_DEFAULT_COLLID_FINTEST1 are individually listed at the beginning of the PKLIST option.

Recommendation: Use this form of the REBIND PLAN command to convert all of your DBRMs if the earlier queries show that you have multiple versions of DBRMs in multiple plans or you have DBRMs also bound into a collection. This is the safest way to convert all of your PLANS to contain only packages. The downside to this method is that you generate one collection per plan. You will eventually want to safely consolidate these collections.

REBIND PLAN(FINTEST1) COLLID(collection ID)

Use this form of the command to convert the DBRMs from one specific plan into packages in a specific collection. If there are both DBRMs and packages in this plan, entries from the newly converted DBRMs are inserted in front of the existing package list.

If you do not have any DBRMs that are in multiple plans with different CONTOKENs and you do not have any DBRMs that are also packages in this collection (see Example 4-6 on page 54, Figure 4-7 on page 55 and Example 4-8 on page 55), then you can safely bind all of your plans into the collections you choose using this method.

If you have a DBRM that is either used in multiple plans or also has been bound into a collection, be careful that you do not overlay the current package while converting. The conversion process removes the DBRM and you will not be able to recreate it if you do not have the original DBRM in a dataset.

4.3.8 Rebinding multiple plans

There are eight combinations that can be used with the COLLID and PKLIST options in the REBIND PLAN command. The PKLIST option is shown here to demonstrate that you can change the order of the existing PKLIST while performing the conversion. It also demonstrates that you can change other options in the package while performing the conversion. The combinations are listed in Table 4-2.

Table 4-2 Multiple PLAN binds

PLAN	COLLID	PKLIST	
Individually listed	Explicitly named	Not specified	Packages are be bound into the named COLLID.
	*	Not specified	Packages are bound into the DSN_DEFAULT_COLLID_*planname* collection.
	Explicitly named	Specified	Packages are bound into the named COLLID. Created packages are at the beginning of the PKLIST, specified PKLISTs follow.
	*	Specified	Packages are bound into the DSN_DEFAULT_COLLID_*planname* collection. Created packages are at the beginning of the PKLIST, specified PKLISTs follow.
*	Explicitly named	Not specified	Packages are bound into the named COLLID
	*	Not specified	Packages are bound into the DSN_DEFAULT_COLLID_*planname* collection.
	Explicitly named	Specified	Packages are bound into the named COLLID. Created packages are at the beginning of the PKLIST, specified PKLISTs follow.
	*	Specified	Packages are bound into the DSN_DEFAULT_COLLID_*planname* collection. Created packages are at the beginning of the PKLIST, specified PKLISTs follow.

You cannot use the command REBIND PLAN(FIN*) to rebind all of the plans that begin with FIN. If you use any form of the REBIND PLAN(*) command, you rebind *all* of your plans, not just the plans with directly bound DBRMs.

REBIND PLAN(FINTEST1,FINTEST2) COLLID (collection_id or *)

The REBIND processes each plan specified in the list and does the conversion accordingly, until all plans specified have been converted.

REBIND PLAN (*) COLLID (collection_id or *)

You can convert ALL of the plans listed in SYSPLAN in one command using the form REBIND PLAN (*) COLLID (collection_id or *). The REBIND processes *all* plans in the system, not just the DBRM-based plans and performs the conversion of each plan until all plans have been converted. Collection name can be explicitly specified or defaulted by specifying an asterisk. For plans with existing package lists, the converted packages are listed individually at the beginning of the package list.

Note: While the REBIND command allows you the flexibility of converting all plans in your system at once, we advise against this practice and suggest you perform your conversion in a controlled manner to improve the probabilities of success.

REBIND PLAN (plan list or *) COLLID (collection_id or *) PKLIST(col1.*)

You can convert all of the plans in your system, or a list of plans using this form of the REBIND PLAN command. The REBIND converts each plan until all plans have been converted. Collection name can be explicitly specified or defaulted by specifying an asterisk. Using this form enables you to specify the PKLIST(s) that follow the converted packages in the PKLIST.

Important: If choosing this option, avoid overlaying any packages. Be careful if you specify a different PKLIST than the current PKLIST.

REBIND PLAN (FINTEST1) COLLID (collection_id *) NOPKLIST

You would use the NOPKLIST option to remove the current PKLIST. Use this option with caution. Table 4-3 shows the options for using NOPKLIST.

Table 4-3 NOPKLIST processing

Plan contains	What happens to existing plans
DBRMs and packages	Deleted and replaced by converted packages
DBRMs only	Converted to package based plans
Packages only	Existing PKLIST deleted

- If plan FINTEST1 contains both DBRMs and PKLIST, the existing package list is deleted, while the new package list that is converted from the DBRMS are bound into plan FINTEST1.
- If plan FINTEST1 only has DBRMs, the DBRMs is converted into packages accordingly, and NOPKLIST is ignored.
- If plan FINTEST1 has only packages, no conversion is performed and the current PKLIST is removed from the plan.

Important: If Plan FINTEST1 does not have any DBRMs, using the NOPKLIST option results in the existing package list being deleted. The PLAN is redefined with no DBRMs and no package list. You will not be able to execute any SQL using this plan.

4.4 Impact on DB2 catalog and REBIND panel

This section covers changes to the DB2 catalog tables as well as a change to the REBIND panel to allow you to specify the conversion from the DB2I REBIND panel.

4.4.1 REBIND panel parameter

A new parameter CONVERSION COLLECTION ID is added to the DSNEBP03 panel. This field is meaningful only when you REBIND a plan that was bound with the MEMBER keyword. and it can assume the following specifications:

- Blank

  A blank in this field means that the plan is not to be converted during REBIND processing.

- *

  An asterisk in this field means that the collection name are composed of "DSN_DEFAULT_COLLID_" concatenated with the plan name.

- Value

  A user specified value for the collection to hold the converted DBRMs

This field corresponds to the COLLID option of the REBIND statement. Refer to *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844 for information about this option.

4.4.2 Impact on DB2 catalog tables

The SYSPACKAGE and all corresponding catalog tables (SYSPACKAUTH, SYSPACKDEP, SYSPACKSTMT, SYSPKSYSTEM) are updated when a REBIND package command is executed.

The SYSSTMT table is read to access all statements for the DBRM to be converted. The entries in SYSSTMT are deleted.

Tables SYSPLAN and SYSPACKLIST are updated with the information for the new package lists.

SYSDBRM contains the information for DBRMs bound into a specific plan. SYSSTMT has all the statements associated with a particular DBRM. After the REBIND PLAN with the COLLID option specified has run, there are no directly bound DBRMs left associated with the plan(s). The rows which were associated with the original plan are removed.

See also Appendix C.1, "Package related catalog tables" on page 288.

4.5 Conversion examples

The examples that are listed here show the following conditions:

- Example 1: This shows how to convert plans, one at a time, and use the COLLID(*) option of the REBIND PLAN command. This is the safest method of performing the conversion. This leaves you with one collection per plan with the default name, which you want to safely integrate with your current collections.
- Example 2: This shows how to convert plans, one at a time, using the same named collection in the COLLID option for both rebinds. In this example, we show how you can overlay a DBRM and lose the only original copy of the DBRM that you have. *This is dangerous and you need to be aware of the implications if you decide to use this method.* The reason that you would want to use this method would be that when the conversion was over, you would have your packages in the proper collection. Mitigate this risk with careful checks before using this approach.
- Example 3: This shows how a conversion would look when you have both DBRMs and package lists already in the plan. We show how the converted packages are listed at the beginning of the new PKLIST in front of the current PKLIST.
- Example 4: This shows how a conversion of multiple plans would look when you have both DBRMs and package lists already in the plan. This example uses a named collection to show that we overlay existing packages. We show here how the converted packages are listed at the beginning of the new PKLIST.

The naming convention that we use in this section is that programs with T_1 after them are considered to be the original program. Programs with T_2 after them are considered to be newer copies with a higher CONTOKEN in SYSDBRM than the T_1 copies.

Note: The CONTOKENs for each DBRM are different in each plan for this example, although the DBRM names are identical.

Table 4-4 shows the contents of SYSDBRM after we have bound plans FINTEST1 with the T_1 versions of the program and FINTEST2 with the T_2 versions of the programs.

Table 4-4 SYSDBRM before running REBIND with COLLID option

DBRM name	PLAN name	CONTOKEN
PGM01	FINTEST1	1866330A1F71E2C6
	FINTEST2	18662C920BC8D185
PGM02	FINTEST1	18662C920A072A30
	FINTEST2	18662C920C96574E
PGM03	FINTEST1	18662C920AF3A02E
	FINTEST2	18662C920D849192

Note: In this example the DBRM names are the same, but the CONTOKENs are different.

4.5.1 Example 1: Converting one plan, default collection, DBRMs only

In this example we show how we convert plan FINTEST1, which contains only DBRMs PGM01(T_1), PGM02(T_1), and PGM03(T_1), using the COLLID(*) option. After we convert plan FINTEST1, we convert plan FINTEST2, which contains only DBRMs PGM01(T_2), PGM02(T_2), PGM03(T_2) using the COLLID(*) option.

The main point to see from this example is that by using the default COLLID option, COLLID(*), we do not overlay like-named packages.

To convert plan FINTEST1 to a package-based plan, issue the REBIND command in Example 4-9.

Example 4-9 REBIND PLAN FINTEST1 using default collection

```
REBIND PLAN(FINTEST1) COLLID(*)
```

This binds plan FINTEST1 with packages PGM01(T_1), PGM02(T_1) and PGM03(T_1) in collection DSN_DEFAULT_COLLID_FINTEST1.

Sample output from this REBIND command is shown in Example 4-10.

Example 4-10 Sample output of REBIND FINTEST1 COLLID(*) command

```
REBIND PLAN(FINTEST1) COLLID(*)
DSNT254I  -DB9A DSNTBCM2 REBIND OPTIONS FOR
           PACKAGE = DB9A.DSN_DEFAULT_COLLID_FINTEST1.PGM01.()
```

The only confirmation that you converted a plan in the output of the REBIND command is that you can see the default collection ID mentioned in message DSNT254I.

The contents of SYSPACKAGE after the conversion of plan FINTEST1 are shown in Table 4-5.

Table 4-5 SYSPACKAGE after REBIND PLAN(FINTEST1) COLLID(*)

COLLECTION	PACKAGE	CONTOKEN	VERSION
DSN_DEFAULT_COLLID_FINTEST1	PGM01	1866330A1F71E2C6	
	PGM02	18662C920A072A30	
	PGM03	18662C920AF3A02E	

The contents of SYSPACKLIST after the conversion of plan FINTEST1 are shown in Figure 4-6.

Table 4-6 SYSPACKLIST after REBIND PLAN(FINTEST1) COLLID(*)

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST1	1	DSN_DEFAULT_COLLID_FINTEST1	PGM01
	2	DSN_DEFAULT_COLLID_FINTEST1	PGM02
	3	DSN_DEFAULT_COLLID_FINTEST1	PGM03

To convert plan FINTEST2 to a package-based plan, issue the REBIND command in Example 4-11.

Example 4-11 REBIND plan FINTEST2 with default collection

```
REBIND PLAN(FINTEST2) COLLID(*)
```

This converts DBRMs PGM01(T_2), PGM02(T_2) and PGM03(T_2) into the default collection DSN_DEFAULT_COLLID_FINTEST2.

The contents of SYSPACKAGE after the REBIND of PLAN FINTEST2 are shown in Table 4-7. Notice that the packages in collection DSN_DEFAULT_COLLID_FINTEST1 and DSN_DEFAULT_COLLID_FINTEST2 both have PGM01, PGM02, and PGM03, but they have different contokens. This shows that using the default collection ID allows the conversion to take place without overlaying any packages.

Table 4-7 SYSPACKAGE after REBIND PLAN(FINTEST2) COLLID(*)

COLLECTION	PACKAGE	CONTOKEN	VERSION
DSN_DEFAULT_COLLID_FINTEST1	PGM01	1866330A1F71E2C6	
	PGM02	18662C920A072A30	
	PGM03	18662C920AF3A02E	
DSN_DEFAULT_COLLID_FINTEST2	PGM01	18662C920BC8D185	
	PGM02	18662C920C96574E	
	PGM03	18662C920D849192	

The contents of SYSPACKLIST after both of the REBIND commands are shown in Table 4-8. This shows that both plan FINTEST1 and plan FINTEST2 are using the default collections.

Table 4-8 SYSPACKLIST after REBIND PLAN(FINTEST2) COLLID(*)

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST1	1	DSN_DEFAULT_COLLID_FINTEST1	PGM01
	2	DSN_DEFAULT_COLLID_FINTEST1	PGM02
	3	DSN_DEFAULT_COLLID_FINTEST1	PGM03
FINTEST2	1	DSN_DEFAULT_COLLID_FINTEST2	PGM01
	2	DSN_DEFAULT_COLLID_FINTEST2	PGM02
	3	DSN_DEFAULT_COLLID_FINTEST2	PGM03

4.5.2 Example 2: Converting one plan, named collection, DBRMs only

In this example, we bind plan FINTEST1 and then bind plan FINTEST2 using the same DBRMs that we used in example 1. The difference is that we convert both plans using the collection FIN_TEST_BATCH. When we convert plan FINTEST2, we overlay the packages created when we converted plan FINTEST1.

We begin by converting plan FINTEST1 issuing the REBIND command shown in Example 4-12.

Example 4-12 REBIND command using named collection FIN_TEST_BATCH

```
REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH)
```

This command executes properly if:

- PGM01(T_1), PGM02(T_1) and PGM03(T_1) are not packages in collection FIN_TEST_BATCH.

  or

- PGM01(T_1), PGM02(T_1) and PGM03(T_1) are packages in collection FIN_TEST_BATCH without VERSION populated.

Plan FINTEST1 might have execution time errors if:

- PGM01(T_1), PGM02(T_1) or PGM03(T_1) exist in collection FIN_TEST_BATCH without a VERSION. In this case, the packages are replaced by the REBIND process. The new packages have the CONTOKEN from (T_1). This could cause a -805 at execution time. We do not show this configuration.

Plan FINTEST1 has:

- PGM01(T_1), PGM02(T_1), and PGM03(T_1) bound into collection FIN_TEST_BATCH.

Plan FINTEST1 has a PKLIST of:

1. FIN_TEST_BATCH.PGM01
2. FIN_TEST_BATCH.PGM02
3. FIN_TEST_BATCH.PGM03

Attention: After the REBIND of plan FINTEST2 has completed, plan FINTEST1 might now have execution time -805 errors.

The contents of SYSPACKAGE after the REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH) are shown in Table 4-9.

Table 4-9 SYSPACKAGE after REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH)

COLLECTION	PACKAGE	CONTOKEN	VERSION
FIN_TEST_BATCH	PGM01	1866330A1F71E2C6	
	PGM02	18662C920A072A30	
	PGM03	18662C920AF3A02E	

The results of SYSPACKLIST after the REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH) are shown in Table 4-10.

Table 4-10 SYSPACKLIST after REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH)

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST1	1	FIN_TEST_BATCH	PGM01
	2	FIN_TEST_BATCH	PGM02
	3	FIN_TEST_BATCH	PGM03

REBIND PLAN(FINTEST2) COLLID(FIN_TEST_BATCH)

This command replaces the packages in collection FIN_TEST_BATCH with PGM01(T_2), PGM02(T_2) and PGM03(T_2). The replace of the package occurs because the program name and version numbers both match. The contents of SYSPACKAGE after the REBIND PLAN(FINTEST2) COLLID(FIN_TEST_BATCH) are listed in Table 4-11.

Attention: After the REBIND of PLAN2 has completed, the packages in collection FIN_TEST_BATCH now have the CONTOKENs from the T_2 copies of the programs as shown in Table 4-11. If you try to run plan FINTEST1, you receive -805 errors at execution time.

Table 4-11 SYSPACKAGE after REBIND PLAN(FINTEST2) COLLID(FIN_TEST_BATCH)

COLLECTION	PACKAGE	CONTOKEN	VERSION
FIN_TEST_BATCH	PGM01	18662C920BC8D185	
	PGM02	18662C920C96574E	
	PGM03	18662C920D849192	

The results of SYSPACKLIST after the REBIND PLAN(FINTEST2) COLLID(FIN_TEST_BATCH) are shown in Table 4-12.

Table 4-12 SYSPACKLIST after REBIND PLAN(FINTEST2) COLLID(FIN_TEST_BATCH)

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST1	1	FIN_TEST_BATCH	PGM01
	2	FIN_TEST_BATCH	PGM02
	3	FIN_TEST_BATCH	PGM03

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST2	1	FIN_TEST_BATCH	PGM01
	2	FIN_TEST_BATCH	PGM02
	3	FIN_TEST_BATCH	PGM03

4.5.3 Example 3: Converting one plan, named collection, DBRMs, packages

This example shows how the REBIND command converts plan FINTEST1 when the plan consists of both DBRMs: PGM01(T_1), PGM02(T_1), PGM03(T_1) and PKLIST(FIN_TEST_BATCH.*). This shows that the newly created packages are placed at the beginning of the PKLIST and the current PKLIST are placed after the new packages.

REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH)

PLAN1 executes properly if:

- PGM01(T_1), PGM02(T_1) and PGM03(T_1) are not packages in FIN_TEST_BATCH.

  or

- PGM01(T_1), PGM02(T_1) and PGM03(T_1) are packages in FIN_TEST_BATCH with different VERSIONs than in the (T_1) versions of the programs.

PLAN1 might have execution time errors after the conversion if:

- PGM01(T_1), PGM02(T_1) and PGM03(T_1) currently exist in collection FIN_TEST_BATCH. This could cause a -805 at execution time.

The DBRMs are the same as listed in Table 4-4 on page 60.

The contents of SYSPACKLIST before the REBIND are shown in Table 4-13.

Table 4-13 SYSPACKLIST before REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH)

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST1	1	FIN_TEST_BATCH	*

We then issue the REBIND command in Example 4-13.

Example 4-13 REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH)

```
REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH)
```

SYSPACKLIST now shows the converted plans in front of the original PKLIST, as shown in Table 4-14.

Table 4-14 SYSPACKLIST after REBIND PLAN(FINTEST1) COLLID(FIN_TEST_BATCH)

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST1	1	FIN_TEST_BATCH	PGM01
	2	FIN_TEST_BATCH	PGM02
	3	FIN_TEST_BATCH	PGM03
	4	FIN_TEST_BATCH	*

4.5.4 Example 4: Converting multiple plans, specific collection, DBRMs, packages

This example shows to use the REBIND command to convert two plans in one run. If you use the COLLID(*) option (not shown), you do not overlay the packages in SYSPACKAGE. By using the COLLID(FIN_TEST_BATCH) option, you overlay the packages created in the first plan conversion with the packages created by the second plan conversion.

- PLAN 1 contains DBRMs: PGM01(T_1), PGM02(T_1), PGM03(T_1) and PKLIST(FIN_TEST_BATCH.*)
- PLAN 2 contains DBRMs: PGM01(T_2), PGM02(T_2), PGM03(T_2) and PKLIST(FIN_TEST_BATCH.*)

REBIND PLAN(FINTEST1,FINTEST2) COLLID(FIN_TEST_BATCH)

Tip: If you misspell any PLAN name using this syntax, the BIND does not work and you get the message:

```
REBIND PLAN(FINTEST1,DFINTEST2) COLLID(FIN_TEST_BATCH)
INVALID PLAN IDENTIFIER, DFINTEST2
MISSING ONE OR MORE PLAN NAMES++
MISSING 1 ALPHABETIC + UP TO 7 ALPHANUM/NATIONAL CHARS
```

Before the conversion, SYSDBRM has the data shown in Table 4-4 on page 60.

SYSPACKLIST has the data shown in Table 4-15.

Table 4-15 SYSPACKLIST before REBIND PLAN(FINTEST1,FINTEST2) COLLID(FIN_TEST_BATCH)

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST1	1	FIN_TEST_BATCH	*
FINTEST2	1	FIN_TEST_BATCH	*

- PGM01(T_1), PGM02(T_1) and PGM03(T_1) are bound into collection FIN_TEST_BATCH.
- PGM01(T_2), PGM02(T_2) and PGM03(T_2) are bound into collection FIN_TEST_BATCH.

The contents of SYSPACKAGE after the conversion are shown in Table 4-16.

Table 4-16 SYSPACKAGE after REBIND PLAN(FINTEST1,FINTEST2) COLLID(FIN_TEST_BATCH)

COLLECTION	PACKAGE	CONTOKEN
FIN_TEST_BATCH	PGM01	18662C920BC8D185
	PGM02	18662C920C96574E
	PGM03	18662C920D849192

The contents of SYSPACKLIST after the conversion are shown in Table 4-17.

Table 4-17 SYSPACKLIST after REBIND PLAN(FINTEST1,FINTEST2) COLLID(FIN_TEST_BATCH)

PLAN	SEQNO	COLLECTION	PACKAGE
FINTEST1	1	FIN_TEST_BATCH	PGM01
	2	FIN_TEST_BATCH	PGM02
	3	FIN_TEST_BATCH	PGM03
	4	FIN_TEST_BATCH	*
FINTEST2	1	FIN_TEST_BATCH	PGM01
	2	FIN_TEST_BATCH	PGM02
	3	FIN_TEST_BATCH	PGM03
	4	FIN_TEST_BATCH	*

Attention: After the REBIND of plan FINTEST2 is completed, the packages in collection FIN_TEST_BATCH now have the contokens from the T_2 copies of the programs. Look at the contokens in Table 4-9 on page 63. If you try to run plan FINTEST1, you receive -805 errors at execution time.

Part 2

Using packages

In this part of the book, we discuss the development of packages. The primary audience for this part is the application developer and the application DBA.

We begin with a review of the steps in developing a package. Because the use of distributed applications has increased, we then cover the special considerations needed for developing distributed applications. We discuss some of the common problems you might run into. Some reference material, such as catalog tables, useful queries, and information about tools that can help you in your development, is also available in Part 4, “Appendixes” on page 257.

This part includes the following chapters:

5

Application development framework

Explanation of a basic approach to using DB2 packages for application development works best when a simple scenario is used to illustrate the basic techniques and alternatives. In this chapter, we work through the development and promotion process.

We cover the following topics:

- General naming guidelines
- Guidelines for grouping packages into collections
- Contrasting collection, version, and consistency CONTOKEN
- Basic authorizations for development and promotion
- A simple scenario for development and promotion
- Special registers that affect package execution
- Using collections to support access to mirror tables
- Using a try-out collection
- Using packages with routines
- Accessing data from other applications

Where applicable, we point to other chapters of the book that cover some material in more detail. For example, security and access path management are covered in detail in their own chapters.

5.1 General naming guidelines

Some basic naming guidelines are useful, regardless of how program promotion is managed. It is handy to be able to look at the name or qualifier of an object and see immediately what application it belongs to, where it should be used, and so on.

The naming standard for database objects (especially the values for DATABASE and CREATOR) should complement the names used for objects associated with packages. If you already have standards in place for databases and related objects, any naming convention that you develop for objects related to packages should work in harmony with those existing naming conventions.

The proposals that follow extend the sample database environment defined in Appendix D, “Sample test environment” on page 309.

Important: The naming guidelines proposed here are intended to provide a basic framework if you are building a new application and do not have any existing applications, or perhaps want to avoid a poor naming standard from another application.

Our suggestions are just that, and you should feel free to adapt or modify any of these suggestions so they make sense for your local environment. We go beyond the essential basic framework in order to show how the guidelines can fit together, and your organization structure or methodology for program promotion might dictate changes to the suggestions that follow.

Regardless of object types, here are some of the elements that you should consider in building your qualifiers or names:

- Short application code:

  It is common for an application to be assigned a 2-character or 3-character code. It is usually used as the first few characters in each qualifier (where the object has a 2-part name, such as table) or name (where the object has a 1-part name such as database or plan)

  For the sample environment, we have used the following 3-character codes:

  - “FIN” represents the financial application
  - “PAY” represents the payroll application.

- Application environment (development, test, QA, production, and so on):

  If each application environment were in a different DB2 subsystem, theoretically the qualifier could be the same everywhere. However, authorization is often granted by a shared group authorization ID and this would globalize access. Also, external objects, such as source code, need a different qualifier in data set names. Therefore, each application environment needs a code.

  A 4-character code used to identify the application environment is used in other parts of the book to more simply provide examples, such as DEVT, TEST, PROD. In this section, it is shortened to make more effective use of the limited number of characters sometimes available in qualifiers and names.

  - “D” represents the “Development” environment
  - “T” represents the “Test” environment
  - “P” represents the “Production” environment

- Runtime environment and instance (batch, CICS, routines, and so on):

  Plans, collections, and packages are executable, and it can be useful to distinguish between the environments they are running in. For example, if there is a performance problem with a currently-executing package, the plan name can help easily identify that the problem is with a batch job rather than an online transaction.

  Accounting and audit often start their data aggregation at the DB2 plan level. DB2's connectivity is geared to plans. Having many plans improves granularity of measurement, but requires more administration. Having too few does not provide any granularity of measurement.

  As collections can be easily associated with any plan via the PKLIST option, grouping of collections is flexible. Note that every package bound into a collection requires an individual bind, so binding the same package into multiple collections is generally avoided unless there is a good reason.

5.1.1 Plans

The maximum name for a plan is 8 bytes. For this reason, you need to have a strict naming standard for your plans. Extending the foregoing structure, we offer a starting point for discussion:

```
aaaerxxx
```

Where:

```
aaa    → application code: FIN, PAY, and so on)
e      → application environment: (D)evt, (T)est, (P)rod, and so on
rxxx   → runtime environment and instance (see the following notes)
```

The middle-ground we offer for the runtime environments for plans is one of many workable options:

- "CICS" represents a common CICS plan
- "Cxxx" represents a CICS-based plan used for 'interesting' transactions
- "BTCH" represents a common plan for batch jobs
- "Bxxx" represents plans defined for individual batch jobs
- "Sxxx" represents plans defined for specific server business processes

You might want individual plans for some transactions or batch jobs for various reasons:

- Long-running batch jobs can qualify because performance and tuning reports can focus in on one or more plans without picking up other jobs.
- Jobs with special charge back requirements can be accounted for separately.
- Transactions with known sporadic problems can qualify for special treatment.
- Key business processes invoked from a client can be grouped and measured independently of other business processes (depending on the mechanism used, you might or might not get to choose the name of the plan).

Some examples of how the recommendations can be applied:

```
FINPCICS  → Financial application, production CICS, general-purpose
FINTCINQ  → Financial application, test CICS, high-performance inquiries
PAYDBTCH  → Payroll application, development batch, general-purpose
PAYPBOFF  → Payroll application, production batch for off-site employees
PAYPSINQ  → Payroll application, production server business inquiry process
```

5.1.2 Collections

DB2 allows up to 128 bytes for a collection ID. Your software vendor's tools might place a restriction shorter than this. It is usually practical to use much shorter names.

Extending the foregoing structure, we offer a naming convention that has consistency:

```
aaa_eeee_xxxxx_yyyyy
```

Where:

```
aaa    → application code: FIN, PAY, and so on)
eeee   → application environment: DEVT, TEST, PROD, and so on
xxxxx  → runtime environment and instance
         (see notes in 5.1.1, "Plans" on page 71)
yyyyy  → any qualifier to the runtime environment (see the following notes)
```

The same general components detailed previously apply to collections. Additionally, some other groupings can be useful, depending on the requirements of the application:

- Isolation level:

  For example, if a few programs are bound with ISOLATION(UR), you might need to avoid confusing these packages with the more general ISOLATION(CS) packages

- Parallelism:

  For example, if parallelism, that is, DEGREE(ANY), is used only in special circumstances, you might need to avoid confusing these packages with the more general DEGREE(1) packages

- Special requirements:

  For example, a collection can be dedicated for testing the access paths of new or changed packages

Here are some examples of how the recommendations can be applied:

```
PAY_DEVT_BATCH     → Payroll application, development batch, general-purpose
PAY_PROD_ROUTINES  → Payroll application, production, Triggers and UDFs
FIN_PROD_CICS      → Financial application, production CICS packages
FIN_TEST_CICS      → Financial application, test CICS packages
FIN_PROD_CICS_UR   → Financial application, production CICS, ISOLATION(UR)
FIN_PROD_CICS_FALLBACK  → Financial, fallback for production CICS packages
```

5.1.3 Packages

For COBOL or PL/I programs, the package name is the program name, which can be up to 8 bytes long.

For packages created via the CREATE TRIGGER statement, the package name can be up to 128 bytes long, and is always the same as the trigger name.

5.1.4 Owners or creators

The owner or creator of objects is usually tied to a shared RACF® Group (or equivalent in other security products) or DB2 Role. With up to 8 bytes available, the combination of application code and application environment can be sufficient, and is used for our scenarios.

Here are some example of creators used for tables, schema qualifiers for routines, and so on:

```
PAYDEVT   → Payroll application, development environment
PAYPROD   → Payroll application, production environment
FINTEST   → Financial application, system test environment
```

5.2 Guidelines for grouping packages into collections

Naming standards and package grouping for collections should be established before any significant use of packages is made. Guidelines for the granting of bind and execute privileges for the various activities associated with application development should be included. Controlling the use of collections is an administrative task that is important to the successful implementation of packages.

The approaches available for using collections range from one collection per environment, such as production, to one collection per plan. The latter is not recommended because of the administrative and performance overheads. The manner in which packages are grouped into collections depends partly on the procedures used for the management of source code in the existing test and production environments, and partly on the general administrative and security levels important to the application.

The approaches for using collections include:

- One collection per entire development environment
- One collection per business area
- One collection per application environment
- One collection per plan
- Special-purpose collections

Figure 5-1 highlights the pros and cons of different approaches to using collections.

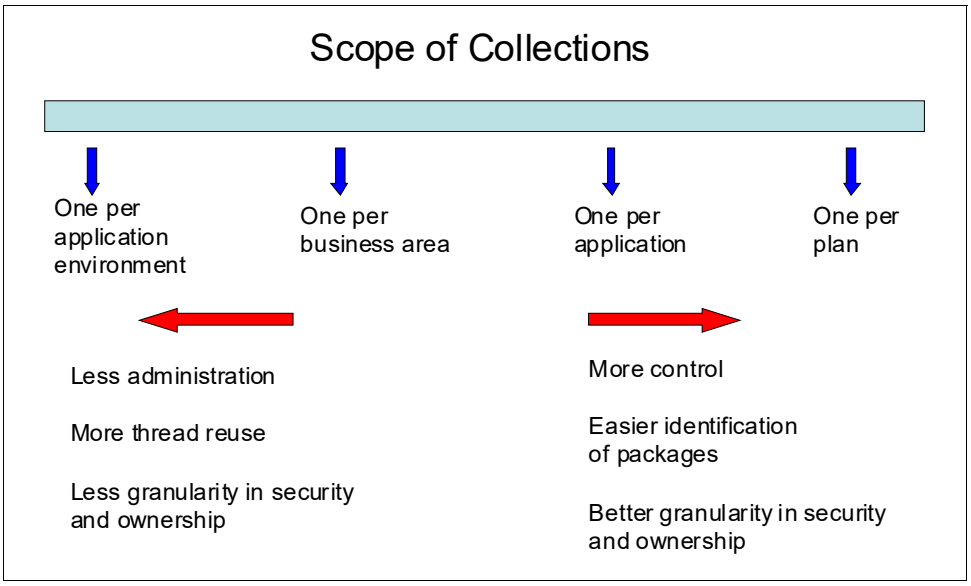

Figure 5-1 Approaches to using collections

5.2.1 One collection per entire development environment

Using one collection for all of the packages in a development environment offers great simplicity in terms of organizing packages. For example, a production DB2 subsystem has a single collection called PROD_CICS into which all CICS program packages are bound. The plans used by the CICS transactions are all bound with PKLIST(PROD_CICS.*).

One advantage of this arrangement is that a small number of plans could be used by many transactions, facilitating broader thread reuse. If you consider this scenario, the RELEASE option used on the BIND PACKAGE command determines when resource will be released. This is discussed in more detail in 11.1, “Binding parameters that impact performance” on page 224

Note that different applications can use different plans, even if they all have the same PKLIST definition.

This approach must be carefully assessed with respect to other requirements, however. One disadvantage would be that when all of the packages are bundled in a big collection, granularity of package security and ownership is not obvious. Also, vendor packages could conflict with in house package names.

Depending on how many plans are used and how packages are used within these plans, it can be harder to interpret reports of package-level performance.

You should examine and test this approach carefully in your own environment before considering it.

5.2.2 One collection per business area

Using one collection for each business area, such as Finance or Human Resources, would involve having a number of related applications in one collection.

This is very similar to the option described in 5.2.1, “One collection per entire development environment”, except with only a slightly more granular set of packages.

5.2.3 One collection per application environment

One collection per application environment provides greater security and administrative control as it allows collection, package, and plan privileges to be limited to the application group. The use of wild carding in collection names for collection privileges and package lists retains authorizations within an application boundary.

This approach also allows easy identification of the packages involved in the application, and clearer boundaries when an application needs to use packages from another application. See 5.10.3, “Using collections to access another application’s functionality” on page 98 for more information on this.

Because CICS and batch programs are typically compiled and linked differently, as are programs that use routines or other connections, each application “environment” can be interpreted as the runtime environment for a specific promotion level, and requires its own collection. For example, the Financial application (known as FIN in our scenario) in the development environment (known as DEVT for some of our examples) could require many collections: FIN_DEVT_BATCH, FIN_DEVT_CICS, and FIN_DEVT_ROUTINES.

This kind of separation is necessary if programs are compiled or linked for different environments. If all programs are compiled and linked the same way, or the program naming conventions are managed carefully enough there is no program used in multiple environments, it is possible to either consolidate the collection to a simple name such as FIN_DEVT, or to put the programs that are in shared between environments in a collection with a name such as FIN_DEVT_SHARED (or FIN_DEVT_COMMON). Note again that these suggestions need tailoring to your environment and are not rigorous recommendations.

This method, with or without the inclusion of runtime environment or qualifier to the runtime environment, is useful when promoting applications between the various test and production environments or in identifying the packages that should be rebound where there has been a significant change to the DB2 environment for a particular application. For instance when a RUNSTATS has just been run on all production Payroll table spaces and indexes.

Having a collection per application environment does not limit the number of plans that can be used.

A final example of where this approach has strengths is where one code base is used against multiple database environments. In early development environments, sometimes developers have their own database (or a subset of the database), or multiple testing teams have their own database for different kinds of testing against a set of application code. The approach proposed here allows for a package to be easily bound and identified as belonging to a specific application environment.

Another advantage of the application grouping is the ability to easily identify all the packages, for instance, in a Payroll collection when using a tool to compare “before” and “after” access paths after DB2 maintenance is applied.

5.2.4 One collection per plan

Because collections tend to reflect the compile and link process, there is no reason to consider providing each plan with its own collection. Any program shared between multiple load modules could end up in different plans, with the management overhead that this would entail. Each shared program would need to be bound to many collections, and changes to program hierarchies would need to be reflected in package binds.

5.2.5 Special-purpose collections

Collections can be used for special-purpose grouping of programs.

One example would be a collection used to provide fallback in production where a program modification fails. This might or might not be necessary, depending on how DB2 is selecting, at runtime, which version of the package will be executed. Where a blank value is used for the VERSION option, fallback management is usually considered to be essential and discussed in detail in 5.3.2, “Choosing between the Blank VERSION or VERSION(AUTO) methods” on page 77.

Another example would be a collection used to try out the bind process (especially to confirm suitable access paths) for a new or updated package. A package can be bound to a collection that is not included in any plan. This is discussed in 5.8, “Using a try-out collection” on page 94.

An earlier example already provided is where programs can be shared between multiple environments: callable by CICS transactions or batch jobs.

5.3 Contrasting collection, version, and consistency CONTOKEN

In this section we discuss the differences among collection, version, and consistency CONTOKEN, how they affect the way DB2 chooses a package at runtime, and how they impact the choices we make regarding program promotion.

5.3.1 Understanding VERSION versus CONTOKEN

One of the very first things you need to know about developing using packages is what *versions* and *consistency tokens* (appearing as CONTOKEN or DCONTOKEN in the DB2 catalog) are used for, and how they interact. Understanding what they are used for is *key* to understanding how the bind process chooses a package to add or replace, and how DB2 selects a package at execution time.

Two unique keys exist on SYSIBM.SYSPACKAGE. Both include the LOCATION name, which is always set to blank by DB2, so we ignore it for this discussion and show only the relevant columns of the two unique indexes:

SYSPACKAGE index DSNKKX01:

1. COLLID
2. NAME
3. VERSION

SYSPACKAGE index DSNKKX02:

1. COLLID
2. NAME
3. CONTOKEN

It is not obvious, but for a given COLLID and NAME combination, both the VERSION and CONTOKEN *must* be unique. There are implications that flow from this:

- When you bind a program, the VERSION is important.
- When you execute a program, the CONTOKEN is important.

When you BIND a program

You always bind a program to a specific collection (which corresponds to the COLLID and NAME values in the foregoing indexes), and DB2 uses the VERSION value in the DBRM to decide which existing package is added or replaced.

Note: There is no VERSION parameter on BIND PACKAGE... MEMBER because the value comes *only* from the DBRM.

For discussion of the implications of using the REPLVER option, refer to 2.1.2, “ACTION (and REPLVER and RETAIN) in BIND PACKAGE and BIND PLAN” on page 18.

For a given version of a program, only one ‘copy’ can exist at a time within a collection. A bind will replace an existing package with the same version. VERSION(AUTO) creates a unique version, avoiding the replace. The consistency token is not part of the decision-making process during bind processing, but it is stored in the DB2 catalog for use during program execution.

In summary, the VERSION value is used to determine which package is to be added or replaced.

When you EXECUTE a program

When a program is executed, DB2 uses PKLIST entries from the plan to determine the valid list of collections to search for the package, and whether execution is allowed at this specific location.

DB2 tries one entry in PKLIST after another looking for a match, plugging in COLLID and NAME values. It uses the consistency token contained in the code for the final part of the key to find a matching package. The consistency token match checks that the SQL in the program matches the SQL in the package. A more detailed explanation of the overall search process is given in 3.2, “Package list search ” on page 41.

In summary, during execution, the consistency token is used to look for a matching package, and the VERSION value is ignored.

5.3.2 Choosing between the Blank VERSION or VERSION(AUTO) methods

As detailed previously, the choice of package at execution time is based on consistency tokens, but the bind process depends on versions, and this creates a subtle distinction worth examining carefully because the overall program promotion process depends on understanding the difference between two approaches:

- If DB2 has only a *single* version of a package in each collection, but the same package name is found in many collections, the consistency token in the program’s load module decides from which collection the package is retrieved.
- If DB2 has *multiple* versions of a package in a single collection, the consistency token in the program’s load module decides which of the versions in the collection is used.

Because packages were introduced with DB2 V2.3, two distinct paths for developing programs and promoting changes through application environments have generally been used (a third, less common, alternative is discussed in “Setting specific VERSION values” on page 79):

- The “Blank VERSION” method:

  During a compile/link/bind process, the VERSION value can be allowed to default to blank during the precompile or coprocessor step. In the bind step, a package is bound to a specific Collection, and always replaces an old version (if one exists). This happens in the same way that the link step replaces any existing load module in the load library — the matching bind process replaces any existing package.

  Application environments often allow for multiple code levels. This is intended to group the current production code, allow for an option to temporarily override the production code on an emergency basis, and some mechanism for falling back to an old version of the code if something goes wrong.

  In the same way that a STEPLIB can be used in batch jobs to support access to multiple levels of code, the PKLIST for a plan can contain references to multiple Collections. In this way, whichever library a load module executes from, DB2 finds a matching package ready for execution:

  – An obvious reason to use this approach is that the general concept of “STEPLIB” is matched by the DB2 concept of package list or current package path and replacing/moving code is matched by BIND and BIND... COPY processes. Whichever version of the program is executed, no-one gets SQLCODE -805 in any application environment.

- An obvious problem with this approach is that the process of migration of code must be *exactly* matched by BIND and BIND... COPY processes — any deviation can result in SQLCODE -805.

- The “VERSION(AUTO)” method:

  Using VERSION(AUTO) during the precompile/coprocessor process, multiple versions of a program can exist with the same COLLID and NAME combination. The sequence of concatenation of load libraries determines which program executes and DB2 always finds a matching package ready for execution amongst the many that are available:

  - A good reason to use this approach is that many versions of a program can be easily managed without getting SQLCODE -805 in any application environment.
  - An obvious problem with this approach is that multiple versions in the DB2 Catalog are not needed forever, and a process needs to be executed to figure out which package versions can be removed because they are no longer needed.

  Note: If some developers have their own load library, but are sharing a database, VERSION(AUTO) can allow each to make use of their own versions of packages without replacing package used by other developers.

Recommendations

Each development team needs to decide which approach works best for their organization. Sometimes the decision is driven by an external factor, such as the use of distributed routines (stored procedures) that have different processes for managing code versions. Sometimes the decision is driven by mandated use of a tool to manage code changes and the subsequent migration of load modules.

With small-scale environments (small to medium databases, low to medium transaction rate) and tight control over source code promotion, the use of the “COLLECTION method” means that each bind of a program to a collection automatically removes the previous version of it (just as the link overwrites previous version of the code), and each collection is mirrored by a load library and/or database environment. As code is moved into the production environment, some kind of fallback collection is used to create a back out option if it is needed.

Because users are already familiar with the concepts of load library management and promotion, the “COLLECTION” method is workable for smaller and/or simpler environments.

As volumes and transaction rates increase, problems emerge:

- The process for creating a fallback copy of a package invokes the optimization process, and an access path change resulting from this package copy process could cause a critical performance problem if the fallback package is needed.
- The act of promoting into production requires the existing production version of the package to be *replaced*. At high transaction rates, the package might be almost constantly in use and the replacement process could cause unacceptable locking issues.
- If code is being changed frequently, the number of levels of fallback required could exceed the number of fallback collections that are created. Also, managing more than a single fallback collection creates procedural complexities.

We recommend that the “VERSION(AUTO) method” be used for the highest levels of granularity in control over packages, but we let you decide where the line is drawn between the two approaches.

To complicate things further, we offer a third alternative that has special applicability, as described in the following sections.

Setting specific VERSION values

So far, we have bypassed discussion of another key approach to managing packages: using specific version values, for example V4R9 or V490.

This approach is typically used in applications that are focused on software releases, with the VERSION value tied to the software release, and fixes to the version tagged with same version value, for example V490_FIX22101 might be the VERSION value for a fix to the V490 code.

It is assumed in this approach that all programs are recompiled for a new 'version' of the software, and interim releases or fixes use new package versions that co-exist with the existing software version rollout. If this assumption is false, over time it can become difficult to determine which are 'live' versions of packages.

This approach is a very powerful way of managing release-based software products because it avoids SQLCODE -805 errors, allows for simple roll-out of new software versions without impact to current software, and provides for easy clean-up of old packages after a new software version has been implemented.

It should be used by all release-based software product vendors.

5.4 Basic authorizations for development and promotion

To configure a basic development environment, a minimal set of authorizations need to exist. The sample environment detailed in Appendix D, "Sample test environment" on page 309 is used here to illustrate how a basic set of GRANT statements can be used to get started.

A real-world environment typically requires more granular authorization and the suggestions that follow are just that. The hierarchy of authorities that follow define only the DB2 security requirements. After a privilege has been delegated to the DBA team, or the DBA team has further delegated the privilege to another team, the "parent" is usually not further involved. It is important to note that the hierarchy of DB2 authorizations does *not* represent organizational structures. Instead the hierarchy encapsulates security at each level so that subsequent organizational changes are reflected as minimally as possible within DB2.

For a detailed examination of the security requirements associated with package usage, refer to Chapter 9, "Security considerations" on page 173.

It is assumed that normal DB2 security is being used. A role operating within a trusted context is also a valid target for authorization, but the purpose of this chapter is to provide a simple scenario for illustration of the concepts.

5.4.1 BINDNV in DSNZPARM

We recommend that the DSNZPARM parameter BINDNV (Bind New Version) is set to BIND. This value means that the creation of a new package does not require the BINDADD privilege, only the BIND privilege.

With BINDNV=BINDADD, the global BINDADD privilege that needs to be granted allows someone to create new plans as well as packages. It is not important whether or not broad access is needed to BINDADD (for plans) because that privilege can be granted independently if it is required. The issue is that BINDADD is "overloaded" and granting package privileges provides no options but to also grant plans privileges.

With BINDNV=BIND, PACKADM, CREATE IN, and BIND, privileges can be independently granted or combined to provide granular access to package-level binds without the need to allow additional plans to be created. This setting allows the BINDADD to be independently granted from package privileges.

This is a trade-off option for users who want the security with only BIND versus those who want BINDADD.

5.4.2 Initial delegation from SYSADM to DBA team

The DB2 SYSADM delegates authority to various objects in each application environment to the DBA team so that the DBA team can become responsible for a database environment, including the plans, collections and packages that will access the database.

The DBA team can be represented by:

- A shared primary authorization ID (with consequent lack of accountability)
- A group authorization ID (for example, a RACF group that each DBA's user ID is connected to)
- Participation in a DB2-defined “Role”

For details on how roles are analogous to shared authorization IDs, but add accountability, see 9.6, “Packages using roles in a trusted context” on page 188.

For illustrative purposes, only authorization to the initial FINDEVT environment is detailed. In this case, “FIN” represents the mnemonic code for the Financial application, and “DEVT” represents the mnemonic code for the Development environment.

Plans

Many organizations restrict general access to the broad BINDADD privilege to avoid the proliferation of plans. When a new development environment is created, the SYSADM delegates BINDADD to the DBA authorization ID or role.

BINDADD is given by SYSADM only to the DBA team. By omitting the WITH GRANT OPTION clause, proliferation of plans can be avoided.

Note that subsequent REBIND PLAN commands used by the DBA team (or delegated to other teams) do not need the BINDADD privilege, only the BIND privilege on the plan.

Collections and packages

There is no explicit DB2 definition for the object “Collection.” Unlike a database or table, its existence is defined only in terms of authorization. The delegation by SYSADM to the DBA authorization ID or role allows the DBA to make use of the collection.

In almost every case, the DBA team further delegates privileges, and the responsibility to use those privileges, to other teams. For example, developers are given access to bind programs and do not rely on the DBA team to bind packages, and another team is typically responsible for migrating programs between development environments.

SYSADM gives the DBA team complete control over any collections need for the development environment via the PACKADM privilege. Use of the WITH GRANT OPTION clause allows for further delegation as needed.

Access could be granted to the test environment at the same time. After the development environment, we assume that BINDAGENT is used to allow packages to be bound and/or copied, so the WITH GRANT OPTION clause is not needed.

Schema privileges for routines

The creation and maintenance of routines such as stored procedures and user-defined functions requires all of the schema privileges (CREATEIN, ALTERIN, DROPIN):

```
SET CURRENT SQLID = 'some-sysadm-authid';
GRANT CREATEIN,ALTERIN,DROPIN ON SCHEMA FINDEVT TO FINDEVT WITH GRANT OPTION;
```

Where the second FINDEVT here is a user ID or RACF group or role.

Database privileges (DBADM)

Access to the application database is outside the scope of this book, and included only to allow a complete picture of basic authorizations.

DBADM should already be delegated to the DBA authorization ID or role using the WITH GRANT OPTION clause to give the DBA complete control of the application database.

It is useful that development DBAs have equivalent privileges (in the development environment) to the privileges held by production DBAs in the production environment. For example, issues can arise if DDL scripts are run in development via a SYSADM privilege and same scripts are expected to run in production via DBADM privilege.

Non-database privileges

Access to the application database is outside the scope of this book, and included only to allow a complete picture of basic authorizations.

The actual list of system privileges needed by the DBA team is site-dependent, but can include one or more privileges from this list:

- USE OF BUFFERPOOL
- USE OF STOGROUP
- CREATEALIAS
- CREATETMTAB
- DEBUGSESSION
- DISPLAY
- MONITOR1
- MONITOR2
- STOSPACE
- TRACE

Some of these privileges would be further delegated to other teams, and some would be retained for use by DBAs under controlled conditions.

5.4.3 Delegation from DBADM to developers

Developers write code for external programs and routines. DBAs usually provide support for schema changes. Authorization is scoped appropriately.

Plans

In our defined scenario, DBAs create and manage the plans:

- For developers to be able to execute batch jobs, the EXECUTE privilege is needed on any plans used in the development environment(s).
- For developers to execute CICS workloads, typically authorization is provided to the CICS region rather than the individual executing the code. This allows the CICS region to execute the plans defined for use by CICS. CICS can be granted execute authorization in different ways, using CICS RDO definitions that are usually maintained by CICS systems programmers.

Collections and packages

At the least, a pair of privileges is needed:

- To be able to create new packages in a collection, the CREATE IN privilege is required.
- For developers to be able to issue BIND PACKAGE commands as part of the compilation process, the BIND privilege is needed on packages in the collections used in the development environment(s).

The foregoing privileges are not useful unless either paired, or grouped as the broader PACKADM privilege.

For the bind to succeed, a further privilege is required: access to the tables used by the program. In this scenario, we assume the shared developer authorization ID (FINDEVTD) is used as the OWNER value for the packages (with a QUALIFIER value of FINDEVT). This allows the development team to own and manage the packages, yet point to the DBA-controlled tables/views using unqualified SQL.

As we get close to a production environment, security is more rigid, but PACKADM could be suitable for our simple scenario.

In our scenario, DBADM delegates PACKADM for each of the collections to an authorization ID shared by the development team.

Schema privileges for routines

In our scenario, DBAs create the definitions for stored procedures, user-defined functions, triggers, and so on. Developers code the programs and manage the packages.

After being instantiated in the DB2 database using a CREATE FUNCTION or CREATE PROCEDURE statement, the authorization to execute Procedures, Functions, or Specific Functions is given via SQL using the GRANT EXECUTE statement.

The authority to copy routines or migrate routines to other development environments is not needed for a basic development security structure.

Note that when a trigger is instantiated into the DB2 database using a CREATE TRIGGER statement, the package is automatically created. Authorization to execute the trigger is not required because it becomes part of the database schema itself.

Native SQL procedures also automatically create packages, and authority is only needed to migrate (deploy) them to another collection or to a remote DB2 subsystem.

Authorizations to database objects

When developers have access to bind packages into a collection, they need privileges on the database objects to be allowed to use the OWNER option of BIND PACKAGE successfully.

Access to the application database is outside the scope of this book, and included only to allow a complete picture of basic authorizations.

Database object authorizations are outside the scope of this book, but the general concept is for the DBA team (FINDEVT) to grant SELECT / INSERT / UPDATE / DELETE authority to tables and views to the development team (FINDEVTD). How broad this authorization is, or how it might be segmented to sub-teams, is dependent upon the security requirements for the application.

Authorizations to non-database objects

Access to the application database is outside the scope of this book, and included only to allow a complete picture of basic authorizations.

For developers to manage data in tables in a development environment, additional privileges beyond the basic table/view privileges are generally required. Depending on the tools in use, one or more privileges from this list might be needed:

- DISPLAYDB
- IMAGCOPY
- LOAD
- RECOVERDB
- REORG
- STARTDB
- STATS
- STOPDB

Additional privileges are needed if additional databases are set aside for developers to create copies of tables, save test data in tables, and so on

5.4.4 Delegation from DBADM to a migration team

In our fictional scenario, a single team is charged with the responsibility of copying packages from one environment to another. The purpose of this inclusion is to show how basic authorizations for collection and package management can be delegated independently of the authorizations needed for development.

This could be done via a compile/link/bind process in early application environments, and a BIND PACKAGE.... COPY process for the transition into the production environment.

We assume that this team has no access to data.

Plans

In our scenario, DBAs create and manage the plans.

Collections and packages

As the migration team is not given access to underlying data, a direct grant of the PACKADM collection privilege or the package BIND privilege is not useful. The bind process will fail because the migration does not have the required table/view authorities, and because the migration team would not have access to the authorization ID shared by the DBA team.

Instead, this team needs the BINDAGENT privilege, granted by the DBA team. This allows the migration team to specify FINTEST or FINPROD as an owner in binds without posessing all of the other privileges this implies.

5.4.5 Delegation from DBADM to a production control team

In our scenario, once the application is in a production environment, the packages have already been bound by the migration team. The team that controls production execution does not need any specific package privileges at all because these are incorporated into the plans themselves.

The EXECUTE privilege on a plan is sufficient for batch jobs that do not make use of DB2 Utilities.

To use the LOAD, REORG, UNLOAD and other utilities, DBA-related privileges are required that are outside the scope of this book, or the job needs to execute using a DBA user ID.

5.5 A simple scenario for development and promotion

In this section we walk through a basic development scenario using the naming conventions and basic authorizations already developed earlier in this chapter. Where relevant, we refer to both the “Blank VERSION” method and the “VERSION(AUTO)” method detailed in 5.3.2, “Choosing between the Blank VERSION or VERSION(AUTO) methods” on page 77.

5.5.1 Getting started in the development environment (DEVT)

In our scenario, developers use either FIN_DEVT_BATCH or FIN_DEVT_CICS collections to create and update packages as unit testing takes place of new program code. (Routines and programs shared between CICS and batch environments are set aside to simplify the examples).

Important: The essential element of this scenario is that an ID with DBADM authority has delegated authorization to the development team (identified by a shared authorization ID FINDEVTD) to own and manage the packages.

Example 5-1 shows a sample BIND PLAN fragment, and the use of collections. If DBADM has bound the plan, the BIND and EXECUTE privileges need to be delegated to developers.

Example 5-1 BIND PLAN fragment for the development environment

```
BIND PLAN(FINDEVTC) PKLIST(FIN_DEVT_CICS.*) ...
```

Binding the DB2 plan with “COLLECTION.*” eliminates the need to bind replace the plan every time a new package is added to the collection which would be needed if the packages were explicitly listed. Also, dropping a package from the collection does not invalidate the plan. This is preferred from an administration viewpoint, however there is a small performance overhead for the administrative convenience.

Developers perform package binds after making sure that the DBRMLIB DD name has been assigned to a development DBRM data set. Example 5-2 shows a BIND PACKAGE fragment that shows the difference between OWNER and QUALIFIER values:

Example 5-2 Sample BIND PACKAGE fragment for an application program

```
BIND PACKAGE (FIN_DEVT_CICS) OWNER(FINDEVTD) MEMBER(ACCNTPGM) QUALIFIER(FINDEVT)
...
```

After some effort, our sample code is ready for promotion.

If a blank VERSION value has been used, there is only a version of each program ready for promotion. For each bind, only one version of any package can be kept. The programmer has to bind replace the existing package every time there is a modification. This is acceptable in the development environment.

If VERSION(AUTO) has been used, there could be many versions of each program, but the most recent version is probably the one intended for promotion. Because VERSION(AUTO) can result in too many versions in a development environment as programmers compile

multiples times per day, some teams choose to avoid VERSION(AUTO) in the initial development environment. If VERSION(AUTO) has been a used, and old package versions are not removed, a list of programs and matching package versions is needed as input to the promotion process.

5.5.2 Promoting from development (DEVT) to system test (TEST)

The defined scenario does not gives developers access to the packages in the system test environment, so the promotion process is managed by the migration team. (As mentioned earlier, this is not intended to reflect an entirely realistic environment, but to illustrate the concepts of managing packages through application environments).

The migration team, for the purposes of this discussion, uses a fresh compile and link process to promote the code into the system test environment. As the consistency token has changed for the load modules, there is no option to copy packages forward from the development environment, the packages are bound from the DBRMs created during the compile process.

The ID with DBADM privilege has bound the plans used for the testing. Example 5-3 shows a BIND PLAN fragment for a CICS system test plan.

Example 5-3 BIND PLAN fragment or the system test environment

```
BIND PLAN(FINTESTC) PKLIST(FIN_TEST_CICS.*) -
     OWNER(FINTEST) QUALIFIER(FINTEST) ...
```

DBADM has not delegated access to the FINTEST group to *own* the packages, but has provided the BINDAGENT privilege so that the testing team can manage packages in the system test environment. The system test team performs package binds as shown in Example 5-4.

Example 5-4 Sample BIND PACKAGE fragment for an application program

```
BIND PACKAGE (FIN_TEST_CICS) OWNER(FINTEST) MEMBER(ACCNTPGM) QUALIFIER(FINTEST)
...
```

This time, for both plans and packages, the OWNER and QUALIFIER clauses are identical because ownership has switched to the DBADM team.

In our scenario, a group of testers work day and night on validating the quality of the code. When a problem is discovered, a developer uses the development environment to code a fix for the problem and tests a solution in that environment. After the problem is corrected, a new version of the program is created in the system test environment using the same process that promoted the bulk of the code earlier.

Eventually the testing team declares the code ready for promotion into the production environment.

If a blank VERSION value has been used, there is only version of each program available for promotion to the production environment.

If VERSION(AUTO) has been used, and some programs have been updated multiple times to fix errors, there might be multiple versions in the system test environment. Before migration into the production environment, unneeded versions need to be cleaned up, or the process for package promotion needs to identify which versions of packages accompany code into the production environment.

5.5.3 Promoting from system test (TEST) to production (PROD)

In this simplified scenario, the promotion process is again managed by the migration team.

This is not intended to reflect a very realistic environment, but to illustrate the concepts of managing packages. Further, the concept of “team” should be clarified. In large organizations, individual teams of people can perform the different roles in this scenario. Smaller applications can use the same authorization structure, but an individual can participate in multiple roles: for example, they could be a tester and also be responsible for migration. Understanding the privileges related to package management allows for flexible assignment of individuals to DB2 roles.

The migration team, to provide a contrast to the approach used to promote into system test, does not compile and link for the move to production. Instead, code is moved to production, and binds are made for each of the modules moved.

Promotion to a production environment using a blank VERSION value

If a blank VERSION value has been used, the act of binding to the production environment replaces existing packages. Organizations typically require an option to fall back if something goes wrong with new code, so before packages are replaced during the promotion process, backups are required for existing load modules and packages. This is where the concept of a fallback collection is derived from. Packages are copied from a production collection to a fallback collection in case they are needed.

The process in detail is as follows:

1. When a blank VERSION value is used, before anything is promoted, a backup of the current production package is required. Because a package name must be unique within a collection, the fallback (or backup) package has to be copied into a “fallback” collection. Figure 5-2 shows the association between the program load libraries and the collections used. The FIN_PROD_CICS collection stores the current packages used by the production load library. The FIN_PROD_CICS_FALLBACK collection contains a backup copy of the current package. This backup is created by copying the package from the current collection before it is replaced during program promotion.

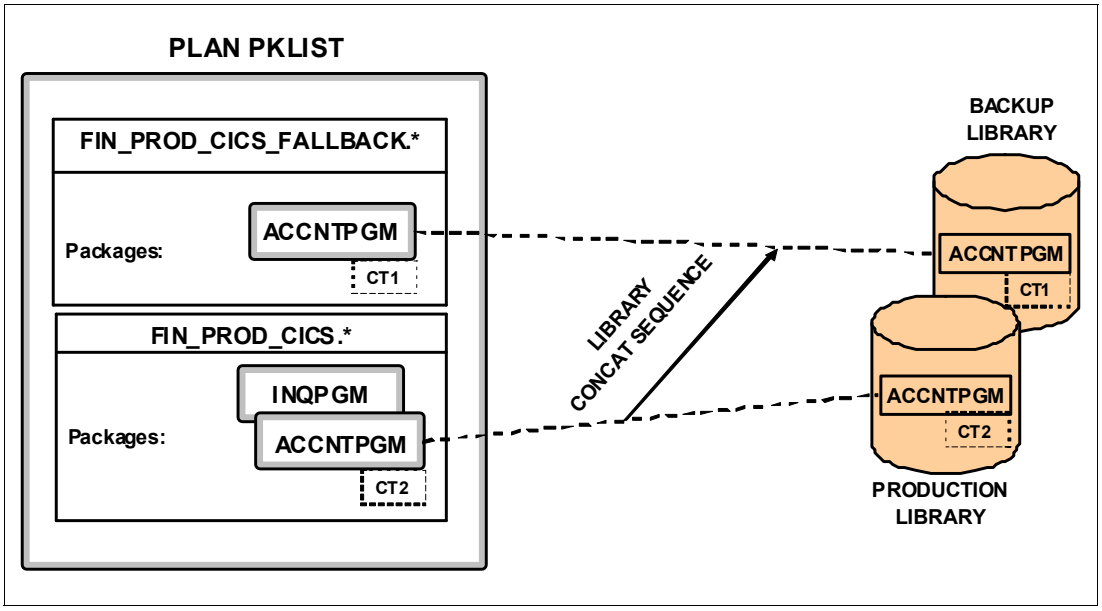

Figure 5-2 Current and backup versions using collections

Using collections to store multiple versions of a package mirrors the method of managing different versions of load libraries. Figure 5-2 shows how the production and backup libraries are organized uses the ACCNTPGM program as an example, which has consistency token CT1 originally, and now has consistency token CT2. The backup library is concatenated after the production library. Hence, the search sequence of a load module is the production library followed by the backup library.

The plan is bound with both the current package and the fallback package in the PKLIST sequence. By coding the fallback collection in the plan, the plan can have access to the backup packages if required. In a fallback, this arrangement allows instantaneous access to the backup package without rebinding the plan or packages. Example 5-5 shows a BIND PLAN fragment that allows for fallback in a production environment.

Example 5-5 BIND PLAN fragment for the production environment

```
BIND PLAN(FINPRODC) PKLIST(FIN_PROD_CICS.*, FIN_PROD_CICS_FALLBACK.*)
```

The package can be bound using the DBRM, or copied from the system test environment using the COPY option of the BIND PACKAGE command. Example 5-6 shows how to create a fallback package, using ACCNTPGM as an example.

Example 5-6 BIND PACKAGE fragment for creating a fallback package

```
BIND PACKAGE (FIN_PROD_CICS_FALLBACK) -
     COPY(FIN_PROD_CICS.ACCNTPGM) -
     OWNER(FINPROD) QUALIFIER(FINPROD) ...
```

Note: The administrator should check the access path of the backup package. It might have changed. Historical EXPLAIN output or equivalent information should be kept for such purposes.

2. After fallback copies are made of packages, the current load modules are also copied to a backup library.
3. Before moving any new load modules into the production environment, the bind process is executed, otherwise if a process were to start running immediately, it would fail with SQLCODE -805.

   A package that is being promoted can be bound using the DBRM or it can copied from the system test environment using the COPY option of the BIND PACKAGE command.

   The promotion of packages with a blank VERSION value can be managed by a bind that uses the promoted DBRM as input, or by copying the package from the system test environment as shown in Example 5-7 (you need to be connected to the test environment to execute this bind).

Example 5-7 BIND PACKAGE fragment for copying a package to the production environment

```
BIND PACKAGE (PRODDB2.FIN_PROD_CICS) -
     COPY(FIN_TEST_CICS.ACCNTPGM) -
     OWNER(FINPROD) QUALIFIER(FINPROD) -
```

Note: The access path of a new production package, replacing an existing one, should always be verified by checking the EXPLAIN output and comparing it with the previous access path.

4. After the new packages have been bound into the production environment, the load modules can be copied.

Figure 5-3 summarizes the setup and procedures to promote a change in program from development to system test and finally to production when using a blank VERSION value.

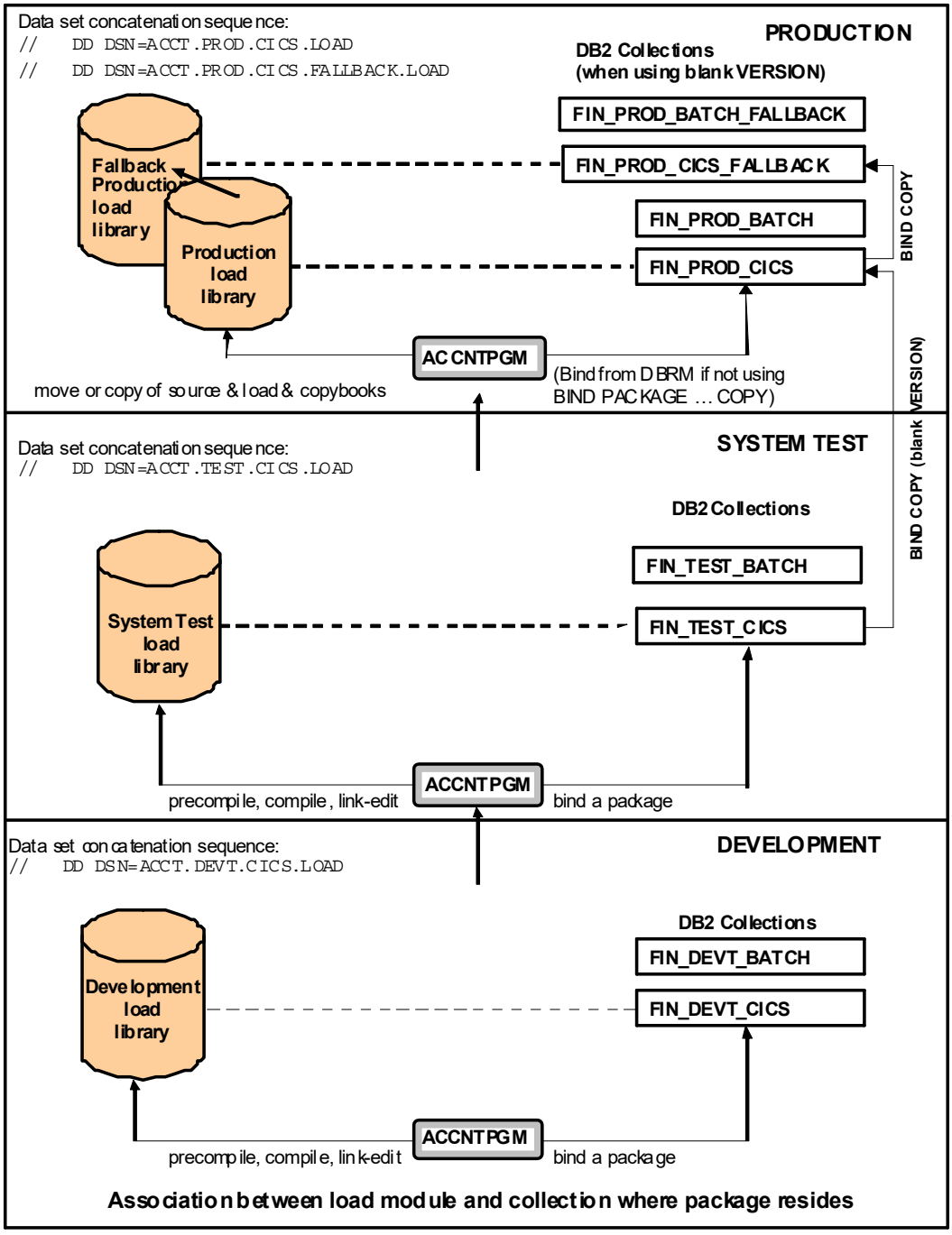

Figure 5-3 Complete promotion path for ACCNTPGM CICS program with blank VERSION value

Promotion to a production environment using VERSION(AUTO)

If VERSION(AUTO) has been used, introducing packages into a production environment does not replace any existing packages, so the process is simpler than that required when using a blank VERSION value.

Fallback is intrinsically possible because taking fallback copies of the load modules means that it's possible to continue to execute the current versions of the packages if required. Old load modules will automatically pick up earlier versions of packages in the same collection if they are executed.

The process is straight-forward:

1. Packages are promoted first. The simplest approach is to bind using the promoted DBRM as input. This automatically picks up the correct VERSION value from the DBRM.

   If packages are copied from the test environment using the COPY option of BIND PACKAGE, the COPYVER option also needs to be specified. The effort required to determine this value for each BIND PACKAGE command might not be justified.

   Note: The access path of a new production package, replacing an existing one, should always be verified by checking the EXPLAIN output and comparing it with the previous access path.

2. The current load modules are copied to a backup library.
3. The new load modules can be promoted to the production library.

Figure 5-4 summarizes the final promotion process into the production environment when using VERSION(AUTO).

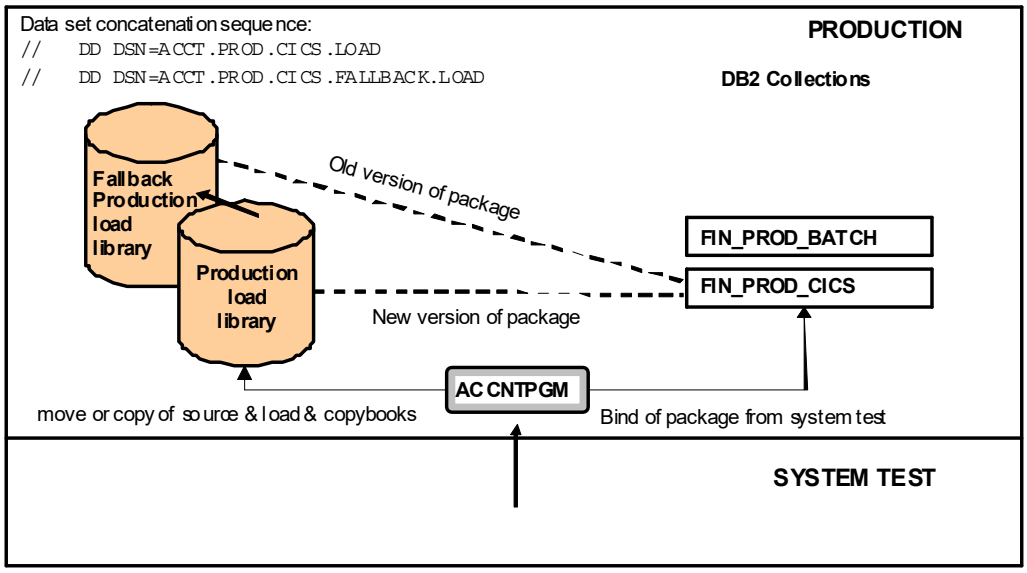

Figure 5-4 Promoting into the production environment using VERSION(AUTO)

5.5.4 Fallback to a previous version of a production program

If the new version of a program fails for any reason, fallback to the old version is easy and fast. For this example, we assume the ACCNTPGM program has an error.

Figure 5-5 shows what happens at fallback:

1. The ACCNTPGM module is either deleted from the current library or renamed, depending on local conventions.

2. There are no further DB2 requirements. The next invocation of ACCNTPGM finds and executes the old version of the program in the backup library because both the current and backup libraries are concatenated. Figure 5-5 illustrates how the old version of ACCNTPGM executes with consistency token CT1, and DB2 retrieves the package from the fallback collection rather than the current collection because of the matching consistency token CT1.

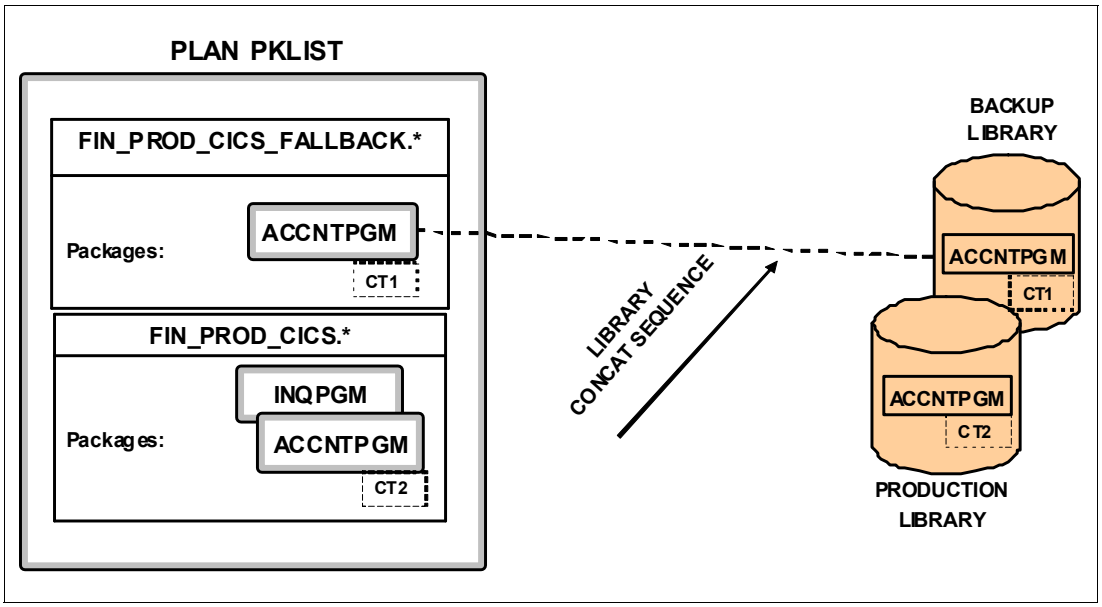

Figure 5-5 Fallback to Old Version of ACCNTPGM Program

Note that the program does not use the SET CURRENT PACKAGESET statement to select the collection it wants. Instead, it allows DB2 to automatically pick up the package based on the consistency token embedded in the program.

5.6 Special registers that affect package execution

Next we discuss some DB2 special registers, which have a significant impact on either package selection or package execution at program execution time. These registers can be set dynamically and alter one of the following elements:

- Location of execution
- Collection to be used for retrieving the package
- The schemas to be used for retrieving the package

5.6.1 CURRENT SERVER special register

When selecting a package at program execution time, the value in the CURRENT SERVER special register is used as the first package qualifier: location name.

The default value of the CURRENT SERVER special register is the local DB2. If your local DB2 has no location name, the default is blank. To set the CURRENT SERVER register with an appropriate value, use:

- The SQL CONNECT statement in the program
- The CURRENTSERVER option when binding the plan.

The PKLIST option specified in the BIND PLAN command can limit the values allowed for the CURRENT SERVER special register. If the location name was specified as "*", any location can be used to execute the package. If the location was explicitly specified, an error is returned when execution at an invalid location is attempted. For more details, see 2.1.9, "PKLIST in BIND PLAN" on page 21.

5.6.2 CURRENT PACKAGESET special register

When selecting a package at program execution time, the value in the CURRENT PACKAGESET special register is used as the collection ID for the package being searched. Blank is the default value of the CURRENT PACKAGESET special register. There is a major difference between a blank value and a set value in the package search algorithm.

When the value in this special register is blank, DB2 searches for the appropriate executable packages using entries specified by the PKLIST option on the BIND PLAN command.

When the value in the CURRENT PACKAGESET special register is not blank, DB2 searches for the package using PKLIST entries from the BIND PLAN that contain a collection ID with the same value as the CURRENT PACKAGESET special register. Setting the special register that does not match any entry in the package list returns an error at execution time.

Note: The SET CURRENT PACKAGESET statement does not fail if the value is invalid for any reason. The register is not checked until the next SQL statement is executed (not including CONNECT statements). An example of the runtime error can be found in 7.4.8, "Result of a SET CURRENT PACKAGESET to a missing PKLIST entry" on page 156

Special rules apply to this special register when used with routines. You can find more details in *DB2 Version 9.1 for z/OS SQL Reference*, SC18-9854.

5.6.3 CURRENT PACKAGE PATH special register

In principle, this special register is similar to the CURRENT PACKAGESET special register, except that a *series* of schemas (collections) can be specified instead of a single collection ID.

In practice, the usage can provide complex support for a hierarchy of sources for resolution of package names. A single collection ID can be specified, or other special registers and variables can be used to create a string that is expected to represent a concatenation of collection IDs.

The best example for when this can be useful is when a package is being executed using a remote connection to DB2. The DB2 plan, DISTSERV, does not allow collections to be defined. A stored procedure can use the COLLID option to specify a single collection, but allowing a hierarchy of collections is often more desirable.

This special register is particularly useful when nested programs are used that can be invoked as outside the direct scope of the application being executed. For example, code written to calculate credit risk could be packaged as a stored procedure or function and called by any authorized application. The set of calls to subprograms and/or routines might not be known to the calling application. The credit risk routine can use the CURRENT PACKAGE PATH special register to specify the list of collections relevant to its processing without requiring bind changes in the calling application.

5.7 Using collections to support access to mirror tables

Collections can be used to support parallel or mirror tables.

Mirror tables are tables that have identical table names and structures but different qualifiers to distinguish different instances. Assume that a set of tables with one owner are bound to packages in a collection, and assume that another set of tables *with the same structure but a different owner* are bound to packages in a different collection ID.

If both collection IDs are included in the PKLIST for a plan, then from the DB2 perspective, either collection can be used to get to tables. However, the packages are all bound from the same programs, so we need to tell DB2 which collection to access when the programs are running.

The SET CURRENT PACKAGESET statement is used at runtime to set a special register containing the collection ID to be used by DB2 for accessing packages. Use of this in the program tells DB2 which collection ID to use and, therefore, which set of tables to access.

Figure 5-6 shows the principles involved in supporting mirror tables using collections.

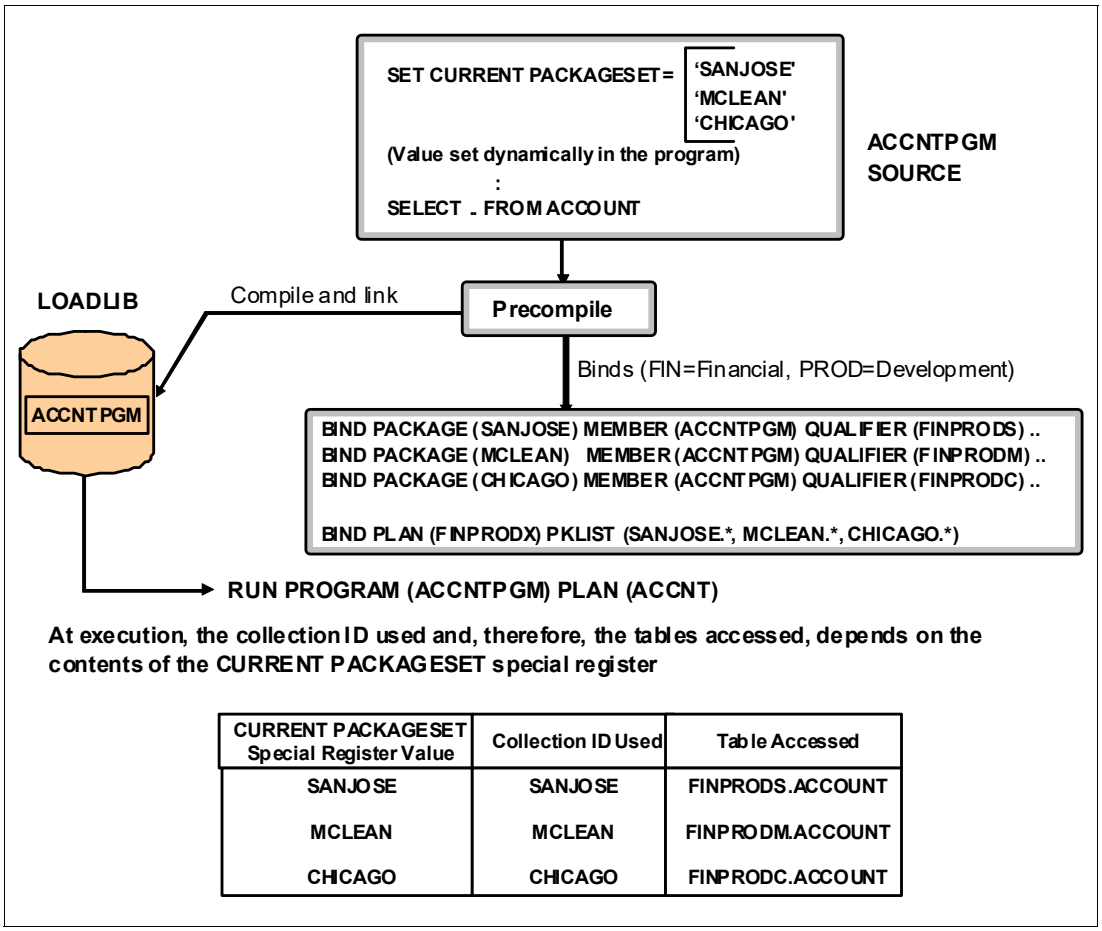

Figure 5-6 Collection support of mirror tables

There are two points to reiterate in the architecture shown:

- The first point to note is the SET CURRENT PACKAGESET statement, which indicates the collection ID to be used. In this example, three sets of tables contain stock data for warehouses at three locations, namely, SANJOSE, MCLEAN, and CHICAGO.
- The second point is that, as always, the table names in the program are unqualified.

The program involved, ACCNTPGM, is precompiled, compiled, and linked only once. However, the DBRM for the program is bound as a package three times, with each bind specifying different a collection ID and different value for the QUALIFIER option to identify the set of tables to be used. Note that the same OWNER option can be specified each time if table authorizations allow it.

In this case, when the package in collection SANJOSE is used, the table that would be accessed is FINPRODS.ACCOUNT. Either the same DBRM can be used for all three binds, or the COPY option of the BIND PACKAGE command can be used after the first bind to create the additional packages.

At runtime, the collection ID to be used is identified by the value in the CURRENT PACKAGESET register set dynamically by the program. This value could be set via a parameter to the program, passed at execution time.

The program could also access data for all locations in one execution of the program by looping and setting the special register value in turn to each of the collection IDs.

Note that the CURRENT PACKAGESET special register is local to the thread (user) executing the plan. When the special register is set to a collection ID using the SET CURRENT PACKAGESET statement, it remains effective until it is reset by another SET CURRENT PACKAGESET statement or until the thread terminates.

Figure 5-7 illustrates a programming option using the SET CURRENT PACKAGESET statement. Program A calls program B. Program B is a routine bounds into multiple collections, and the program needs to know which collection ID to use. Program B issues SET CURRENT PACKAGESET = 'PAYROLL'. Before this, program B must save the previous value of the CURRENT PACKAGESET special register so that, just before it exits, it can reset the register to the original value.

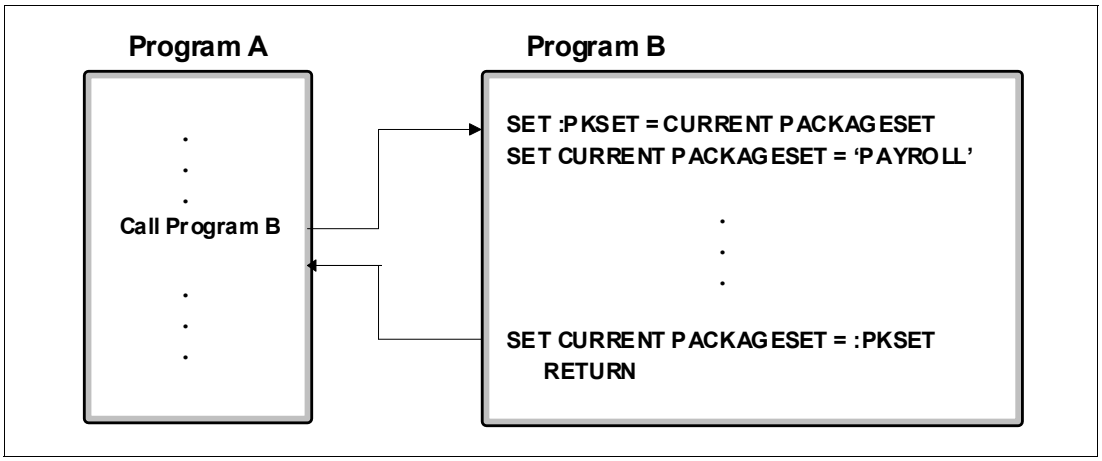

Figure 5-7 Saving CURRENT PACKAGESET register

Tip: The CURRENT PACKAGESET register is not global. The setting of it by one user of a plan does not affect another user executing the same plan.

5.8 Using a try-out collection

The try-out technique enables users to test or try out a new package without touching the package that is currently used. This technique can be used whether an updated program is being moved into a new environment, or an unchanged program is being tested to see if the access paths change as a result of a rebind.

This technique can be used in a number of scenarios:

- To evaluate the impact of a new index
- To evaluate the impact of updated statistics in the DB2 Catalog
- After maintenance has been installed on DB2 that might affect optimization

The package is bound into a try-out collection, either from a DBRM or using the COPY option. The try-out collection is not included in the package list for any plan, and so cannot affect any executable code in any environment. Also, the original package is untouched and remains accessible. The process is also discussed in 10.5, “Simulating possible access path changes” on page 203.

5.9 Using packages with routines

Packages are implemented and managed differently depending on what kind of routine is being used.

Special considerations for native SQL procedures

Unlike traditional application programs, the CREATE PROCEDURE statement automatically creates an associated package.

To use the procedure, EXECUTE authorization needs to be explicitly granted, even if it is called from within a plan for which the user already has the EXECUTE privilege.

Migration from one environment to another, whether it is to a different collection in the same DB2 subsystem or a remote DB2 subsystem requires use of the DEPLOY option on the BIND PACKAGE command.

Special considerations for external SQL procedures

Stored procedures backed by traditional programming languages follow more closely to traditional package-based development.

To use the procedure, EXECUTE authorization needs to be explicitly granted to the procedure and the supporting package, even if it is called from within a plan for which the user already has the EXECUTE privilege. For more details, see 9.4, “Privileges required to execute stored procedures” on page 183.

Migration from one environment to another requires that the procedure be defined in the target schema, whether it is local or remote to the current DB2 subsystem, then the supporting package needs to be bound or copied as for any other application package.

Special considerations for external user-defined functions (UDFs)

Although their definition is more complicated because of the need for precision in defining the parameters passed to the UDF (known as the *function signature*), in other respects their execution and migration processing follows that of an external stored procedure.

Note that *sourced* and *SQL scalar* functions do not use packages.

Special considerations for triggers

Like a native SQL procedure, creation of a database trigger automatically creates an associated package. Many of the package options are defined automatically; some can be changed using the REBIND TRIGGER PACKAGE command.

Unlike a native SQL procedure, it is not possible to migrate it to another environment using a BIND PACKAGE command; the CREATE TRIGGER definition must be made for each schema independently. It is not possible to create multiple versions of a trigger.

No authority is needed to execute a trigger because it becomes part of the database schema itself. In addition, the user that invokes the trigger does not need any privileges to execute any routines invoked by the trigger.

5.10 Accessing data from other applications

Package-based plans provide various mechanisms for interfacing between the current application and other applications, which we discuss in the following sections:

- Gaining direct access to another application's data locally
- Gaining direct access to another application's data remotely
- Using collections to access another application's functionality
- Using stored procedures from another application
- Using other mechanisms to invoke another application's functionality

5.10.1 Gaining direct access to another application's data locally

Aliases can provide pointers to tables with different owners in local or remote DB2 subsystems. We assume that proper authorization has been granted to access and/or update data from another application and our discussion is focused on the mechanics of using an alias with packages.

If the application is coded normally, all tables referenced in the SQL are unqualified — no CREATOR value is specified. When the DBRM for the program is bound into a package, the tables are all qualified with the same QUALIFIER value. However, the tables in another application almost certainly have a different CREATOR value. Defining an alias allows for direct access from the local application, and SQL that is statically-bound to the data in the other application. The following examples illustrate the scenario and resolution.

Example 5-8 shows why we consider a requirement for access to another application.

Example 5-8 Accessing the current application

```
SELECT CONTRACT_VALUE,
       CONTRACT_START_DATE,
       CONTRACT_MANAGER_ID
FROM CONTRACT
WHERE CONTRACT_ID = :contract-id
```

If we bind the SQL from Example 5-8 with QUALIFIER(FINTEST), we can retrieve the ID of the employee managing a Contract as well as other information that we find useful. But, if we want the *name* of the manager, perhaps we could need to join to the EMPLOYEE table in the Payroll application. Example 5-9 shows how we could join the tables.

Example 5-9 Extending our SQL to include data from another application

```
SELECT CNT.CONTRACT_VALUE,
       CNT.CONTRACT_START_DATE,
       CNT.CONTRACT_MANAGER_ID,
       EMP.EMPLOYEE_NAME AS CONTRACT_MANAGER_NAME
FROM CONTRACT CNT
   , EMPLOYEE EMP
WHERE CNT.CONTRACT_ID = :contract-id
  AND EMP.EMPLOYEE_ID = CNT.CONTRACT_MANAGER_ID
```

If the creator of the tables in the Payroll application is PAYTEST, then binding with a QUALIFIER value of FINTEST fails with SQLCODE -204 because we can only find the tables associated with the Financial application. The same thing would happen if we tried to access the EMPLOYEE table directly, without a join.

Example 5-10 creates the alias that we need.

Example 5-10 Defining an alias

```
SET CURRENT SQLID = 'FINTEST';
CREATE ALIAS EMPLOYEE
FOR PAYTEST.EMPLOYEE;
```

With the alias in place, the PAYTEST.EMPLOYEE is also known as FINTEST.EMPLOYEE and binding with a QUALIFIER value of FINTEST is successful, and retrieves the data.

5.10.2 Gaining direct access to another application's data remotely

There are two main ways for directly accessing data from another location when that data is on a remote DB2 subsystem:

- Aliases can be used point to remote tables (known as "system-directed access")
- A local program can issue a CONNECT statement to establish a connection to another DB2 (or non-DB2) location, and execute SQL at the remote location (known as "application-directed access")

In the previous examples, FINTEST and PAYTEST were qualifiers for tables on the same DB2 subsystem. It is possible to continue to operate when they are not on the same DB2 subsystem. However, there is a restriction: you cannot join tables from different locations. To continue the previous example, Example 5-11 shows how a remote alias could be defined.

Example 5-11 Creating an alias that can be used to access a remote table

```
-- Create the local copy of the Alias
SET CURRENT SQLID = 'FINTEST';
CREATE ALIAS EMPLOYEE
FOR SANJOSE.PAYTEST.EMPLOYEE;
-- Create the remote copy of the Alias
CONNECT TO SANJOSE;
SET CURRENT SQLID = 'FINTEST'
CREATE ALIAS EMPLOYEE
FOR PAYTEST.EMPLOYEE;
```

The use of the DRDA protocol for package, and the requirements it imposes for binds is discussed in more detail in 6.2, “Program preparation for distributed applications in DRDA” on page 105.

Without the ability to join tables between multiple DB2 subsystems, the SQL to retrieve data from the CONTRACT and EMPLOYEE tables needs to be separated, as shown in Example 5-12.

Example 5-12 Accessing the local and remote data via a 3-part alias

```
-- Get some data from the local location
SELECT CONTRACT_VALUE,
       CONTRACT_START_DATE,
       CONTRACT_MANAGER_ID
FROM CONTRACT
WHERE CONTRACT_ID = :contract-id;
-- Now get the rest of the data from the remote location using the Alias
SELECT EMPLOYEE_NAME AS CONTRACT_MANAGER_NAME
FROM EMPLOYEE
WHERE EMPLOYEE_ID = :ws-contract-manager-id;
```

As for all DRDA programming, the package containing this SQL needs to be bound at both the local and remote locations.

Note that it is possible to create a local temporary table, populated with the data from the remote location. This would allow full SQL capabilities against the EMPLOYEE table. Consideration would need to be given to locking and data currency issues with this approach. A complete discussion of the issues is outside the scope of this book, but you would need to decide whether or not to hold locks on the data at the remote site, or whether it is acceptable for the data at the remote site to be changed while the local copy of the table is used.

An alternative to using a 3-part alias that points to a remote location is to connect to the remote location (application-directed access), and retrieve data from the EMPLOYEE table as a ‘local’ table. This is very similar to the previous example, except there is only a requirement for a 2-part alias to be defined at both location, using the same syntax as shown in Example 5-13.

Example 5-13 Creating an alias that can be used to access a table at a remote location

```
-- Create the local copy of the Alias
SET CURRENT SQLID = ‘FINTEST’;
CREATE ALIAS EMPLOYEE
FOR PAYTEST.EMPLOYEE;
-- Create the remote copy of the Alias
CONNECT TO SANJOSE;
SET CURRENT SQLID = ‘FINTEST’
CREATE ALIAS EMPLOYEE
FOR PAYTEST.EMPLOYEE;
```

Again, the SQL to retrieve data from the CONTRACT and EMPLOYEE tables requires two SQL statements, as shown in Example 5-14.

Example 5-14 Accessing the local and remote data via a CONNECT

```
-- Get some data from the local location
SELECT CONTRACT_VALUE,
       CONTRACT_START_DATE,
       CONTRACT_MANAGER_ID
FROM CONTRACT
WHERE CONTRACT_ID = :contract-id;
-- Now get the rest of the data from the remote location
CONNECT TO :our-remote-location;
SELECT EMPLOYEE_NAME AS CONTRACT_MANAGER_NAME
FROM EMPLOYEE
WHERE EMPLOYEE_ID = :ws-contract-manager-id;
-- Time to go home...
CONNECT RESET;
```

When we created an alias, we explicitly specified the location name in the CONNECT statement because this is a one-off database change. When a location is accessed in a program, flexibility is enhanced by using a host variable instead of an explicit location name. It allows the program to connect to multiple locations in a loop using a table, and it allows for future changes to the location name.

5.10.3 Using collections to access another application's functionality

There can be good reasons why accessing the data directly is not considered sensible:

- The owner of the data might be concerned about the access paths needed to support access to the data.
- There might be security issues that make only some rows or columns in the tables available for access by other applications (this can sometimes be resolved via a view if direct access to the data is still a preferred solution).
- The data might be on a remote DB2 subsystem where the tables are not available for direct access or join processing.
- Changes to the database structure or column data types might be planned.

In these and other scenarios, calling the functionality (or *program code*) of the other application can make more sense:

- The owner of the data retains control over the access paths used to get to the data.
- The owner of the data is able to provide defined interfaces to the data.
- The owner of the data can provide additional validation of authorization to access and/or update the data.
- The other application can be located on a local or remote DB2 subsystem.
- Planning the implementation of database changes is simpler when external organizations are not directly retrieving data.

Invoking a package in the other application

We might have our application defined in Production as simply as this:

```
PKLIST(FIN_PROD_CICS.*)
```

We can include code from the Payroll application in the plan (support must also be provided in the load library hierarchy):

```
PKLIST(FIN_PROD_CICS.*, PAY_PROD_CICS.*)
```

A call from a Financial program to a Payroll program allows DB2 to load the package associated with the Payroll program successfully as it finds the package in the FIN_PROD_CICS collection.

Note that no changes are required to the Financial plan when packages are added or changed in the Payroll environment.

Note also that the financial application team might want to be aware of changes to the Payroll code they use, especially in the event of a sudden degradation in the performance of the changed payroll code. Another situation to be concerned about is when changed Payroll code has introduced a bug and needs to fallback.

5.10.4 Using stored procedures from another application

It can be difficult to invoke packages from another application directly. The other application might be in a different DB2 subsystem, or it might not be possible to access the other application's load libraries. In this case, calling a DB2 stored procedure defined by another application can provide tightly-defined access to its functionality.

This allows the code for the other application to be located at the most sensible location, to be defined with different environmental parameters to the local application, and to be maintained independently (again, provided the interface is carefully managed and documented).

If the application is on the local DB2 subsystem, the SQL CALL statement invokes the stored procedure. Resolving the name (which probably has a different schema qualifier than the one used by the current application) normally requires the CURRENT PATH special register to be set. When the stored procedure is invoked, if it is supported by a program (rather than native SQL), the COLLID or PACKAGE PATH options used in the CREATE PROCEDURE statement define where DB2 looks for the supporting package(s) at runtime. Example 5-15 shows how this could work.

Example 5-15 Calling a local stored procedure from another application

```
SET CURRENT PATH = :ws-payroll-schema-list
CALL GET_CURRENT_SALARY (:ws-emp-id, :ws-curr-salary)
```

If the application is on a remote DB2 subsystem, the only difference is that it is necessary to connect to the remote DB2 subsystem before issuing the SET and CALL statements, as shown in Example 5-16.

Example 5-16 Connecting to a remote DB2

```
CONNECT TO :ws-payroll-locn
SET CURRENT PATH = :ws-payroll-schema-list
CALL GET_CURRENT_SALARY (:ws-emp-id, :ws-curr-salary)
CONNECT RESET
```

There are some limitations on the parameters that can be passed, and you should be aware that a stored procedure can return an entire result set at once, without multiple invocations.

5.10.5 Using other mechanisms to invoke another application's functionality

Accessing the program directly or via a DB2 stored procedure might not always be possible.

Linkage to another application can be managed via many other mechanisms, including but not limited to:

- CICS Distributed Program Link (DPL)
- CICS Multi-Region Operation (MRO)
- CICS Inter-System Communication (ISC)
- Messaging via MQ Series or equivalent tool
- Invocation of a remote Web service using one of the IBM-supplied SOAP UDFs

6

Developing distributed applications

Due to the evolution of the computing model, support for distributed applications has become one of the most important functions for databases. In this chapter, we describe the basic concepts of Distributed Relational Database Architecture™ (DRDA), connection and transaction pooling and multi-site updates as the basis for distributed database applications.

Given the widespread use of Java for distributed applications, we also describe Java DataBase Connectivity (JDBC) and Structured Query Language for Java (SQLJ). We introduce pureQuery as a new and efficient option for Java application development with DB2.

We cover the following topics:

- DRDA and distributed relational databases
- Program preparation for distributed applications in DRDA
- Connection pooling
- DB2 sysplex workload balancing
- Coordinating updates across multiple systems
- DB2 drivers and packages
- Considerations for distributed packages
- SQLJ
- pureQuery
- Private protocol

6.1 DRDA and distributed relational databases

In this section, we provide a brief overview of DRDA, focusing on the points that are essential to an understanding of distributed relational database applications.

For the architecture, see *DRDA Version 4, Volume 1: Distributed Relational Database Architecture (DRDA).* available from:

http://www.opengroup.org/bookstore/catalog/c066.htm

For the DB2 functions, see *DB2 Version 9.1 for z/OS Reference for Remote DRDA Requesters and Servers,* SC18-9853, and *Distributed Functions of DB2 for z/OS and OS/390*, SG24-6952.

6.1.1 DRDA overview

IBM DB2 distributed database functionality is based on DRDA, an open, vendor-independent architecture for providing connectivity between a client and a database server. DRDA has been adopted by The Open Group as an industry standard interoperability protocol. Connectivity is independent from hardware and software architecture as well as from vendors and platforms.

DRDA supports SQL as the standardized API for execution of applications and it defines flows and logical connections between the application and a relational database management system (DBMS), program preparation, and BIND SQL statements for target DBMS.

Figure 6-1 shows an overview of DRDA topology.

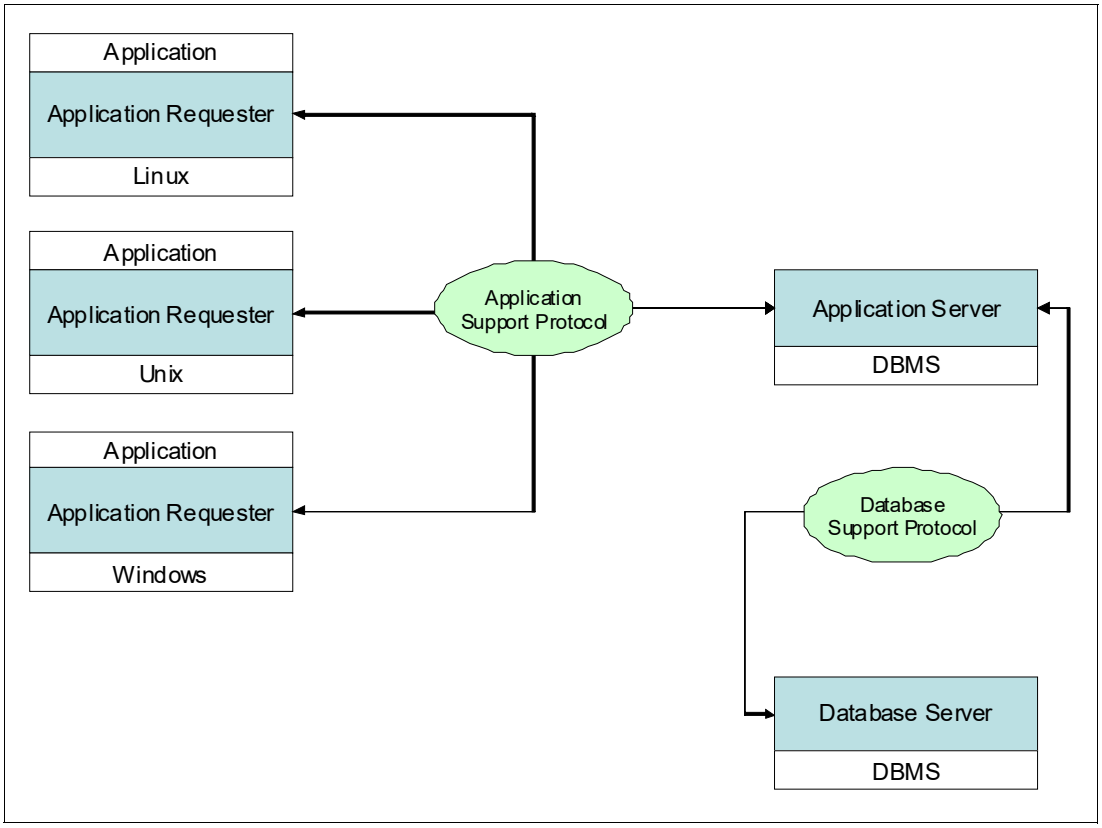

Figure 6-1 DRDA topology overview

DRDA provides three function types. Each function type plays its own role in DRDA:

- Application Requester (AR):

  Supports SQL and program preparation services from applications.

- Application Server (AS):

  Supports requests that application requesters have sent and routes requests to database servers by connecting as an application requester.

- Database Server (DS):

  Supports requests from application servers, forwarding them to other database servers, and the propagation of special register settings.

Also, DRDA provides two protocols:

- Application Support Protocol:

  Provides connection between ARs and ASs.

- Database Support Protocol:

  Provides connection between ASs and DSs.

DRDA has four degrees of distribution (DRDA levels 1, 2, 4, and 5), as shown in Table 6-1. Each level of DRDA defines what can be performed within a unit of work (UOW).

Table 6-1 DRDA degrees of distribution

DRDA levels (1, 2, 4, 5)	**SQL statements per UOW**	**DBMS per UOW**	**DBMS per SQL statement**
1. Application-directed remote UOW	Many	1	1
2. Application-directed distributed UOW	Many	Many	1
4. Database-directed access	Many	Many DBMS coordinates the UOW with multiple DBMSs	Many DBMS distributes the request
5. XA distributed transaction processing	Many UOW is identified by an XID. XID stands for “transaction branch identifier”.	Many With XID, Transaction Manager coordinates multiple UOWs with the registered application requester	Many Application server can distribute the requests to multiple DBMSs

- Application-directed remote unit of work (RUOW):

  This level is architected in DRDA level 1 and it is not recommended. This level supports, within a logical UOW, application access to just one remote database system. Before a program can access another database, it must end the current UOW before starting another one. This is done by issuing either a COMMIT or ROLLBACK statement.

- Application-directed distributed unit of work (DUOW):

  This level is architected in DRDA level 2. This level extends RUOW distributed database capability by allowing, within a single logical UOW, application read and write access to data at the local and multiple remote database systems. The program can read and update multiple databases before ending the UOW and starting another one. However, a single SQL statement cannot access more than one database.

- Database-directed access:

  This level is architected in DRDA level 4. In addition to RUOW and DUOW capability, it allows a single SQL statements to access multiple databases. In other words, an application connects to a relational database management system (DBMS) that can execute one or more SQL requests locally or route some or all of the SQL requests to other DBMSs. DBMS determines which system manages the data referenced by the SQL statement and directs the request to that system.

- XA distributed transaction processing:

  This level is architected in DRDA level 5. The XA-compliant Transaction Manager coordinates UOW among the registered application requestors. The application server can distribute single SQL statements to access multiple databases. DB2 for z/OS supports a subset of the DRDA XA TM. See SYNCCTL command instance variable in the *DB2 Version 9.1 for z/OS Reference for Remote DRDA Requesters and Servers,* SC18-9853 for more details on the XA support.

Each DRDA level of the implementation as defined by the Open Group documentation represents an increasing level of distributed functions supported. Support for decimal floating point, XML, and binary string data types have been added with DB2 9 for z/OS.

6.1.2 Applications and data layer

In this section, we describe the layers used to implement distributed relational database applications.

By definition, *distributed* means that the application requires data from a remote DBMS rather than a local DBMS. Using DRDA connectivity, you can access data on a DB2 for z/OS from client applications running on remote and different platforms, and from DB2 for z/OS access remote non-DB2 for z/OS systems. Figure 6-2 illustrates these layers.

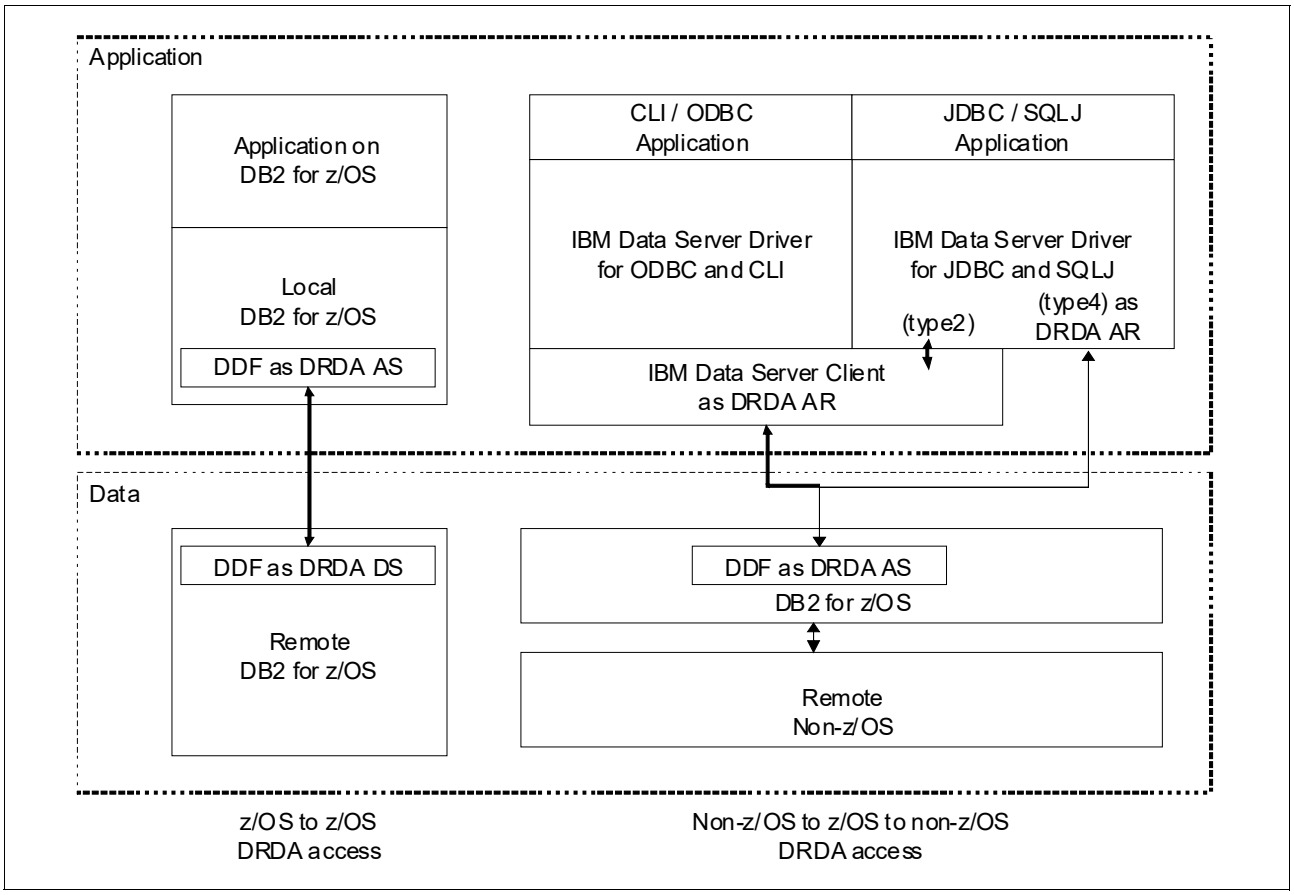

Figure 6-2 Application and data layer

In case of z/OS to z/OS access, the local DDF plays a role of DRDA AS and remote DDF plays the role of DRDA DS.

In case of non-z/OS to z/OS, IBM Data Server Client, which is a part of DB2 Connect™, plays the role of DRDA AR, and remote DDF plays the role of DRDA AS. When DB2 Connect is used as a gateway, it is an AS. IBM Data Server Driver for JDBC and SQLJ can connect directly to DB2 for z/OS using its type 4 connectivity. We discuss Java Database Connectivity (JDBC) in “DB2 drivers and packages” on page 114.

DDF supports both network protocols, SNA and TCP/IP.

6.2 Program preparation for distributed applications in DRDA

In this section, we describe the steps to bind DRDA packages in DB2. Note that *local* also means the AR in DRDA and *remote* means the AS.

6.2.1 Distributed packages

With distributed applications, a package is the control structure produced when the SQL statements in the application program are bound to a relational DBMS. The DBMS uses the control structure to process SQL statements encountered during statement execution.

There is no DBRM on non-z/OS systems, because a DBRM is a z/OS only object. The program preparation and the parameters are different, depending on the target system to connect to.

DISTSERV is a standard plan provided by IBM for distributed requests on z/OS. You do not need to explicitly bind the plan. You need to bind packages. These packages run under DISTSERV.

6.2.2 DB2 for z/OS to DB2 for z/OS

In this section, we describe the preparation of the embedded SQL programs in DB2 for z/OS.

Precompiler options

Before bind, you need to precompile the source program. The following precompiler options are important for distributed application:

CONNECT Use the default CONNECT(2) precompiler option because it allows the flexibility to access more than one DBMS in a UOW. Using CONNECT(1) explicitly is not recommended because it only allows your programs to connect to one DBMS at a time.

SQL If you are accessing a non-DB2 for z/OS system, use the SQL(ALL) option. This option allows you to use any statement that is in the DRDA standard. If you are only accessing DB2 for z/OS, you can use SQL(DB2). In that case, the precompiler only accepts statements that are used by DB2 for z/OS.

Binding

Note that binding remote packages means that the remote DB2 records the description of the package in its catalog table and saves the prepared package in its directory. To perform this operation, you need a DBRM in the local DBRM library or a package in the local DB2 directory.

In our environment, DB9B is the remote site and DB9A is the local site, both DB2 for z/OS.

In general, you can bind remote packages in two ways:

- Remote bind a package at the remote site using a DBRM which resides at the local site. See Example 6-1.

Example 6-1 Remote binding a package from a DBRM

```
-- Remote package bind running on the local site, binding DBRM PKG
-- in local DBRM library on remote location DB9B into collection CREMOTE
BIND PACKAGE(DB9B.CREMOTE) MEMBER(PKG)
```

- Remote bind a package at the remote site using a copy of a local package that resides in the local DB2 directory as input. See Example 6-2.

Example 6-2 Remote binding a package from a copy of package

```
-- Remote package bind running on the local site,
-- copying package PKG in collection CLOCAL on local DB2 to remote DB2 DB9B,
-- and binding on remote site collection CREMOTE
BIND PACKAGE(DB9B.CREMOTE) COPY(CLOCAL.PKG)
```

In addition, you can use the following procedure, which only applies if the AS is a DB2 for z/OS, because it can handle DBRMs:

1. Ship the DBRM to the remote site, for example using FTP
2. Locally bind the DBRM into a package at the remote site. See Example 6-3.

Example 6-3 Binding a package from a shipped DBRM

```
-- Bind package at the remote site DB9B after shipping the DBRM PKG
-- This is identical to a local bind
BIND PACKAGE(CREMOTE) MEMBER(PKG)
```

Tip: You need to transfer the DBRMs to the z/OS system to a partitioned data set as binary files with record format FB and record length 80.

Bind package options

The following options can be useful in some cases. See also Chapter 2, “BIND options” on page 17.

DEFER — For efficient dynamic SQL, use DEFER(PREPARE) to improve performance instead of NODEFER(PREPARE). DB2 does not prepare the dynamic SQL statement until that statement executes. The exception to this situation is dynamic SELECT, which combines PREPARE and DESCRIBE, regardless of whether the DEFER(PREPARE) option is in effect. When a dynamic SQL statement accesses remote data, the PREPARE and EXECUTE statements can be transmitted over the network together and processed at the remote location. Responses to both statements can be sent back to the local subsystem together. This reduces network traffic, which improves the performance of the dynamic SQL statement.

SQLERROR — Use SQLERROR(CONTINUE) so that when there are statements that are not valid at the current server, the package is still created. Also, use this option when binding the same package at multiple servers where objects might not be valid on all of the servers, because the objects do not exist there. Using this option solves this problem.

KEEPDYNAMIC — Use KEEPDYNAMIC(YES) when you need to cache prepared statements. See also 6.7.4, “Dynamic SQL statement caching” on page 122.

6.2.3 DB2 for z/OS to DB2 for Linux, UNIX, and Windows

In Example 6-4, we describe the preparation of an embedded SQL programs in DB2 for z/OS. In this case we bind SPUFI to a DB2 for LUW system called REMLUW.

Example 6-4 Binding SPUFI locally and remotely

```
//BNDSPUFI JOB 'USER=$$USER','<USERNAME:JOBNAME>',CLASS=A,
//         MSGCLASS=A,MSGLEVEL=(1,1),USER=ADMF001,REGION=0K,
//         PASSWORD=USERA
//JOBLIB  DD  DISP=SHR,DSN=DB8A.TESTLIB
//        DD  DISP=SHR,DSN=DB8A.SDSNLOAD
//BNDSPUFI EXEC  PGM=IEFBR14
//* NOTE IF JOBLIB PRESENT, MUST BE PLACED BEFORE IEBR14 STEP
//BNDSPUFI EXEC TSOBATCH,DB2LEV=DB8A
```

```
//DBRMLIB  DD DSN=DB8A.SDSNDBRM,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
DSN SYSTEM(DB8A) R(1) T(1)
*
BIND PACKAGE (SC63TS.DSNESPCS)    MEMBER(DSNESM68) -
     ACTION(REPLACE) ISOLATION(CS)
BIND PACKAGE (REMLUW.DSNESPCS)    MEMBER(DSNESM68) -
     ACTION(REPLACE) ISOLATION(CS)
BIND PLAN(DSNESPCS) PKLIST(SC63TS.DSNESPCS.DSNESM68 -
                           REMLUW.DSNESPCS.DSNESM68  -
     ISOLATION(CS) ACTION(REPLACE)
BIND PACKAGE(SC63TS.DSNESPRR) MEMBER(DSNESM68) -
     ACTION(REPLACE) ISOLATION(RR)
BIND PACKAGE(REMLUW.DSNESPRR) MEMBER(DSNESM68) -
     ACTION(REPLACE) ISOLATION(RR)
BIND PLAN(DSNESPRR) PKLIST(SC63TS.DSNESPRR.DSNESM68  -
                          REMLUW.DSNESPRR.DSNESM68  -
     ISOLATION(RR) ACTION(REPLACE)
RUN PLAN(GDNTEP3) PROGRAM(DSNTEP3)
END
/*
//SYSIN    DD *
  GRANT EXECUTE, BIND ON PACKAGE DSNESPCS.* TO PUBLIC;
  RETCODE = SQLCHECK(0,+562);
  GRANT EXECUTE, BIND ON PACKAGE DSNESPRR.* TO PUBLIC;
  RETCODE = SQLCHECK(0,+562);
  GRANT EXECUTE, BIND ON PLAN DSNESPCS TO PUBLIC;
  RETCODE = SQLCHECK(0,+562);
  GRANT EXECUTE, BIND ON PLAN DSNESPRR TO PUBLIC;
  RETCODE = SQLCHECK(0,+562);
 COMMIT;
 RELEASE ALL;
CONNECT TO REMLUW;
*-104 on this wildcard syntax on udb server, and collid
* GRANT EXECUTE, BIND ON PACKAGE DSNESPCS.* TO PUBLIC;
  GRANT EXECUTE, BIND ON PACKAGE DSNESPCS.DSNESM68  TO PUBLIC;
  RETCODE = SQLCHECK(0,+562);
* GRANT EXECUTE, BIND ON PACKAGE DSNESPRR TO PUBLIC;
* RETCODE = SQLCHECK(0,+562);
  GRANT EXECUTE, BIND ON PACKAGE DSNESPRR.DSNESM68  TO PUBLIC;
  RETCODE = SQLCHECK(0,+562);
*no plan in udb
* GRANT EXECUTE, BIND ON PLAN DSNESPCS TO PUBLIC;
* RETCODE = SQLCHECK(0,+562);
* GRANT EXECUTE, BIND ON PLAN DSNESPRR TO PUBLIC;
* RETCODE = SQLCHECK(0,+562);
 COMMIT;
 RELEASE ALL;
/*
```

6.2.4 DB2 for Linux, UNIX, and Windows to z/OS using DB2 Connect

DB2 Connect forwards SQL statements submitted by application programs running in non-z/OS, such as Linux®, UNIX and Windows® (LUW), to z/OS database servers. DB2 Connect can forward almost any valid SQL statement, as well as the supported DB2 APIs:

- JDBC
- SQLJ
- ADO.NET
- OLE DB
- ODBC
- C
- Perl
- PHP
- DB2 CLI
- Embedded SQL

In an LUW environment, we usually use two types of packages, packages with embedded SQL and the DB2 CLI packages. In this section, we describe embedded SQL programs. For DB2 CLI packages, see 6.6.5, “Packages used by DB2 drivers” on page 118.

In general, the program preparation steps for embedded SQL programs are similar to the ones for DB2 for LUW. See Figure 6-3.

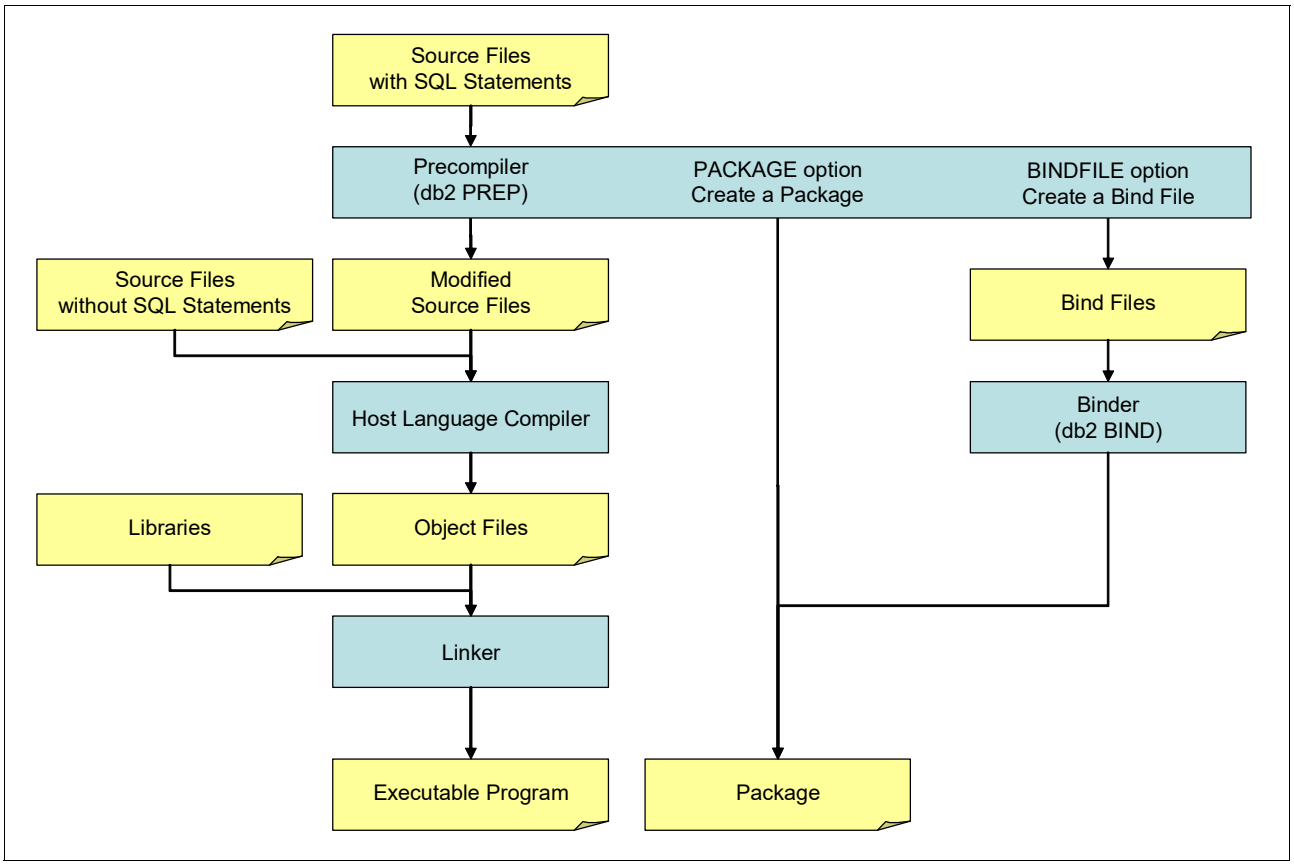

Figure 6-3 Program preparation flow for embedded SQL programs in LUW

You can bind remote packages in two ways:

- Let the precompiler bind a package at the remote site with the PACKAGE option.
- Let the precompiler generate a bind file with BINDFILE; bind a package at the remote site.

Example 6-5 shows the steps for the DB2 command line processor (CLP).

Example 6-5 Preparing and binding an embedded SQL C program

```
-- connect to remote database DB9A
db2 CONNECT TO DB9A

-- let precomplier bind package at remote site in one action with PACKAGE option
db2 PREP testapp.sqc PACKAGE COLLECTION CREMOTELUW

-- let precompiler generate bind file with BINDFILE option
db2 PREP testapp.sqc BINDFILE
-- bind package at remote site collection CREMOTELUW
db2 BIND testapp.bnd COLLECTION CREMOTELUW
```

6.3 Connection pooling

In this section, we describe the basic concept of connection pooling. Connection pooling is the generic term given to techniques that allow the connection resources for distributed database access to be pre-loaded, shared, and reused efficiently. Figure 6-4 illustrates various possible types of the poolings in a 3-tier client-server system in the case of WebSphere Application Server as application server and a single DB2 for z/OS subsystem as data server.

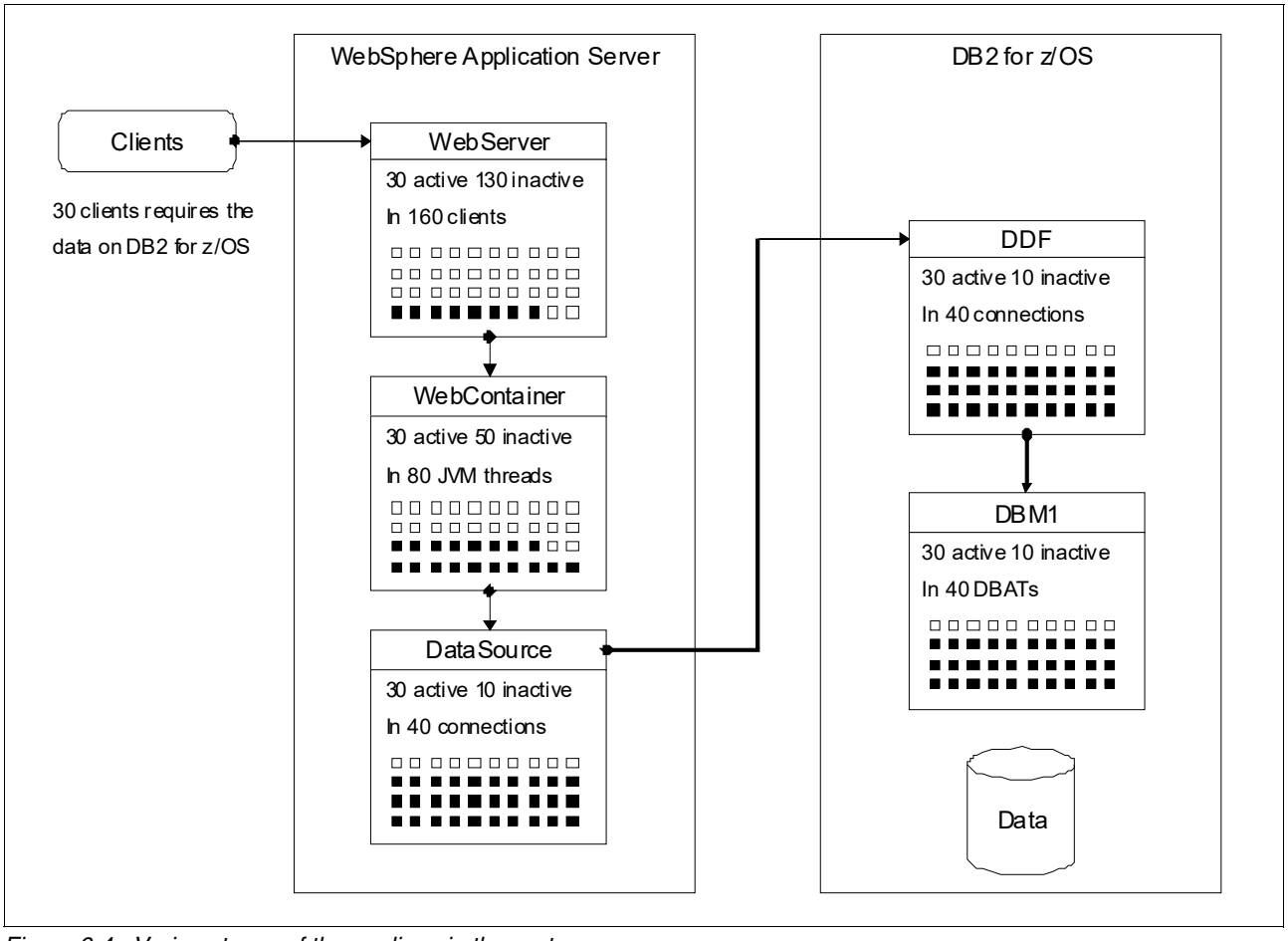

Figure 6-4 Various types of the poolings in the system

The reasons for the pools are as follows:

- To reduce CPU cost to allocate connections or executable units by reusing them
- To reduce virtual and real memory by sharing among the queued users
- To provide a way to limit the virtual and real memory used for each subsystem
- To provide a way to balance between costs and returns of the subsystems and optimize the whole in total

Because it looks like a network of pools and each pool behaves like a queue, it is also called a *queueing network*. For a theoretical review, see *Quantitative System Performance - Computer System Analysis Using Queueing Network Models,* available from:

http://www.cs.washington.edu/homes/lazowska/qsp/

6.3.1 DB2 thread pooling

In DB2 for z/OS, each inbound connection from distributed environments requires a DDF connection and a DB2 database access thread (DBAT). DB2 thread pooling is a mechanism to share DBATs between connected applications. It is implemented within DB2 for z/OS, and can be used by connections from any DRDA AR. This support is sometimes called type 2 inactive thread support, but DB2 thread pooling is a far better and more accurate description of this type of pooling.

DB2 thread pooling operates by separating the distributed connections in DDF from the DBATs that do the work in DBM1. A pool of DBATs is created for use by inbound DRDA connections. A connection makes temporary use of a DBAT to execute a UOW, and then releases it back to the pool at commit time for another connection to use. The result is a given number of inbound connections that require far fewer DBATs to execute the work within DB2. It is enabled through a DSNZPARM by specifying CMTSTAT=INACTIVE, which means that the DBATs status switches to INACTIVE at commit.

Note that an inactive type 2 thread is not a thread at all. It is a DDF connection that has no database stateful context information. It stores authentication and network connection information relating to the distributed application and can become associated with a pooled DBAT at any time, if the application executes another UOW.

Table 6-2 shows the behavior with DB2 thread pooling.

Table 6-2 Summary of behavior with and without DB2 thread pooling

Event	Behavior with DB2 thread pooling	Behavior without DB2 thread pooling
connect	► Create a new DDF connection. ► Reuse an existing DBAT.	► Create a new DDF connection. ► Create a new DBAT.
commit	► DDF connection is exclusively retained for the connection. ► DBAT is released to the pool	► DDF connection is exclusively retained for the connection. ► DBAT is exclusively retained for the connection
disconnect	► Destroy the connection in DDF and do not reuse. ► No impact on DBAT.	► Destroy the connection in DDF and do not reuse. ► Destroy the DBAT and do not reuse.

6.3.2 Considerations for packages

There is a DB2 limit to the number of times a DB2 thread can be reused. DB2 automatically recycles a thread after this limit has been reached, to prevent the possibility of storage creep in threads.

DB2 thread pooling is designed to automatically adjust the size of the thread pool to the incoming workload. If the workload increases, new DBATs are created, but they are only created once, and then reused. If the workload decreases, the unused DBATs are automatically terminated, based on the setting of the DSNZPARM POOLINAC time-out value, 120 seconds as default.

DBATs can be reused by any incoming connection, regardless of things such as inbound location and user ID. The only factors that prevent a thread being returned to the pool at commit time for reuse are programming constructs that maintain context after a commit statement. If the application has used packages bound with KEEPDYNAMIC YES, the thread has to keep the prepared SQL statement in memory for reuse.

6.4 DB2 sysplex workload balancing

The differentiator for DB2 data sharing is the workload balancing used to balance connections, at the transaction boundary, across the members of the data sharing group for application servers using a connection pools.

DB2 sysplex workload balancing functions provided by DB2 Connect Server and the IBM DB2 Driver for JDBC and SQLJ, and Connection Concentrator support are critical components of the DB2 data-sharing fault-tolerant environment. DB2 sysplex workload balances work across the DB2 group using connection concentration. The connection concentrator balances application connections by multiplexing transactions across a set of managed server connections to DB2.

To balance transactions (individual units of work delimited by a commit) across individual members of the DB2 group, unique member IP addresses are configured for each member using a Dynamic Virtual IP address (DVIPA) called the member-specific DVIPA. Member IP addresses are not used by remote applications because they route connections to a specific member. Setting up a member IP address as a DVIPA allows routing to work even if a member fails over to another LPAR using VIPA takeover. As soon as the member is started after a failure, any distributed in-doubt threads can be quickly recovered.

Refer to the IBM Redpaper publication, *DB2 9 for z/OS Data Sharing: Distributed Load Balancing and Fault Tolerant Configuration*, REDP-4449 for details.

6.5 Coordinating updates across multiple systems

In this section, we discuss the system's capability for two-phase commit and recovery of in-doubt UOW.

6.5.1 Two-phase commit capability

Two or more updates need to be coordinated if they must all commit or all roll back in the same UOW. This situation is common in banking. Suppose that an amount is subtracted from one account and added to another. The data changes must be confirmed to both accounts, or neither accounts.

To prepare for coordinated updates to two or more data sources, make sure that all systems that your program accesses implement two-phase commit processing. This processing allows updates to two or more systems to be coordinated automatically.

The coordinator and participants jointly implement a two-phase commit process. Let us consider an example in z/OS environment that IMS acts as the coordinator and DB2 as a participant. You can update an IMS database and a DB2 table in the same coordinated UOW. If a system or communication failure occurs between committing the work on IMS and on DB2, the two programs restore the two systems to a consistent point when activity resumes.

If you have a system that does not support two-phase commit, the coordinator does not allow you to update that system together with any other system in the same UOW. You have to commit before you can update another system. In this case, providing coordinated updates depends on the way you code your application. Your program must provide the coordination between systems.

6.5.2 Considerations for recovery of in-doubt UOW

Some kinds of communication failures or system failures during two phase commit processing can result in having in-doubt threads at the server side. There is always a short phase during two-phase commit processing when a DBAT is in the in-doubt status. If there is a communication failure between AR and AS during the interval, the DBAT can become an in-doubt thread.

When an in-doubt thread occurs, DB2 (when acting as a participant) cannot determine what action to take until the communication is brought up again and the coordinator notifies DB2. An in-doubt thread can cause locked and unavailable resources (of all changed rows and/or pages touched by the in-doubt unit of recovery) until the in-doubt thread is resolved.

DB2's automatic recovery mechanism resolves most distributed units of recovery, once the participants and the communication link between them are up again. However, if the communication cannot be brought up in a reasonable amount of time, and you do not want your resources to remain locked, or the coordinator has done a cold start, which prevents automatic in-doubt thread resolution, you might need to resolve the in-doubt UOWs manually.

If you are resolving these in-doubt units manually, you must identify which group of threads should be committed or rolled back together on all systems involved in the unit of recovery. A decision to commit or roll back a resource without DB2's automatic resolution is called a *heuristic decision*. If your decision to commit or roll back is different from the other system's decision, data inconsistency occurs and this damage is called *heuristic damage*.

Important: Avoid using the difficult process to make a correct heuristic decision and to resolve in-doubt units of recovery without an automatic recovery mechanism for the consistency of your data. You need to understand what is consistent in a business context, that is why it is called "heuristic."

To gather information about the in-doubt UOWs, use the following sources of information:

- -DISPLAY THREAD(*) TYPE(INDOUBT) command
- MSTR JES log
- MVS system log
- SMF data (statistics trace class 4)
- DB2 logs using DSN1LOGP utility

The information you need must include the LUWID (Logical Unit of Work ID) of the in-doubt thread, and its status, whether it is directed to a COMMIT or a ABORT by querying the coordinator. If you use an XA-compliant transaction manager as the coordinator, you also need XIDs associated LUWIDs. After gathering the necessary information from the coordinator, you can use the command shown in Example 6-6 to resolve the situation.

Example 6-6 Recover in-doubt threads

```
-RECOVER INDOUBT ACTION(COMMIT|ABORT) LUWID(XXX)
```

You might also have to use the command shown in Example 6-7 to purge in-doubt thread information in the following situations:

- For threads where DB2 has a coordinator responsibility that it cannot fulfill because of participant cold start, sync point protocol errors, or in-doubt resolution protocol errors.
- For threads that were in-doubt but were resolved with the RECOVER INDOUBT command, subsequent re-synchronization with the coordinator shows heuristic damage. The RESET column of a DISPLAY THREAD TYPE(INDOUBT) report indicates whether information must be purged.

Example 6-7 Reset in-doubt threads

```
-RESET INDOUBT LUWID(XXX)
```

For details, also see *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840.

6.6 DB2 drivers and packages

For distributed connections, DB2 uses a set of packages common to CLI, .NET, and JDBC drivers all using the same bind options. By default, KEEPDYNAMIC is not enabled, and APPLICATION ENCODING SCHEME is set using EBCDIC even for drivers running on LUW. When specifying non-default option, the drivers allow some of the options to be changed, such as KEEPDYNAMIC. Some other changes are not allowed.

6.6.1 Terminology for DB2 connectivity

Because of recent innovations for application platforms and DB2 connectivity, the names of DB2 connectivity functions and layers can be confusing. Table 6-3 shows the relation between DB2 clients versions and the function names. For more information, see the developerWorks® article, *Which DB2 9.5 client connectivity option is right for you?* at:

http://www.ibm.com/developerworks/db2/library/techarticle/dm-0804zikopoulos/

Table 6-3 DB2 clients versions and names of functions

DB2 8.2 clients	DB2 9 clients	DB2 9.5 clients
DB2 administration client DB2 application development client	DB2 client	IBM Data Server Client

DB2 8.2 clients	DB2 9 clients	DB2 9.5 clients
DB2 run-time client	DB2 runtime client	IBM Data Server Runtime Client
Java common client	IBM DB2 Driver for JDBC and SQLJ	IBM Data Server Driver for JDBC and SQLJ
N/A	IBM DB2 Driver for ODBC and CLI	IBM Data Server Driver for ODBC and CLI
N/A	N/A	IBM Data Server Driver for ODBC, CLI and .NET (for Windows only)

6.6.2 JDBC structure

Java Database Connectivity (JDBC) is an application programming interface (API) defined by Sun™ Microsystems. It provides connectivity to relational databases from Java applications. Java Developers Kit (JDK™) provides JDBC, which is composed of the JDBC driver manager and the JDBC drivers. The JDBC driver manager is a common layer between the Java applications and the specific JDBC driver, and it manages connections between applications and the DBMS.

Figure 6-5 shows the JDBC driver architecture.

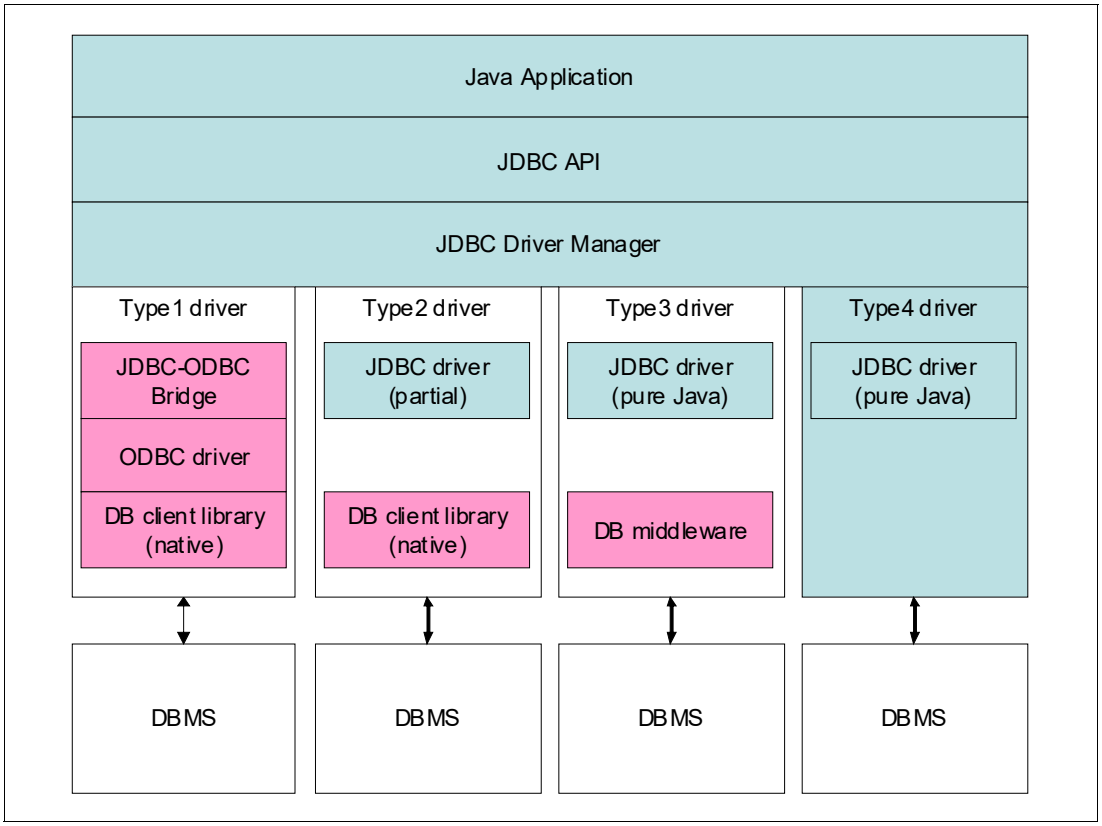

Figure 6-5 JDBC driver architecture

According to the JDBC specification, there are four types of JDBC driver architectures:

Type 1 Drivers that implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability. DB2 does not support a type 1 driver.

Type 2 Drivers that are written partly in the Java programming language and partly in native code. The drivers use a native client library specific to the data source to which they connect. Because of the native code, their portability is limited.

Type 3 Drivers that use a pure Java client and communicate with a database using a database-independent protocol. The database then communicates the client's requests to the data source. DB2 does not support a type 3 driver.

Type 4 Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

DB2 supports a driver that combines type 2 and type 4 JDBC implementations, which is now called IBM Data Server Driver for JDBC and SQLJ.

6.6.3 IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ, formerly known as IBM Driver for JDBC and SQLJ, is a single driver that includes JDBC type 2 and JDBC type 4 behavior. When an application loads the IBM Data Server Driver for JDBC and SQLJ, a single driver instance is loaded for type 2 and type 4 implementations. The application can make type 2 and type 4 connections using this single driver instance. The type 2 and type 4 connections can be made concurrently. To use the IBM Data Server Driver for JDBC and SQLJ, you need Java 2 Technology Edition, SDK 1.4.2 or higher.

Important: If you use the type 4 driver, because it is implemented as a DRDA AR, you can access DB2 for z/OS as well, without having to go through DB2 Connect. However, from a licensing point of view, you still need a DB2 Connect licence to be able to connect to a DB2 for z/OS using the type 4 driver.

The type 4 driver provides direct connectivity to DB2 for z/OS (or any DRDA AS for that matter) with functions comparable to DB2 Connect.

Table 6-4 shows the drivers and supported types for JDBC.

Table 6-4 Driver name, packaged file name, and driver type mapping

Driver name	Packaged as	Driver Type
IBM Data Server Driver for JDBC and SQLJ	► db2jcc.jar and sqlj.zip for JDBC 3.0 support ► db2jcc4.jar and sqlj4.zip for JDBC 4.0 support db2jcc_license_cisuz.jar is a license file.	Type 2 and 4
DB2 JDBC Type 2 Driver for Linux, UNIX and Windows	db2java.zip	Type 2

Notes:

- The IBM Data Server Driver for JDBC and SQLJ is also called the *JCC driver*, and was formerly known as the *IBM Universal Database Driver*.
- The DB2 JDBC Type 2 Driver for Linux, UNIX, and Windows is deprecated. It is also called the *CLI legacy driver*.

6.6.4 JDBC API overview

Figure 6-6 illustrates the relationship among JDBC API interfaces and classes.

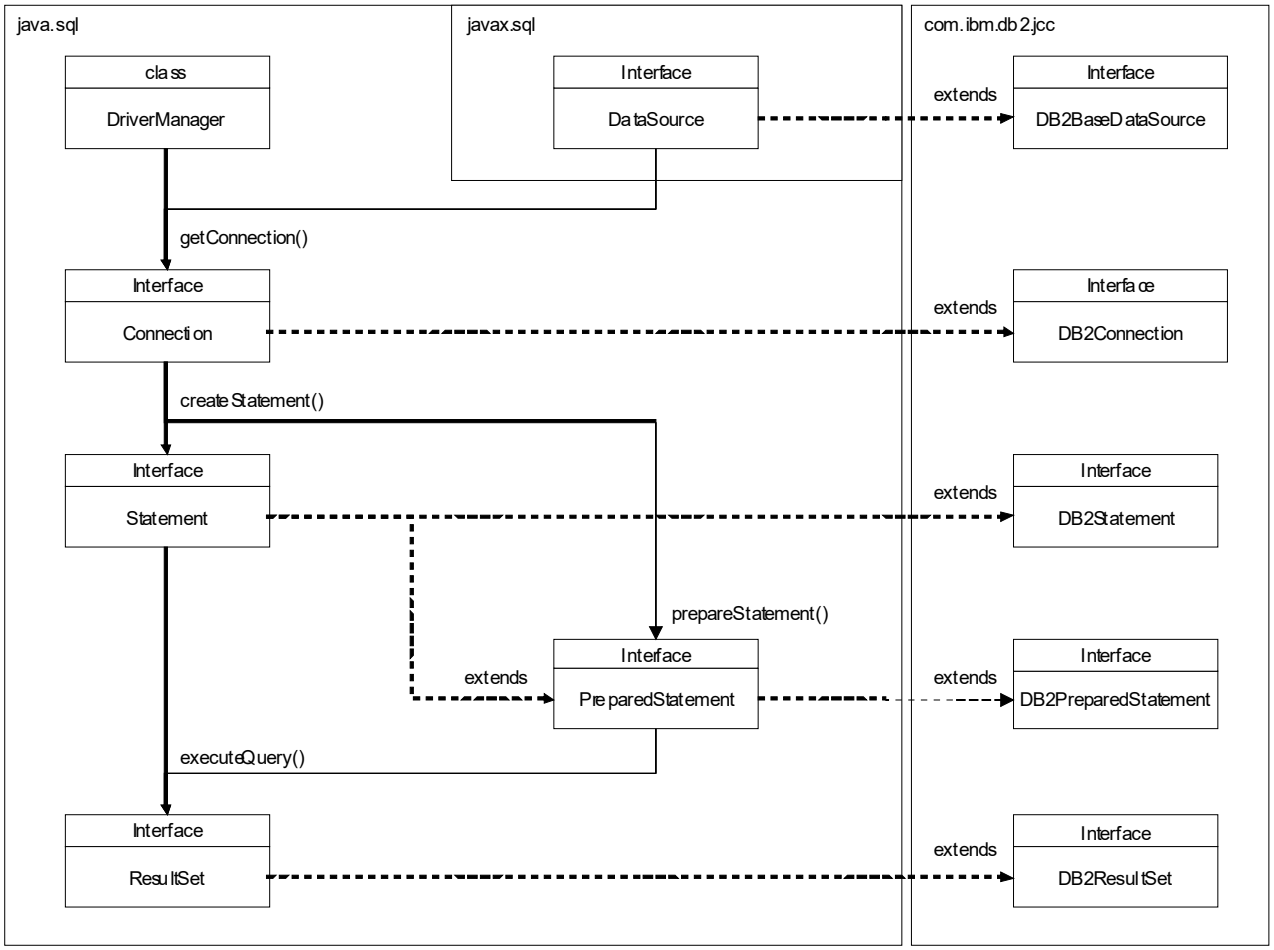

Figure 6-6 Relationship among JDBC API interfaces and classes

DriverManager The basic service for managing a set of JDBC drivers.

DataSource A factory for connections to the physical data source that this DataSource object represents. An alternative to the DriverManager facility, a DataSource object is the preferred means of getting a connection. An object that implements the DataSource interface is typically registered with a naming service based on the Java Naming and Directory (JNDI) API.

Connection A connection with a specific database. SQL statements are executed and results are returned within the context of a connection.

Statement	The object used for executing a SQL statement and returning the results it produces.
PreparedStatement	An object that represents a precompiled SQL statement. An SQL statement is precompiled and stored in a PreparedStatement object. This object can then be used to efficiently execute this statement multiple times.
ResultSet	A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

IBM Data Server Driver for JDBC and SQLJ provides properties and methods specific to DB2 by extending these standard JDBC interfaces.

6.6.5 Packages used by DB2 drivers

IBM Data Server Driver for JDBC and SQLJ includes the utility called DB2Binder. The DB2Binder utility binds the DB2 packages that are used at the database server by the IBM Data Server Driver for JDBC and SQLJ, and grants EXECUTE authority on the packages to PUBLIC. Optionally, the DB2Binder utility can rebind DB2 packages that are not part of the IBM Data Server Driver for JDBC and SQLJ. Example 6-8 shows utility usage.

Example 6-8 DB2Binder utility

```
// DB2Binder binds packages to the target specified by the following:
// IP address or domain name : wtsc63.itso.ibm.com
// port number : 12347
// location : DB9A
java com.ibm.db2.jcc.DB2Binder -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A
-user paolor1 -password xxxxxxxx
```

You can use the DB2Binder utility in z/OS because it does not need the DB2 command line processor (CLP). This is one of the reasons that the DB2Binder utility is introduced.

Both the Legacy CLI-based JDBC driver and IBM Data Server Driver for JDBC and SQLJ type2/4 connectivity use the same packages. They are also called DB2 CLI packages and documented in *DB2 Version 9.5 for Linux, UNIX, and Windows Call Level Interface Guide and Reference, Volume 2.* For a list of them, see Table 6-5.

Table 6-5 DB2 drivers packages

Package name	Description
SYSSHxyy	dynamic placeholders - small package WITH HOLD
SYSSNxyy	dynamic placeholders - small package NOT WITH HOLD
SYSLHxyy	dynamic placeholders - large package WITH HOLD
SYSLNxyy	dynamic placeholders - large package NOT WITH HOLD
SYSSTAT	common static CLI functions

- ‘S’ represents a “small” package with 65 sections, and ’L’ represents a “large” package with 385 sections.
- ‘H’ represents WITH HOLD and ‘N’ represents NOT WITH HOLD.
- ‘x’ is the isolation level: 0=NC, 1=UR, 2=CS, 3=RS, 4=RR.
- ‘yy’ is the package iteration 00 through FF.

For example, for the dynamic package, consider these names:

- SYSSN100: A package with 65 sections where all cursor declarations are for non-held cursors. Bound with isolation level UR. This is the first iteration of that package.
- SYSLH401: A package with 385 sections where all cursor declarations are for held cursors. Bound with isolation level RS. This is the second iteration of that package.

SYSSTAT is also used for a part of processing stored procedures.

6.7 Considerations for distributed packages

In this section we discuss considerations for resolving issues related to distributed packages.

6.7.1 Identifying obsolete DB2 Connect packages

Hundreds of packages can be accumulated in the DB2 for z/OS catalog by successive versions of the product.

The package names follow a naming convention and it is possible to relate a package to a certain DB2 Connect release by package name.

For non-CLI packages, the package name starts with ‘SQL%'. The package name is an 8-byte long string and the naming convention conforms to the following rules:

- The first 3 bytes are ‘SQL’.
- The 4th byte is the DB2 internal component code.
- The 5th byte is the DB2 internal file identifier.
- The 6th byte is the DB2 release code.
- The last 2 bytes (7th-8th) are the sequence number.

As illustrated in the following string sequence:

```
"sql" <1-char-component-code> <1-char-file-identifer> <1-char-major-release-code>
<2-char-sequence-string>
```

The 6th byte in the package name indicates which DB2 Connect release this package belongs to.

The 1-char-major-release-code can assume the values listed in Table 6-6.

Table 6-6 DB2 Connect release code

Release code	Recognition character
V7	D
V8	E
V9	F
V95	G

If there is a package change due to a fixpack, then the 2-char-sequence-string (the last 2 bytes) are changed.

For example, the package name ‘SQLABD01’ indicates that this package is a V7 package as the 6th byte is ‘D’.

The names of packages created prior to V7 still start with ‘SQL%’ but do not follow the naming convention.

You can use the naming convention to determine which package is associated with which DB2 Connect release and decide to remove the package from the system by issuing a FREE PACKAGE command. Keep in mind that once the package is freed, the package no longer exists in the DB2 for z/OS catalog and existing applications might be impacted.

An example of a query for determining the DB2 Connect versions is listed in Example 6-9.

Example 6-9 Determining the DB2 Connect version

```
SELECT
CASE
 WHEN SUBSTR(NAME,6,1) = 'D' THEN 'VERSION 7  SQL CONNECT'
 WHEN SUBSTR(NAME,6,1) = 'E' THEN 'VERSION 8  SQL CONNECT'
 WHEN SUBSTR(NAME,6,1) = 'F' THEN 'VERSION 9  SQL CONNECT'
 WHEN SUBSTR(NAME,6,1) = 'G' THEN 'VERSION 95 SQL CONNECT'
 ELSE 'VERSION UNDETERMINED'
END CASE
,SUBSTRING(NAME,1,8,OCTETS)
,VALID
,OPERATIVE
FROM
SYSIBM.SYSPACKAGE
WHERE
NAME LIKE 'SQL%'
ORDER BY 1,2
```

CLI packages generally do not change between fixpacks, even though the manuals suggest to rebind the package for each fixpack. There are 49 CLI packages currently created by db2clipk.bnd and db2clist.bnd bind files.

6.7.2 SQLCODE=-805 error with CLI packages

By default, each CLI package that is automatically bound has three packages for each characteristic, that is, small versus large, WITH HOLD versus NOT WITH HOLD, and isolation level. For instance, SYSLH2yy has three packages, SYSLH200, SYSLH201, and SYSLH202.

In some cases, you might need to create additional DB2 CLI packages so that they contain more than the default number of dynamic sections. A dynamic section is the actual executable object needed to satisfy a dynamic SQL request. Each statement handle occupies one section within a DB2 CLI package. The default upper limit is 1344 statements for each connection:

```
1344 = (64 + 384) * 3
64    sections available in small package
384   sections available in large package
3     default number of packages for each characteristic
```

Your application might require more than the default number of sections. If you have an application that under heavy load has allocated more statement handles than there are sections, your case falls into this scenario and you get an SQLCODE=-805 error as shown in Example 6-10. Note that SYSLH203, which is not bound, is reported.

Example 6-10 SQLCODE=-805 error

```
com.ibm.db2.jcc.a.SqlException: DB2 SQL Error: SQLCODE=-805, SQLSTATE=51002,
SQLERRMC=NULLID.SYSLH203 0X5359534C564C3031, DRIVER=3.51.90
```

Example 6-11 shows how to create more DB2 CLI packages. You can also use DB2Binder instead of DB2 CLP.

Example 6-11 Enable six DB2 CLI packages in total

```
-- Enable 6 packages in total for each characteristic
db2 BIND @ddcsmvs.lst BLOCKING ALL GRANT PUBLIC CLIPKG 6

-- can use DB2Binder instead of DB2 CLP
java com.ibm.db2.jcc.DB2Binder -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A
-user paolor1 -password xxxxxxxx -size 6
```

Note that this problem also occurs if applications do not close their statement handles properly. In practice, we recommend that you check application logic at first.

6.7.3 Using packages with special bind options

Only a subset of bind options are supported by default when binding DB2 CLI packages. Because DB2 CLI packages are used by DB2 CLI, ODBC, JDBC, OLE DB, .NET, and ADO applications, any changes made to the DB2 CLI packages affect applications of all these driver types.

You can bind DB2 CLI packages with some bind options for special application requirements. To create DB2 CLI packages with bind options that are not supported by default, specify the COLLECTION bind option with a collection ID that is different from the default collection ID, NULLID. Any bind options specified are then accepted, however, they might not be valid for all drivers.

To create a DB2 CLI packages with the KEEPDYNAMIC YES bind option, which is not specified by default, but perfectly valid, issue the command in Example 6-12.

Example 6-12 Bind DB2 CLI package with KEEPDYNAMIC YES

```
// Using DB2 CLP
db2 bind @ddcsmvs.lst collection NEWCOLID KEEPDYNAMIC YES

// Using DB2Binder
java com.ibm.db2.jcc.DB2Binder -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A
-user paolor1 -password xxxxxxxx -collection NEWCOLID -keepdynamic YES
```

In order for CLI/ODBC applications to access the DB2 CLI packages created in the new collection, set the CurrentPackageSet CLI/ODBC keyword in the db2cli.ini initialization file to the new collection ID as Example 6-13 shows. For more information, see *DB2 Version 9.5 for Linux, UNIX, and Windows Call Level Interface Guide and Reference, Volume 2,* SC23-5845.

Example 6-13 db2cli.ini file

```
[DB9A]
DBALIAS=DB9A
UID=PAOLOR1
DESCRIPTION=DB9A
; use schema NEWCOLID instead of default NULLID
CURRENTPACKAGESET=NEWCOLID
```

For JDBC applications, use the SET CURRENT PACKAGESET statement or JDBC driver property. See Example 6-14. For more information about JDBC driver properties, see *DB2 for Linux, UNIX, and Windows: Developing Java Applications*, SC23-5853.

Example 6-14 currentPackageSet property

```
// new driver instance
Class.forName("com.ibm.db2.jcc.DB2Driver").newInstance();

// set Properties
Properties props = new Properties();
props.setProperty("user", "PAOLOR1");
props.setProperty("password", "xxxxxxxx");
// set “NEWCOLID” to currentPackageSet
props.setProperty("currentPackageSet", "NEWCOLID");

// connect using type 4 driver
System.out.println("connect using type 4 driver");
Connection con = DriverManager.getConnection(
                 "jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A",
                 props);
```

6.7.4 Dynamic SQL statement caching

All JDBC applications use dynamic SQL. For efficient processing of dynamic SQL statements, you should use dynamic SQL statement caching.

In DB2 for z/OS, there are two types of SQL statement caching, local and global:

- Local dynamic statement cache is allocated in the storage of each thread in the DBM1. This is enabled by binding packages with KEEPDYNAMIC(YES). This cache, called *statement text cache,* is the character string form of the statement and is kept across commits, but a cache called *prepared statement cache*, which is the executable form of the statement, is not kept without enabling global dynamic statement cache. With only local dynamic statement cache, preparing the executable form from the statement text is virtually needed each time the prepared statement cache is invalidated by commit.
- Global dynamic statement cache is allocated is the EDM pool in the DBM1. You can enable this cache by setting CACHEDYN=YES in DSNZPARM. If it is activated, the prepared statement cache is kept in the global dynamic statement cache inside the EDM pool. Only one prepared statement cache of the same statement (matching text) is kept. An LRU algorithm is used for replacement. With global dynamic SQL cache, the prepared statement cache can be copied from the global cache into the thread’s storage. This is called a *short prepare* and is more effective than local dynamic SQL cache only.

Table 6-7 shows the summary of effects from KEEPDYNAMIC and CACHEDYN. For more information about dynamic SQL statement caching in *z/OS and OS/390®*, see *Squeezing the Most Out of Dynamic SQL with DB2 for z/OS and OS/390*, SG24-6418.

Table 6-7 Summary of effects from KEEPDYNAMIC and CACHEDYN

KEEPDYNAMIC	CACHEDYN	Effect
YES	YES	► Prepared statement cache cached in EDM pool ► Prepare only at first, otherwise short prepares ► Prepared statements kept across commits - avoided prepares ► Statement strings kept across commits
YES	NO	► No prepared statement cache cached in EDM pool ► Need prepares ► No prepared statements kept across commits ► Statement strings kept across commits
NO	YES	► Prepared statement cache cached in EDM pool ► Prepare only at first, otherwise short prepares ► No prepared statements kept across commits ► No statement strings kept across commits
NO	NO	► No prepared statement cache cached in EDM pool ► Need prepares ► No prepared statements kept across commits ► No statement strings kept across commits

6.7.5 Accounting information to manage remote clients

With connection pooling in application servers, all of the applications share the same configuration of connection to DB2. From the point of view of accounting in DB2, this means that all applications have the same PLANNAME, MAINPACK, and CORRNAME, and you cannot distinguish between applications. This is especially significant for JDBC applications because DB2 identifies the JDBC driver itself as a business application (with type 4 connectivity) and records this information to accounting IFC records.

JDBC drivers provide properties for accounting use. These can be useful for auditing and performance monitoring. Example 6-15 shows how to set these properties.

Example 6-15 Set accounting information using JDBC connection properties

```
// new driver instance
Class.forName("com.ibm.db2.jcc.DB2Driver").newInstance();

// set Properties
Properties props = new Properties();
props.setProperty("user","PAOLOR1");
props.setProperty("password","xxxxxxxx");

// Accouting infomation
props.setProperty("clientUser","NORI");
props.setProperty("clientApplicationInformation","TestJdbc");
props.setProperty("clientWorkstation","NSJTOO1");

// connect using type 4 driver
System.out.println("connect using type 4 driver");
Connection con = DriverManager.getConnection(
```

```
                "jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A",
                props);
```

Because these properties can be changed in the same connection, you can pass the accounting information to DB2 for z/OS even with connection pooling. With WebSphere Application Server V6 or earlier, WSConnection supports the setting of these properties. You can set accounting information before starting each transaction to overwrite the default. See Example 6-16.

Example 6-16 Set accounting information using WSConnection properties

```
import com.ibm.websphere.rsadapter.WSConnection;

.....

try {
  InitialContext ctx = new InitialContext();
  // Perform a naming service lookup to get the DataSource object.
  DataSource ds = (javax.sql.DataSource)ctx.lookup("java:comp/jdbc/myDS");
} catch (Exception e) {;}

WSConnection con = (WSConnection) ds.getConnection();
Properties props = new properties();
props.setProperty(WSConnection.CLIENT_ID, "NORI");
props.setProperty(WSConnection.CLIENT_LOCATION, "NSJT001");
props.setProperty(WSConnection.CLIENT_APPLICATION_NAME, "TestJdbc");
con.setClientInformation(props);
con.close();
```

6.8 SQLJ

SQLJ provides support for embedded static SQL in Java applications. SQLJ was initially developed by IBM, Oracle®, Compaq, Informix®, Sybase, Cloudscape, and Sun Microsystems to complement the dynamic SQL JDBC model with a static SQL model. It is now accepted as ANSI standard and known as SQL/OLB (Object Language Bindings).

Besides simplicity, the static SQL model has many advantages in comparison with the dynamic SQL model.

From the point of view of security:

- The user of an application does not need authority on the tables/views.
- It helps guard the data from access outside the application layer.
- It prevents SQL injection.

From the point of view of performance:

- It eliminates access path changes at runtime.
- No prepare or dynamic statement cache tuning is needed.
- Performance fallback with package versioning is possible.
- Standard accounting class data is available and able to distinguish by MAINPACK.

For details, see the paper *Considering SQLJ for Your DB2 V8 Java Applications*, by Connie Tsui, available from:

http://www.ibm.com/developerworks/db2/library/techarticle/0302tsui/0302tsui.html

6.8.1 Interoperability between SQLJ and JDBC

In general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL. However, because SQLJ functions are implemented on the basis of JDBC architecture and can inter-operate with JDBC, an application program can use JDBC and SQLJ within the same UOW.

For example, SQLJ ConnectContext and JDBC connection are mutually exchangeable. See Example 6-17.

Example 6-17 Translate connection between JDBC and SQLJ

```
// Get JDBC connection from SQLJ connection context
Connection con = ctx.getConnection();
// or
Connection con = DefaultContext.getDefaultContext().getConnection();

// Get SQLJ connection context from JDBC connection
Connection con = DriverManager.getConnection(...);
DefaultContext ctx = new DefaultContext(con);
```

You can also translate JDBC ResultSet into SQLJ iterator and vice versa. See Example 6-18.

Example 6-18 Translate JDBC ResultSets and SQLJ iterator

```
// Get SQLJ iterator from JDBC ResultSet
FooIterator iter;
ResultSet rs = stmt.executeQuery();
#sql iter = { CAST :rs};

// Get JDBC ResultSet from SQLJ iterator
ResultSet rs = iter.getResultSet();
```

6.8.2 Program preparation for SQLJ

To prepare SQLJ programs for execution, you use commands to translate SQLJ source code into Java source code, compile the Java source code, create and customize SQLJ serialized profiles, and bind DB2 packages. SQLJ serialized profiles contain embedded SQL statements and play almost the same role as the bind file described in 6.2.4, “DB2 for Linux, UNIX, and Windows to z/OS using DB2 Connect” on page 109.

Figure 6-7 shows program preparation flow for SQLJ programs.

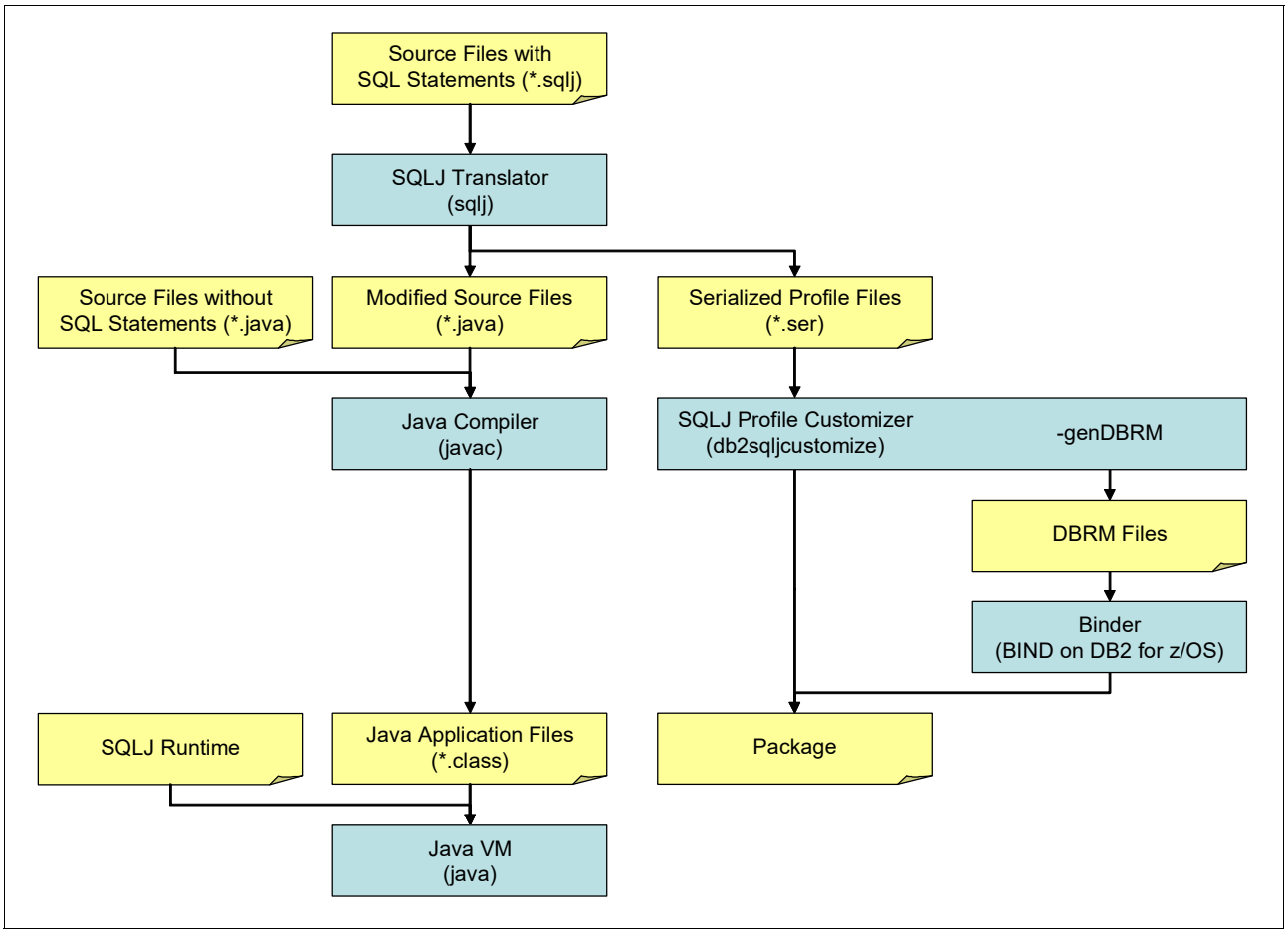

Figure 6-7 Program preparation for SQLJ

SQLJ translator

You invoke the SQLJ translator (sqlj command) to read the SQLJ source file and check the SQLJ syntax in the program. The translator generates a a Java source file and SQLJ serialized profiles, and optionally compiles the generated Java source into byte codes (by default) if there are no errors. The generated Java source file replaces embedded SQL with calls to the SQLJ runtime, which executes the SQL operations.

Here are some useful command options:

`-linemap` Specifies whether line numbers in Java exceptions match line numbers in the SQLJ source file (the .sqlj file), or line numbers in the Java source file that is generated by the SQLJ translator (the .java file). Use **`yes`** if you need them in the SQLJ source file.

`-db2optimize` Specifies that the SQLJ translator generates code for a connection context class that is optimized for DB2. **`-db2optimize`** optimizes the code for the user-defined context but not the default context. Use **`yes`**.

For typical command usage, see Example 6-19.

Example 6-19 sqlj command usage

```
> sqlj -db2optimize -linemap TestSqlj.sqlj
[Translating 1 files.]
[Reading file TestSqlj]
[Translating file TestSqlj]
[Compiling 1 Java files.]
[Instrumenting 1 class files.]
[Instrumenting file TestSqlj.class from TestSqlj.java.]
[Mapped 40 line positions]
```

You can specify your preferred options in the sqlj.properties file so that you do not need to specify all of the options every time you issue the SQLJ command. See Example 6-20.

Example 6-20 sqlj.properties sample

```
# '#'comments out the line.
sqlj.status
sqlj.linemap=YES
#sqlj.ser2class
sqlj.db2optimize
#sqlj.compile=false
```

SQLJ profile customizer

The SQLJ profile customizer (**db2sqljcustomize** command) creates a DB2 customization for the generated serialized profile. By default, the customizer performs an online check for SQL statements that can be dynamically prepared. This check performs syntax, semantic, and schema validation. Also, by default, it calls the SQLJ profile binder to bind the DB2 packages.

Useful command options for DB2 for z/OS are as follows:

-onlinecheck Use YES. This verifies that objects exist at the server, and enables data type matching between the application and server to allow indexes to be considered on predicate data type mismatches.

-automaticbind Use the option that fits your environment. This specifies whether the customizer binds DB2 packages at the data source that is specified by the **-url** parameter. The default is YES. The number of packages and the isolation levels of those packages are controlled by the **-rootpkgname** and **-singlepkgname** options.

-collection Do not omit this option to avoid using the default NULLID for SQLJ application's packages. Treat them as normal packages.

-qualifier Used by online checking to qualify any unqualified tables and views. Not used during the bind; the QUALIFIER specified in the bind options is what the bind uses for this. In other words, put it in both places.

-longpkgname Longer package names and upper/lower case package names fit in the java class names. The goal is to keep your executable names and .class file names in order and unique.

-pkgversion Use AUTO to automatically generate a version name that is based on the consistency token.

-staticpositioned Use YES. Positioned updates and deletes are executed statically now.

-storebindoptions Allows you to execute the db2sqljbind utility to different environments and provides consistent bind options in those environments.

-bindoptions Specifies a list of bind options for DB2 for z/OS, separated by spaces.

The **db2sqljcustomize** command usage is shown in Example 6-21.

Example 6-21 db2sqljcustomize usage

```
// bind a SQLJ package to the target specified by the following:
// IP address or domain name: wtsc63.itso.ibm.com
// port number : 12347
// location : DB9A
> db2sqljcustomize -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A -user paolor1
-password xxxxxxxx -staticpositioned YES -bindoptions "DBPROTOCOL(DRDA) DEGREE(1)
EXPLAIN(YES) ISOLATION(CS) OWNER(FXFQ) QUALIFIER(QA) RELEASE(COMMIT) REOPT(NONE)
SQLERROR(NOPACKAGE) VALIDATE(BIND) CURRENTDATA(NO) COLLECTION(FC) ACTION(REPLACE)"
TestSqlj_SJProfile0.ser

[ibm][db2][jcc][sqlj] Begin Bind
[ibm][db2][jcc][sqlj] Loading profile: TestSqlj_SJProfile0
[ibm][db2][jcc][sqlj] User bind options: DBPROTOCOL(DRDA) DEGREE(1) EXPLAIN(YES)
ISOLATION(CS) OWNER(FXFQ) QUALIFIER(QA) RELEASE(COMMIT) REOPT(NONE) SQLERROR(NO
PACKAGE) VALIDATE(BIND) CURRENTDATA(NO) COLLECTION(FC) ACTION(REPLACE)
[ibm][db2][jcc][sqlj] Driver defaults(user may override): BLOCKING ALL
[ibm][db2][jcc][sqlj] Fixed driver options: DATETIME ISO DYNAMICRULES BIND
[ibm][db2][jcc][sqlj] Binding package TestSqlj_SJProfile0
[ibm][db2][jcc][sqlj] Bind complete for TestSqlj_SJProfile0
```

SQLJ profile printer

To view the profile contents, customized or not, you can use the SQLJ profile printer (**db2sqljprint** command) to print out the profile contents in text format. See Example 6-22.

Example 6-22 db2sqljprint command usage

```
> db2sqljprint TestSqlj_SJProfile0.ser
================================================
printing contents of profile TestSqlj_SJProfile0
created 1225323326078 (2008-10-29)
associated context is sqlj.runtime.ref.DefaultContext
profile loader is sqlj.runtime.profile.DefaultLoader@137c60d
contains 0 customizations
original source file: testSqlj.sqlj
contains 2 entries
================================================
profile TestSqlj_SJProfile0 entry 0
#sql { INSERT INTO company_a
  VALUES(5275, 'Sanders' , 20, 'Mgr'  , 15  , 18357.50)
};
line number: 35
PREPARED_STATEMENT executed via EXECUTE_UPDATE
role is STATEMENT
descriptor is null
contains 0 parameters
result set type is NO_RESULT
result set name is null
contains 0 result columns
================================================
profile TestSqlj_SJProfile0 entry 1
#sql { COMMIT };
line number: 42
```

```
PREPARED_STATEMENT executed via EXECUTE_UPDATE
role is COMMIT
descriptor is null
contains 0 parameters
result set type is NO_RESULT
result set name is null
contains 0 result columns
================================================
```

6.9 pureQuery

pureQuery, previously code-named JLINQ, is a new option for developing Java application using DB2. It provides various benefits:

- Object-Relation Mapping (ORM):

  To achieve the ORM in JDBC or SQLJ, you have to write hundreds of repetitive codes to map relational data to objects. Taking a programming language as an analogy, JDBC and SQLJ is “assembler” for database manipulation in its native level. pureQuery is a “high-level language” that encapsulate the codes mapping relational data to objects.

- Synergy with IBM Data Studio Developer:

  The ability of pureQuery is also encouraged by Data Studio Developer. You can perform the tasks with pureQuery from its Eclipse-based GUI. It also can generate a test code skeleton for Java beans using JUnit framework.

- Query in-memory objects:

  With pureQuery, you can apply SQL on basic Java objects, such as Collection. It simplifies your code to use SQL as a functional language instead of a procedural language such as plain Java.

- Supports both dynamic and static SQL:

  You can switch execution mode by setting the pureQuery runtime properties. The static SQL model provides us a more secure and efficient way for database access.

- Good performance:

  In addition to support static SQL processing, the pureQuery runtime performs the best-practice use of JDBC driver, database-specific APIs and use the most efficient JDBC options.

It is obviously out of the scope of this book to cover all the functions of pureQuery. In this section, we focus on the pureQuery API, program preparation, and the method of running static SQL with JDBC applications.

6.9.1 pureQuery architecture overview

Figure 6-8 depicts the pureQuery architecture.

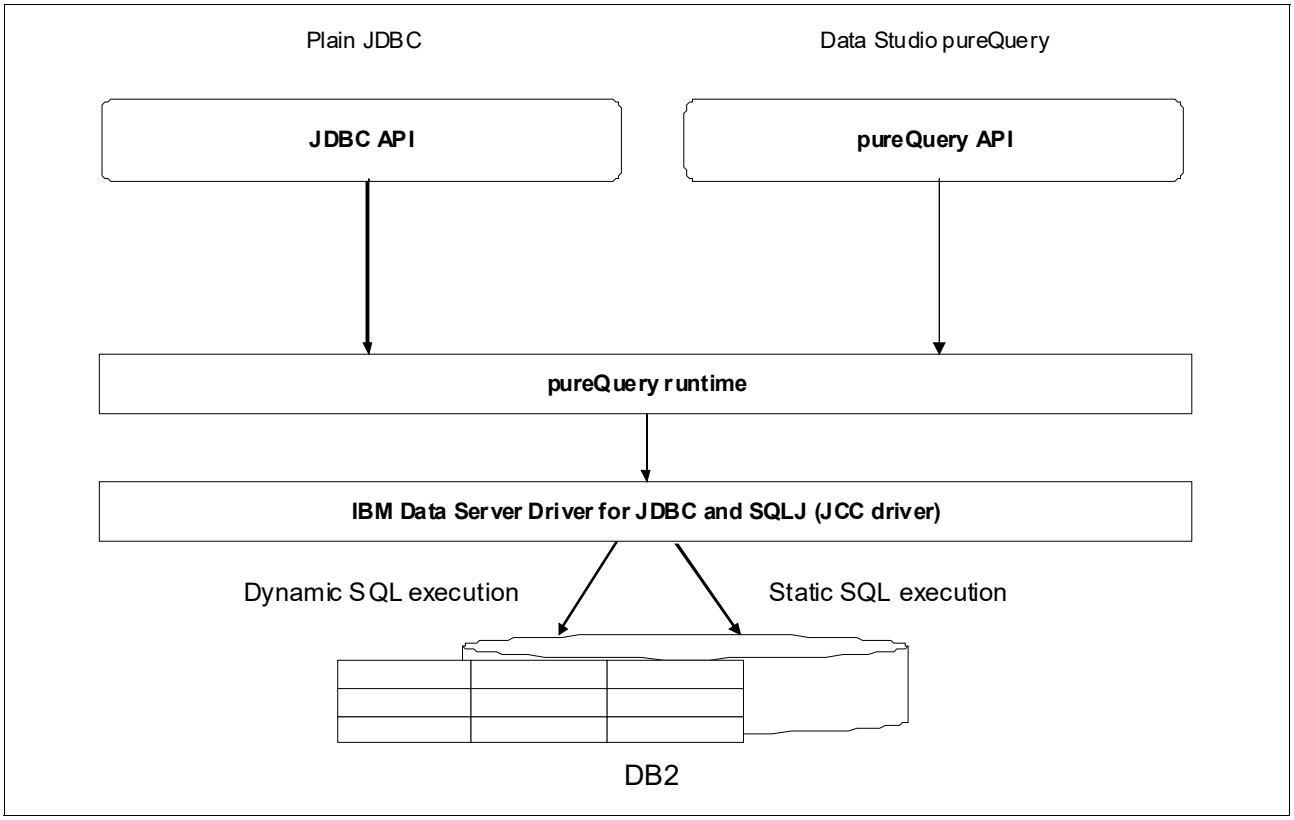

Figure 6-8 pureQuery architecture overview

pureQuery is based on JDBC architecture and IBM Data Server Driver for JDBC and SQLJ (JCC driver). The pureQuery API, which is composed of *inline-methods* and *annotated-methods*, works with the pureQuery runtime.

One of the interesting features of pureQuery is that you can choose execution of SQL statically and dynamically by configuring the pureQuery runtime properties. Applications using plain JDBC API can run their SQL statements statically with the pureQuery runtime, by capturing SQL statements from the application and binding a package to DB2.

We now look at examples of the inline-methods and annotated-methods.

6.9.2 Inline programming style

When you use the inline programming style, you can code your SQL queries, update statements, or CALL statements inline in your applications. An SQL statement appears as a parameter in the invocation of a method. These are some of the advantages of using inline methods:

- They are easy to use because they are defined already in the IBM-supplied Data interface.
- The SQL is visible within your application. Therefore, it is easy to spot errors and make corrections that do not affect any other application.
- You can create SQL statements at runtime.

Example 6-23 shows the in-line programming style.

Example 6-23 inline programming style

```
// Fetch 1 row into a beans
Employee employee =
   data.queryFirst("SELECT * FROM HRDEPT.EMP WHERE lastname = ?1",
      Employee.class, lastName);

// Fetch n rows into List of beans
List<Employee> employees =
   data.queryList("SELECT * FROM HRDEPT.EMP",
      Employee.class);

// Fetch n rows into Array of beans
Employee[] employees =
   data.queryArray("SELECT * FROM HRDEPT.EMP",
      Employee.class);

// Fetch n rows into Iterator of beans
Iterator<Employee> employees =
   data.queryIterator("SELECT * FROM HRDEPT.EMP",
      Employee.class);

// Fetch 1 row into Map of beans
// In a Map, column names become String keys
Map<String,Object> employee =
   data.queryFirst("SELECT * FROM HRDEPT.EMP WHERE lastname=?1",
      lastName);

// Fetch n row into List of Map of beans
List<Map<String,Object>> employees =
   data.queryList("SELECT * FROM  HRDEPT.EMP"
      + "WHERE lastname LIKE ?", "Br%");

// Insert a row
Employee newEmployee =
   new Employee("000010", "CHRISTINE", "I", "HAAS", "A00",
      "3978", new java.sql.Date(System.currentTimeMillis()));
int rowsAffected =
   data.update("insert into hrdept.emp “
      + ”(id, firstname, midinit, lastname, deptno, phoneext, hiredate) ”
      + “VALUES (:id, :firstName, :midInit, :lastName, :deptNo, :phoneExt,”
      + “:hireDate)",
      newEmployee);

// Update
Employee employee =
   new Employee("000010", "CHRISTINE", "I", "HAAS", "A00",
      "3978", new java.sql.Date(System.currentTimeMillis()));
int rowsAffected =
   data.update("update hrdept.emp set firstname = :firstName, “
      + “midinit = :midInit, lastname = :lastName, deptno = :deptNo, “
      + “phoneext = :phoneExt, hiredate = :hireDate “
      + “WHERE id = :id",
      employee);
```

```
// Delete
int rowsAffected = data.update("delete from hrderp.emp where id=?1", id);
```

6.9.3 Annotated-method programming style

When you use the annotated-method programming style, you can code your SQL queries, update statements, or CALL statements in annotations on methods that you define in your own interfaces. Figure 6-9 shows annotated-method and its implementation generated by the pureQuery Generator.

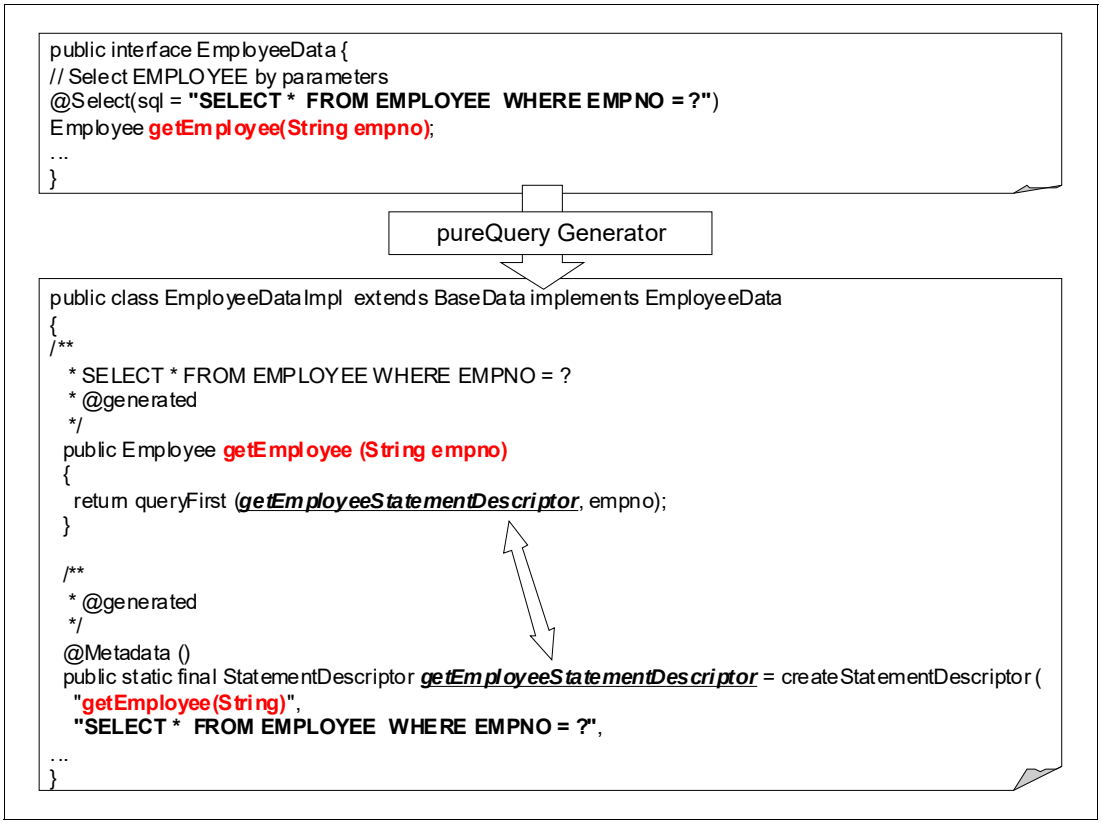

Figure 6-9 Annotated-method programming style

In the ‘EmployeeData’ interface, the logically declared mapping between ‘SELECT * FROM EMPLOYEE WHERE EMPNO = ?’ SQL statement and ‘getEmployee(String empno)’ is done via method by annotation. The pureQuery Generator translates it into the ‘EmployeeDataImpl’ class. In this class, the method is mapped to the SQL statement intermediated by ‘getEmploteeStatementDescript’ StatementDescriptor.

These are some of the advantages of using annotated methods:

- You can develop and test SQL separately from your applications.
- You can logically group SQL statements together into separate interfaces.
- You can reuse interfaces in different applications. When you reuse an interface, you can even override the SQL statements in that interface by providing SQL statements in an XML file.

6.9.4 Program preparation using pureQuery

In pureQuery, there are two ways to create Java beans mapped to databases. After beans are generated from tables and queries, you will go along the same procedure:

- Database-driven objectization (DDO):

  The most common usage of pureQuery is to select a table and to generate the objects for data access and manipulation, otherwise referred to as CRUD (create, replace, update, delete). This is where pureQuery generates the bean interfaces and classes required to perform the CRUD operations on the selected tables. It also generates a special class that can be used to unit test the objects created. In effect this is a Table-to-Java transformation where Java-centric CRUD operations are generated based on the underlying table's characteristics.

- Query-driven objectization (QDO):

  It is often the case that the process of building an application originates from having a set of database query statements (that is, SQL). This is when Java beans need to be created based on existing query statements, which can be used to build the underlying application. In effect this is an SQL-to-Java transformation where Query statements are seamlessly converted into Java-centric objects.

Figure 6-10 illustrates the program preparation flow using pureQuery.

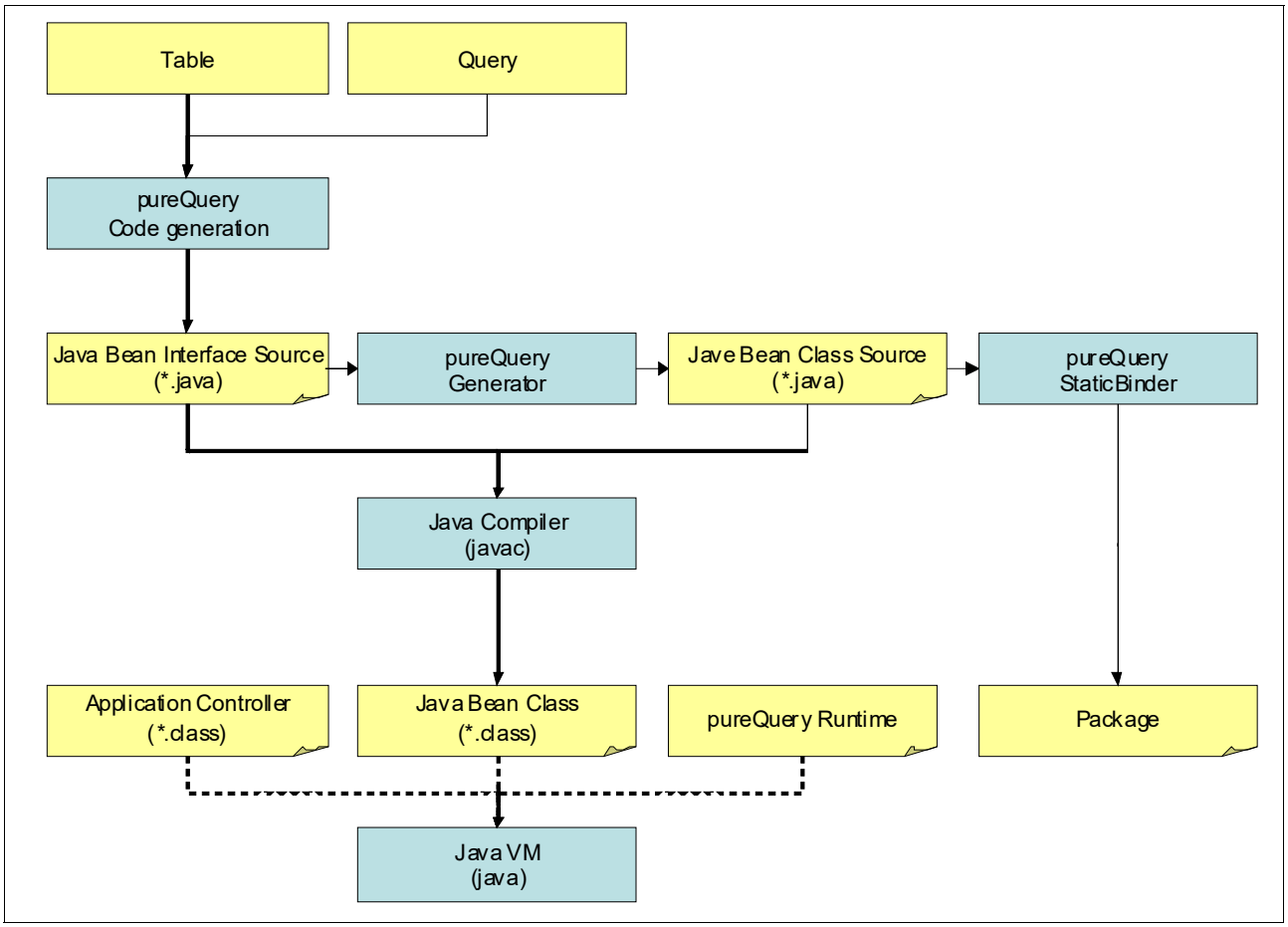

Figure 6-10 Program preparation flow using pureQuery

Code generation

This step creates Java bean interfaces that declare annotated methods from tables and queries. In the Data Studio Environment, you can activate this step from the GUI. Figure 6-11 shows the procedure.

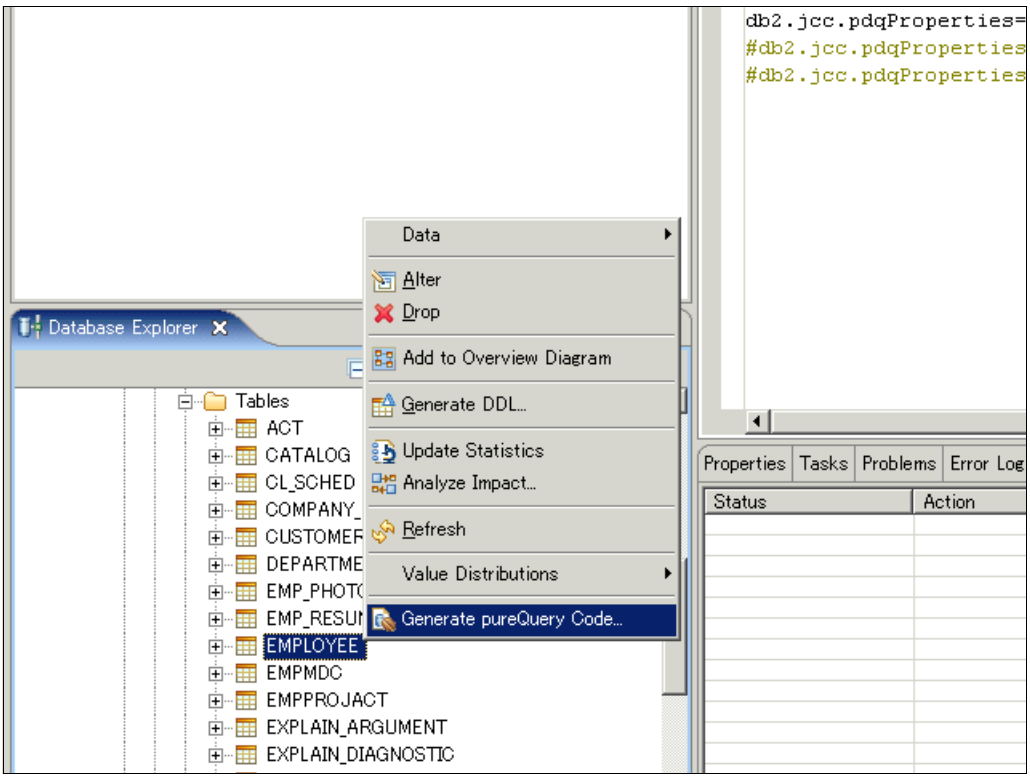

Figure 6-11 Code generation using Data Studio Developer

In Database Explorer, select and right-click a table from which you would like to create a Java bean interface and select 'Generate pureQuery Code' from the pop-up menu.

If you would like to create a Java bean interface from a query, set a cursor on the query string and right-click to select from the pop-up menu 'pureQuery' → 'Generate pureQuery Code'.

Generator

The Generator generates Java bean classes from Java bean interfaces that declare annotated methods.

The Generator analyzes the content of the input interface file and creates Java code that queries or updates the data object referenced in the @Select, @Update, and @Call annotations. It generates the code to populate the target bean on a query operation and uses the bean contents on update as input data. In Data Studio Development, the Generator is automatically invoked for all pureQuery annotated method interfaces that are contained in a Java project that has pureQuery support enabled. The generation of interface implementations happens as part of the normal build process for Java projects.

In Data Studio Environment, the Generator uses Default.genProps as an option file. You can specify the Generator options in this file. See Example 6-24.

Example 6-24 Default.genProps

```
# Provide the default properties that the pureQuery generator stores
# in generated implemenation classes
# and that the pureQuery StaticBinder uses during a bind operation.
# Here is an example:
#    defaultOptions= -collection yourCollection
# These options override the options that you specify on the defaultOptions line.

# Provide properties for specific interfaces that declare annotated methods,
# as in this example:
#    com.yourClass = -collection yourCollectionForClass

# as default
# - trace generator activity level INFO
defaultOptions = -traceFile C:\pureQueryFolder\trace.log traceLevel INFO

# for test.CompanyData
# - TESTPURE for collection id
test.CompanyData = -collection TESTPURE

# for tets.EmployeeData
# - TESTPURE for collection id
# - Use Employee_sql.xml XML file to override statements
test.EmployeeData = -collection TESTPURE -xmlFile Employee_sql.xml
```

For more information about the Generator, see also *The pureQuery Generator utility* in *Information Center for IBM Data Studio Developer*, available from:

http://publib.boulder.ibm.com/infocenter/dstudio/v1r1m0/index.jsp

StaticBinder

The StaticBinder binds the SQL statements in your annotated methods to the data source.

Before running the StaticBinder, run the Generator to generate the implementation classes for the interfaces that contain the SQL statements that you want to bind. When the Generator is run, all of the information necessary to create a static DB2 package is collected and stored in the generated implementation class. This information includes the SQL statements and all applicable data type information from both the Java class definitions and any column or parameter metadata from the target database. The information also includes the root package names and, optionally, the collection IDs and versions for the DB2 packages.

The StaticBinder reads this information from the implementation classes when creating and binding DB2 packages.

By default, the StaticBinder creates four packages, one for each of the four DB2 isolation levels. The StaticBinder identifies the isolation level by appending the following numbers to the root names of the packages. They are:

- '1' for isolation level Uncommitted Read (UR)
- '2' for isolation level Cursor Stability (CS)
- '3' for isolation level Read Stability (RS)
- '4' for isolation level Repeatable Read (RR)

If you use the isolation level option when performing a bind or specify the isolation level in the bind options string, only the package for the isolation level that you specify is created.

In Data Studio Environment, the StaticBinder uses Default.bindProps as an option file. You can specify the StaticBinder options in this file. You can use DB2 for z/OS bind options using -bindoptions. See Example 6-25.

Example 6-25 Default.bindProps for the StaticBinder

```
# Provide the default bind properties for the SQL statements in your application
# on a line that begins with defaultOptions, as in this example:
#    defaultOptions= -bindOptions "qualifier yourSchema"

# If you are developing a pureQuery application, provide bind properties
# for specific interfaces that declare annotated methods, as in this example:
#    com.yourClass = -bindOptions "qualifier yourSchemaForClass"
# These bind options override the options that you specify
# on the defaultOptions line.

# as default
defaultOptions= -bindoptions "DBPROTOCOL(DRDA) DEGREE(1) EXPLAIN(YES)
ISOLATION(UR) OWNER(PAOLOR1) QUALIFIER(PAOLOR1) RELEASE(COMMIT) REOPT(NEVER)
SQLERROR(NOPACKAGE) VALIDATE(BIND) CURRENTDATA(NO) COLLECTION(TESTPURE)
ACTION(REPLACE)"

# special for test.CompanyData
test.CompanyData= -bindoptions "DBPROTOCOL(DRDA) DEGREE(ANY) EXPLAIN(NO)
ISOLATION(UR) OWNER(PAOLOR1) QUALIFIER(PAOLOR1) RELEASE(COMMIT) REOPT(ALWAYS)
SQLERROR(NOPACKAGE) VALIDATE(BIND) CURRENTDATA(NO) COLLECTION(TESTPURE)
ACTION(REPLACE)"
```

For more information about the StaticBinder, see also *The pureQuery StaticBinder utility* in *Information Center for IBM Data Studio Developer*, available from:

http://publib.boulder.ibm.com/infocenter/dstudio/v1r1m0/index.jsp

Running pureQuery applications

To execute pureQuery code with static SQL, you must set the executionMode property to 'STATIC'.

The package that contains the static SQL must be bound to the database that the application uses. If you set the executionMode property to STATIC, but the package is not bound, pureQuery throws an exception. pureQuery does not revert to dynamic SQL.

To set the execution mode to STATIC, use one of the following methods.

- Set a JVM™ system property when invoking the JRE™, as shown in Example 6-26.

Example 6-26 Set JVM system property pdq.executionMode="STATIC"

```
java -Dpdq.executionMode="STATIC"
```

- Create a configuration file that contains the executionMode parameter as shown in Example 6-27. Data Studio Developer generates and uses a file called 'DB2JccConfiguration' in default. See Example 6-27.

Example 6-27 Sample configuration file

```
pdq.executionMode="STATIC"
```

For more information about db2.jcc.pdqProperties, see also *Syntax for setting db2.jcc.pdqProperties* in *Information center for IBM Data Studio Developer*, available from:

http://publib.boulder.ibm.com/infocenter/dstudio/v1r1m0/index.jsp

6.9.5 Converting JDBC dynamic SQL statements into static

With pureQuery, you have the option to run statically, rather than dynamically, the SQL statements that are embedded in Java applications that connect to databases by using JDBC.

Figure 6-12 illustrates the steps to enable pureQuery run SQL statically.

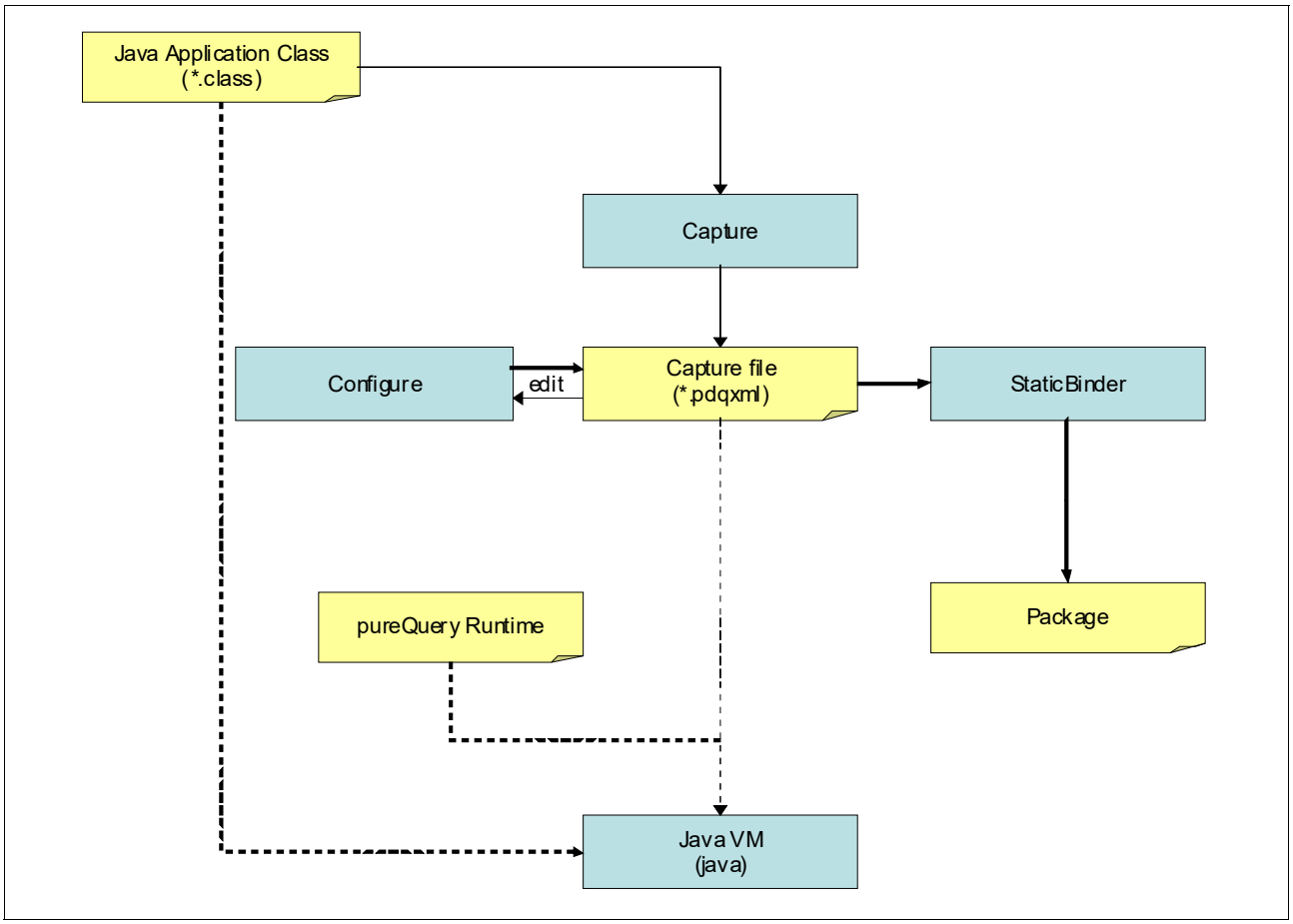

Figure 6-12 Capture, Configure, and Bind

There are four processes:

1. In the capture process, pureQuery traps SQL statements issued from running JDBC applications and records them into a capture file.
2. In the configure process, the pureQuery Configure utility specifies collection, package name, and version about the target package, and sets them to the capture file.
3. In the bind process, the pureQuery StaticBinder utility reads the capture file configured in the previous step and binds packages to the target DB2.
4. In the run process, pureQuery lets JDBC applications run captured SQL statically with the capture file when executionMode is static.

Capture

The capture process tells pureQuery which SQL statements to run statically when you run your JDBC application. In this process, you run the JDBC application in capture mode. You turn on capture mode by setting db2.jcc.pdqProperties, which is pureQuery's property for the IBM Data Server Driver for JDBC and SQLJ.

When you run your JDBC application with captureMode set to ON as shown in Example 6-28, you must run all paths that contain the SQL statements that you want to run statically. pureQuery captures SQL statements only when they are run.

Example 6-28 Specify captureMode=ON in DB2JccConfiguration.properties

```
db2.jcc.pdqProperties=captureMode(ON),executionMode(DYNAMIC),pureQueryXml(pureQuer
yFolder/capture.pdqxml),stackTraceDepth(10)
```

Captured information is placed in an XML file called a capture file. The Configure and StaticBinder utilities use capture files as input. When you run your JDBC application to execute your statements statically, pureQuery uses capture files.

If an application submits identical SQL statements from multiple places, only one instance of each identical statement is captured. Two identical SQL statements can appear in the same capture file if they differ in one or more characteristics, such as resultSetHoldability or columnIndexes.

For the most part, capture files contain information about only those SQL statements that ran successfully. However, batched statements are captured even if all statements in a batch fail to run successfully.

Configure

With the configure process, you can set the collection, version, and root package name of DB2 packages in your capture files. You can also set the maximum number of SQL statements that are allowed in a package.

The characteristics that you set are added to your capture file. When you run the StaticBinder to bind the SQL statements that are in a capture file into a DB2 package, the StaticBinder reads these characteristics.

You can specify configure options. See Example 6-29.

Example 6-29 Default.genProps for the Configure

```
# If you have enabled SQL caputuring and binding for JDBC applications
# and you are configuring the static execution of SQL statements
# in a JDBC application,
# provide bind properties for all of the SQL statements in a capture.pdqxml
# file, as in this example:
#    capture.pdqxml= -rootpkgName yourPreferredRootPkgName
# The option -rootPkgName is mandatory.
C:\pureQueryFolder\capture.pdqxml=-collection PUREQUERY -rootPkgName TestJdbcL
```

For more information about the Configure, see also *The pureQuery Configure utility* in *Information center for IBM Data Studio Developer*, available from:

http://publib.boulder.ibm.com/infocenter/dstudio/v1r1m0/index.jsp

Bind

With the StaticBinder, you can create and bind DB2 packages that contain the SQL statements in your capture files. This step is as almost same as the one in “StaticBinder” on page 135 except for the input.

Note: The StaticBinder creates one package from one XML file. If you capture all SQL statements in one XML file, only one package is created. If your package has a large number of statements and is bound with RELEASE(COMMIT), it might cause some performance degradation because of the repetition of allocation and deallocation in EDM pool and reduce the benefit of static SQL. In this scenario, you can bind the package with RELEASE(DEALLOCATE). See 11.5, “Considerations for packages with a large number of sections” on page 246 and *DB2 for z/OS: Considerations on Small and Large Packages*, REDP-4424 for details.

Run

After you create and bind DB2 packages that contain the captured SQL statements, you can run your JDBC application so that is uses static SQL.

When the application requests to run an SQL statement, pureQuery checks for:

- The existence of that statement in the capture file
- The connection that the statement must run against

To determine whether to run SQL statements statically or dynamically, pureQuery checks for these conditions:

- If the SQL statement exists in the capture file, pureQuery runs the statement statically.
- If the SQL statement does not exist in the capture file and allowDynamicSQL is TRUE in the db2.jcc.pdqProperties for the connection, pureQuery runs the statement dynamically.
- If the SQL statement does not exist in the capture file and allowDynamicSQL is FALSE in the db2.jcc.pdqProperties for the connection, pureQuery throws an exception and does not run the statement.
- If the SQL statement is a DDL statement and not a DECLARE GLOBAL TEMPORARY TABLE statement, pureQuery does not run the statement. Instead, pureQuery reports that the statement ran successfully. The exception to this rule occurs when DDL statements appear in a Statement object as part of a batch.

Example 6-30 shows a sample configuration for running.

Example 6-30 Running configuration sample

```
# the statements in capture.pdqxml are run statically when issued,
# the other statements are run dynamically.
db2.jcc.pdqProperties=executionMode(STATIC),pureQueryXml(capture.pdqxml),allowDyna
micSQL(TRUE)

# the statements in capture.pdqxml are run statically when issued,
# the other statements cause exception and are not run.
db2.jcc.pdqProperties=executionMode(STATIC),pureQueryXml(capture.pdqxml),allowDyna
micSQL(FALSE)

# All statements issued are run dynamically.
db2.jcc.pdqProperties=executionMode(DYNAMIC)
```

6.10 Private protocol

At DB2 Version 2.2 time, when the Distributed Data Facility was first introduced, the initial DB2-to-DB2 (at mainframe level) communication mechanism provided through DDF was called private protocol. It allowed a program on DB21 to access a table on a different, even geographically dispersed, DB22 using a three-part name (location.owner.table-name), effectively moving data across DB2 subsystems.

6.10.1 DRDA

DB2 Version 2.3 introduced DRDA. DRDA required an explicit connect to a remote DB2 server, but it could be used to access DB2 data on any platform and allowed static SQL statements to be executed at the DB2 server location.

This is done by binding the application program once to create a package on the requester side and once to create a package on the server side.

DB2 Version 3.1 provided the two-phase commit protocol support capability for distributed transactions.

DB2 Version 4 added the ability to call a stored procedure at a remote DB2 server.

DB2 Version 6 delivered support for DRDA access to remote DB2 objects by way of three-part names. When combined with locally-defined DB2 aliases, it effectively supports full location transparency.

6.10.2 DRDA versus private protocol

SQL applications in DB2 for z/OS could access remote data using the two protocols or methods: private protocol or DRDA protocol. However, you should use DRDA access instead of DB2 private protocol access because DRDA access has the following advantages over DB2 private protocol access:

- Compatibility:

  Private protocol is deprecated in DB2 9 for z/OS. It might be removed in a future release.

- Integration:

  DRDA access is available to all DBRMs that implement Distributed Relational Database Architecture (DRDA). Those include supported releases of DB2 for z/OS, other members of the DB2 family of IBM products, and many products of other companies.

  DB2 private protocol access is available only to supported releases of DB2 for z/OS.

- SQL compatibility:

  DRDA access allows any statement that the server can execute. DB2 private protocol access supports only data manipulation statements: INSERT, UPDATE, DELETE, SELECT, OPEN, FETCH, and CLOSE. In addition, you cannot use any syntax of an SQL statement that was introduced after DB2 Version 5. You cannot invoke user-defined functions and stored procedures or use LOBs or distinct types in applications that use DB2 private protocol access.

- Reduced network load:

  DRDA access uses a more compact format for sending data over the network, which improves the performance on slow network links. DRDA also uses fewer messages across the network.

- Reduced bind processing:

  A DBRM for statements executed by DRDA access is bound to a package at the server only once. Those statements can include PREPARE and EXECUTE, so your application can accept dynamic statements that are to be executed at the server. Binding the package is an extra step in program preparation. Queries that are sent by DB2 private protocol access are bound at the server whenever they are first executed in a unit of work. Repeated binds can reduce the performance of a query that is executed often.

- Stored procedures:

  You can use stored procedures with DRDA access. While a stored procedure is running, it requires no message traffic over the network. This reduces the biggest obstacle to high performance for distributed data.

- Scrollable cursors:

  You can use scrollable cursors if you use DRDA access.

- Savepoints:

  You can set savepoints only if you use DRDA access with explicit CONNECT statements. If you set a savepoint and then execute an SQL statement with a three-part name, an SQL error occurs.

  The site at which a savepoint is recognized depends on whether the CONNECT statement is executed before or after the savepoint is set. For example, if an application executes the statement SET SAVEPOINT C1 at the local site before it executes a CONNECT TO S1 statement, savepoint C1 is known only at the local site. If the application executes CONNECT to S1 before SET SAVEPOINT C1, the savepoint is known only at site S1.

6.10.3 The removal of private protocol

In summary, the reasons why support for private protocol is planned to be removed from DB2 after Version 9. are:

- Private protocol is only used by DB2 for z/OS.
- Networks are rarely homogeneous.
- Private protocol has not been functionally enhanced since DB2 Version 5.
- DRDA's support for data blocking and its improved performance make it the preferred vehicle for remote data access.

Private protocol has been deprecated in DB2 9 for z/OS and might be removed in a future release of DB2. Shipped in DB2 Version 8 for z/OS and DB2 9 for z/OS is a utility to assist in the conversion of packages bound with DB2 private protocol.

Important: Use DRDA because DB2 Private Protocol is deprecated and might no longer be supported in a future release of DB2 for z/OS.

DBPROTOCOL BIND option

It is no longer possible to set a default value for the DBPROTOCOL BIND option. This is always DRDA.

Restriction: BIND and REBIND commands, which use DBPROTOCOL(PRIVATE), implicitly or explicitly generate a new warning message, DSNT226I, indicating that this option is not recommended.

6.10.4 Remote generic binding

Remote generic binding is the term used to allow remote subsystems to bind packages on your local subsystem using the default collection ID DSNCOLLID.

Sample job DSNTIJSG is provided to create a generic collection ID called DSNCOLLID. By default, the step that creates DSNCOLLID is commented out in the JCL. To have DB2 create DSNCOLLID, uncomment the step:

- DSNCOLLID is new generic collection ID, intended to reduce the amount of administrative tasks required to maintain collections for remote packages.
- If you wish to allow remote generic binding, run the step in DSNTIJSG to set up the proper security for the collection.
- There is no requirement to use the generic collection ID. It is to be used at your discretion either as a temporary collection for the automatically generated remote packages, or as a permanent authorization simplification.

6.10.5 Positioning DB2 for removal of private protocol

Recommendation: We recommend that you run the DBRM conversion before running the private protocol tool, in order to lessen the chance of overlaying existing packages or deleting DBRMs. Refer to Chapter 4., “Conversion to packages” on page 49 for details on DBRM conversion.

Because you should be prepared for the eventual removal of private protocol support, DB2 for z/OS provides a three-pronged approach to help you start the move towards DRDA remote processing.

DB2 for z/OS provides:

- A catalog analysis tool to be used on DB2 V8 and DB2 9 catalogs:

  The catalog analysis tool is provided with the DB2 installation JCL in the form of a REXX program. This new program searches the system's DB2 catalog tables to determine all private protocol dependencies known to DB2 in existing bound applications. From this search a series of bind jobs that specify DBPROTOCOL(DRDA) are automatically created.
- A trace tool to be used on DB2 V8 and DB2 9 systems:

  Specific traces and records are suggested for your use. By running these traces during execution windows of opportunity, you get a record of the packages and DBRMs that are executing private protocol statements. The trace records produced would have information such as package name, DBRM name, section number, remote location name, statement type, and SQL statement before and after alias resolution. With this information, you can then create lists of packages that should be bound in preparation for DB2's elimination of Private Protocol. This is not a new trace, but rather is a performance trace of specific IFCIDs. This trace can be run on any DB2 for z/OS system.

- The removal of the capability to alter the default DBPROTOCOL value from DRDA to PRIVATE:

  Since V6 when the DRDA protocol was enhanced to support three-part name object references, either through fully named objects or aliases, there has been the capability to set the default of the DBPROTOCOL BIND option to PRIVATE protocol when binding packages or plans of applications utilizing three-part name references. Starting with DB2 9, it is no longer possible to change the default to private protocol. However, it is still possible to specify DBPROTOCOL(PRIVATE) when performing a BIND, and the request completes with a warning message.

DB2 performance trace

Because DRDA requires packages at remote sites, all programs currently using private protocol must have packages bound to the remote sites referenced in those applications. One way to determine the remote sites that are referenced in currently running programs is to run the performance trace for IFCIDs 157 and 168 to obtain this information. The START TRACE commands to start these traces is shown here:

```
START TRACE(PERFM) CLASS(30) IFCID(157,168) DEST(GTF)
```

IFCID(168) is now designated for general use. IBM Tivoli® OMEGAMON® XE for DB2 Performance Expert on z/OS (OMEGAMON PE) is enhanced to format these trace records for DB2 Version 8 as well as for DB2 9.

The trace records obtained from this command show information such as:

- The call type
- The section number
- The name of program
- The responding location
- The SQL statement before any alias resolution
- The SQL statement after alias resolution

With this information, you can see which statements are using private protocol, and which packages need to be bound to specific remote sites.

Figure 6-13 shows an edited DB2 Performance Expert Record Trace report of the IFCID 168 trace. The report shows an SQL statement that references aliases before they have been resolved. The package or plan that the statement is running under was bound with DBPROTOCOL(PRIVATE). The aliases that are referenced in the statement are EC2BTABLES and EC2BCOLUMNS.

```
PRIMAUTH INSTANCE     IFC DATA
PLANNAME              ID
-------- ------------ --- ---------------------------------------------
SYSADM   BDF2D3FF0645 168 NETWORKID:  USIBMSY   LUNAME:  SYEC2DB2  LUWS
DSNESPCS                  REQUESTING LOCATION:  'BLANK'
                          REQUESTING TIMESTAMP: N/P
                          AR NAME: 'BLANK'              PRDID: N/P
                          QW0168LL:   224
                          QW0168ST:   SELECT T.CREATOR AS TBCREATOR, T.
                                    TBNAME, C.COLNO AS COLNO, C.NAME AS
                                    C.COLTYPE AS COLTYPE, C.LENGTH AS C
                                    C.SCALE AS COLSCALE, C.NULLS AS COL
                                    FROM EC2BTABLES T, EC2BCOLUMNS C WH
                                    T.CREATOR=C.TBCREATOR AND T.NAME=C.
                                    ORDER BY 1,2,3
```

Figure 6-13 OMEGAMON PE record trace report before alias resolution

Figure 6-14 shows the resolved aliases in the SQL statement. The aliases EC2BTABLES and EC2BCOLUMNS have been replaced by SYSIBM.SYSTABLES and SYSIBM.SYSCOLUMNS, respectively.

```
PRIMAUTH INSTANCE     IFC DATA
PLANNAME              ID
-------- ------------ --- ----------------------------------------------
SYSADM   BDF2D3FF0645 168 NETWORKID:  USIBMSY   LUNAME:  SYEC2DB2  LUWS
DSNESPCS                  REQUESTING LOCATION:  'BLANK'
                          REQUESTING TIMESTAMP: N/P
                          AR NAME: 'BLANK'              PRDID: N/P
                          QW0168LL:   244
                          QW0168ST:   SELECT T.CREATOR AS TBCREATOR, T.
                                    TBNAME, C.COLNO AS COLNO, C.NAME AS
                                    C.COLTYPE AS COLTYPE, C.LENGTH AS C
                                    C.SCALE AS COLSCALE, C.NULLS AS COL
                                    FROM "SYSIBM"."SYSTABLES" T, "SYSIB
                                    "SYSCOLUMNS" C WHERE T.CREATOR=C.TB
                                    AND T.NAME=C.TBNAME ORDER BY 1,2,3
```

Figure 6-14 OMEGAMON PE record trace report after alias resolution

The private to DRDA protocol REXX tool (DSNTP2DP)

To help you convert your plans and packages from using private protocol to DRDA protocol, DB2 provides the private to DRDA protocol REXX tool, DSNTP2DP. DSNTP2DP scans the catalog and generates the necessary commands to convert all objects that have a private protocol dependency to DRDA. The generated output can then be tailored and run at a convenient time. You can use job DSNTIJPD, which is customized during migration, to invoke DSNTP2DP.

A package or plan has a remote location private protocol dependency only when the tool can extract from the catalog *remote location dependency* information that is related to a plan or package. Just having the DBPROTOCOL column of the catalog tables that manage plans and packages set to a value of ‘P’ (Private) does not mean that the plan or package has a remote location private protocol dependency. The syntax and options for the tool DSNTP2DP are shown in Figure 6-15.

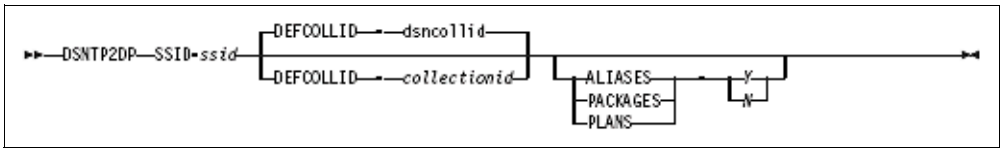

Figure 6-15 Options for tool DSNTP2DP

The tool has the following three parameters:

- Subsystem ID of DB2 to be examined (mandatory parameter) specified as SSID=ssid:

  This first parameter is mandatory so that DSNTP2DP can connect to the DB2 subsystem.

- Default collection name (optional parameter) specified as DEFCOLLID=collectionid:

  The second parameter is the default collection name to be used in the generated BIND commands where a collection name cannot be determined. If this parameter is not specified, then the tool assumes a default collection name of DSNCOLLID.

- Run options (optional parameters) ALIASES=Y/N, PACKAGES=Y/N, PLANS=Y/N:

  The run parameters trigger what processing the tool performs. If none of the run option parameters are specified, then all catalog objects are examined by the tool (that is, a value of Y is assumed for processing the plans, packages, and aliases in the catalog). If any of the run option parameters are specified and the value assigned to a particular run option is an N, then the processing for those catalog objects controlled by that run option is not performed.

There are two catalog tables that have a database protocol (DBPROTOCOL) flag, SYSPLAN and SYSPACKAGE. The catalog analysis tool uses this flag to extract the potential packages, DBRMs, or plans to be converted to DRDA from private protocol. Commands are only generated for those packages, DBRMs, or plans, that have a remote location dependency.

The output of the DSNTP2DP REXX exec with ALIASES=Y contains SQL statements to create aliases at the remote locations. You can execute these statements using DSNTEP2 or DSNTEP4, or any application that can process SQL statements, including CREATE ALIAS, CONNECT TO, RELEASE, and COMMIT.

Determine whether there are existing packages that are bound with DBPROTOCOL(PRIVATE)

Bind those packages with DBPROTOCOL(DRDA) at the correct locations. Running the DSNTP2DP exec with PACKAGES=Y, which is run as part of job DSNTIJPD, provides output that assists with this task.

Restriction: Applications that contain CONNECT statements with host variables are not handled by DSNTP2DP, so you need to create BIND statements for packages in which locations are specified in host variables.

Aliases

If the ALIASES run option is Y, then all aliases are examined. For each local package and plan that has a DBPROTOCOL set to ‘P’ and a dependency on a remote location, a command is generated to connect to the remote system and create the necessary aliases needed for these plans and packages. See Figure 6-16 for an example of the commands in the OUTALIAS dataset.

```
 ******************************** Top of Data **********
-- This file can be used as input to DSNTEP2 or DSNTEP4.
-- It must be run against the subsystem or data sharing
-- group that DSNTP2DP was run against.  The authids
-- under which the CREATE ALIAS statements will be
-- executed at the remote locations will depend on this
-- DB2's CDB setup.  Also, since these new aliases at
-- the remote locations will be local to those systems,
-- the privilege set required to issue the CREATE ALIAS
-- statement(s) must be met by the authids utilized.
  CONNECT TO DB8A ;
  CREATE ALIAS ADMF001.LI682DEPT
        FOR DSN8910.DEPT ;
  RELEASE DB8A ;
  COMMIT;
  CONNECT TO DB9B ;
  CREATE ALIAS PAOLOR3.TABLES
        FOR SYSIBM.SYSTABLES ;
  RELEASE DB9B ;
  COMMIT;
```

Figure 6-16 Sample output for aliases

Packages

If the PACKAGES run option is Y, then all the packages in the catalog are examined next. For each DBRM that is bound directly into a plan that has a remote dependency, a BIND PACKAGE command is generated that binds the DBRM as a package locally within a specified collection (or default collection DSNCOLLID) using BIND PACKAGE parameters that can be extrapolated from the current plan parameters and any parameter values from the corresponding SYSDBRM row, but verifies that DBPROTOCOL(DRDA) is specified.

The source for the DBRM is obtained from the PDSNAME column of the corresponding SYSDBRM row. The next set of generated commands is BIND PACKAGE COPY with specified collection (or default collection DSNCOLLID) specified against each server location to be accessed by the DBRM package, and the source is the package just created. In addition, as already mentioned, the binding of these packages to remote location servers also has the SQLERROR CONTINUE option specified.

Figure 6-17 shows the commands generated to create the local package and to copy the local package to the remote systems.

```
 DSN SYSTEM(DB9A)
* This file can be used as input to TSO batch.
* Note:  for each target location referenced in remote
* bind requests, a different userid (other than the one
* running the TSO batch job) may be used to access the
* target location depending on the configuration within
* this subsystem's CDB.  That userid must either have
* SYSADM authority on the target syssubsytem or it must
* have suitable privileges and authorities to bind the
* package into the collections of the target location.
 REBIND PACKAGE(LI682COLLID.LI682C) -
   DBPROTOCOL(DRDA)
 BIND PACKAGE(DB8A.LI682COLLID) COPY(LI682COLLID.LI682C) -
   OPTIONS(COMPOSITE) -
   OWNER(PAOLOR3) QUALIFIER(ADMF001) -
   DBPROTOCOL(DRDA) SQLERROR(CONTINUE)
 REBIND PACKAGE(LI682COLLID.LI682D.(V1R1M0)) -
   DBPROTOCOL(DRDA)
 BIND PACKAGE(STLEC4B.LI682COLLID) COPY(LI682COLLID.LI682D) -
   COPYVER(V1R1M0) OPTIONS(COMPOSITE) -
   OWNER(PAOLOR3) QUALIFIER(ADMF001) -
   DBPROTOCOL(DRDA) SQLERROR(CONTINUE)
```

Figure 6-17 Sample output for packages

Plans

Next, if the PLANS run option is set to Y, then the tool examines all the plans in the catalog. Regardless of the setting of the PACKAGES run option, no further packages in the catalog are examined as part of this phase of the tool's processing. Thus, this phase of the tool only generates actions to convert all the DBRMs that are bound directly into plans that have a remote dependency.

As a final step of this phase, a BIND PLAN is generated to replace the existing PLAN with a new PKLIST if none was previously present, or an updated PKLIST if one was previously present and DBPROTOCOL(DRDA) is specified. PLAN BIND PKLIST parameter must now include the local and remote collections.

Important: If you run this tool against your DBRM-based plans using private protocol, you create statements that will create packages, both locally and remotely. When the BIND PLAN statement is generated, it will not have any DBRMs bound directly into the plan, and all access will be through the packages listed in the PKLIST.

As mentioned, the tool generates the commands that should be performed to make a plan or package utilize DRDA protocols when accessing remote locations. These commands with the appropriate JCL are stored in a file that can then be tailored for the environment.

Figure 6-18 shows the commands generated to create the local packages from DBRM-based plans and to BIND the local plans with the remote packages.

```
 DSN SYSTEM(DB9A)
* This file can be used as input to TSO batch.
* Note:  for each target location referenced in remote
* bind requests, a different userid (other than the one
* running the TSO batch job) may be used to access the
* target location depending on the configuration within
* this subsystem's CDB.  That userid must either have
* SYSADM authority on the target syssubsytem or it must
* have suitable privileges and authorities to bind the
* package into the collections of the target location.
 BIND PACKAGE(DSNCOLLID) MEMBER(PGM04) -
   LIBRARY('PAOLOR3.TEMP.DBRM1') -
   OWNER(PAOLOR3) QUALIFIER(PAOLOR3) -
   ...
   DBPROTOCOL(DRDA) SQLERROR(CONTINUE)
 BIND PACKAGE(DB9B.DSNCOLLID) COPY(DSNCOLLID.PGM04) -
   OPTIONS(COMPOSITE) -
   OWNER(PAOLOR3) QUALIFIER(PAOLOR3) -
   DBPROTOCOL(DRDA) SQLERROR(CONTINUE)
 BIND PLAN(FINPRIV1) ACTION(REPLACE) RETAIN -
   ACQUIRE(USE) CACHESIZE(256) DISCONNECT(EXPLICIT) -
   OWNER(PAOLOR3) QUALIFIER(PAOLOR3) -
   VALIDATE(RUN) -
   ISOLATION(CS) -
   RELEASE(COMMIT) -
   CURRENTDATA(NO) -
  NODEFER(PREPARE) -
  DEGREE(1) -
  DYNAMICRULES(RUN) -
  REOPT(NONE) -
  KEEPDYNAMIC(NO) -
  ENCODING(37) -
  IMMEDWRITE(NO) -
  SQLRULES(DB2)  -
  PKLIST(DSNCOLLID.*,DB9B.DSNCOLLID.*) -
  DBPROTOCOL(DRDA)
```

Figure 6-18 Sample output from PLANS dataset

Complete the conversion

After you have run the DSNTP2DP REXX exec and have the SQL created for the aliases, and the bind commands necessary for both packages and plans, you execute these commands. You can use SPUFI, DSNTIAD or your favorite SQL processor to create the aliases. Run the BIND commands for the packages and plans using JCL you would use to run any other bind statements.

You might also want to replicate plans and packages that need to be converted into a development environment, run the tool there, exercise the new binds, test the code and then deliver tested scripts to production.

7

Common problems and resolutions

In this chapter we discuss some of the common problems that you might encounter when working with packages, and how you can resolve such problems. Also see Appendix A, “Useful queries” on page 259, where we have provided catalog queries to help you manage your package environment.

We cover the following topics:

- -805 errors
- Access path changes after a bind/rebind
- DB2 hint not applied
- Authorization errors
- Dynamic SQL authorization errors
- Inoperative packages

7.1 -805 errors

This is, by far, the common problem encountered when using packages. The application receives this error:

```
-805 DBRM OR PACKAGE NAME
location-name.collection-id.progname.
consistency -token NOT FOUND IN PLAN plan-name REASON reason
```

Simply stated, it means that an application program attempted to use a package 'location-name.collection-id.progname.consistency-token' that was not found.

The resolution depends on the reason code that accompanies the message:

- 01: The package name was not found because there is no package list for the plan.
- 02: The DBRM name ‘dbrm-name’ did not match an entry in the member list or the package list (it is missing in the PAKCLIST). Keep in mind that “*” in location or collid use wild cards and this error can mean it is not found *any* location and/or *any* collection.
- 03: The DBRM name ‘dbrm-name’ matched one or more entries in the package list and the search of those entries did not find the package (that is, it is present but the consistency token does not match). In this case the LOADLIB and the DBRM from which the package was created are not the same.
- 04: The package, ‘collection-id.dbrm-name.consistencytoken,’ does not exist at the remote site, ‘location-name’. This applies to remote packages only.

Tip: The most common of these are reason codes 02 and 03:

- 02 means missing in PKLIST.
- 03 means timestamp conflict.

Here is a checklist to gather the required information:

- Program name
- Information from SYSPACKLIST:

```
SELECT PLANNAME,COLLID,NAME
FROM SYSIBM.SYSPACKLIST
WHERE PLANNAME = your_plan_name;
```

- Information from execution:

```
**SQLCA :h NJDB2C..MSD07ODP.171971AA03D240F5 ESTSTBAT 03 DSNXEPM
```

- Information from SYSPACKAGE:

```
SELECT COLLID, NAME, HEX(CONTOKEN), BINDTIME,PCTIMESTAMP,PDSNAME
FROM SYSIBM.SYSPACKAGE
WHERE LOCATION = ‘ ‘
AND COLLID = your_collid
AND NAME = your_prog_name;
```

- From DBRMLIB:

  Timestamp is found in positions 25-28 of the first line of the DBRM.

  For example: DBRM timestamp in hex is 171971AA.

- From your change control software (for example, ENDEVOR from footprint):
  - Date of last compile
  - Date of last link
  - Was last BIND successful?
- From the LOAD module:
  - Load library
  - LOAD module timestamp (see Example: A-10, “CONTOKEN for a package” on page 265 for an example of obtaining this for COBOL)

These are some conclusions that you can draw from this information:

- If the HEX(CONTOKEN) from the SYSPACKAGE is greater than the CONTOKEN from the SQLCA, then the program has been bound with a later version than the load module. Check the load module to see if you are running from the right library.
- If the HEX(CONTOKEN) from the SYSPACKAGE is less than the CONTOKEN from the SQLCA, then the BIND of the package probably did not work or did not migrate properly. Investigate the BIND step. Check the CONTOKEN in the DBRM to see if the DBRM has a CONTOKEN that matches the LOADLIB.
- If the HEX(CONTOKEN) from the SYSPACKAGE matches the CONTOKEN from the SQLCA, then the package is not in the list of executable packages for this plan. Check the SYSPACKLIST and SYSPACKAGE for the allowable packages and the packages in the collections.

An example of this type of error is shown in Example 7-1. The reason code 03 indicates that PGM08 is in the PACKLIST of plan PLAN1 but the timestamps of the LOADLIB and the package do not match.

Example 7-1 -805 errors

```
NON-ZERO RETURN CODE FROM OPEN
SQLCODE = -0805
SQLTATE = 51002
SQLERRM =   DB9A..PGM08.186954941A8AE865 PLAN1 03
```

7.2 Access path changes after a bind/rebind

It is possible to have a regression in performance after a rebind if the access paths change drastically. To manage this regression, you should save the old catalog statistics or SQL before you consider making any changes to control the choice of access path. Before and after you make any changes, take performance measurements. When you migrate to a new release, evaluate the performance again. Be prepared to back out any changes that have degraded performance.

Various techniques for influencing the access path are discussed in detail in Chapter 14 of *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851.

These techniques include:

- SQL “tricks”
- Alteration of catalog statistics
- OPTHINTs

In addition, the *package stability* option allows for a seamless fallback to a previous access path. This is discussed in 10.13, “Package stability” on page 216.

IBM tools that can assist with access path change are mentioned in Appendix B.3, “Tools dealing with packages” on page 282.

An example of the access path change (before and after EXPLAIN output) is shown in Example 7-2. In this example, the access path switched from using INDEX1 to using INDEX2. You need to evaluate whether or not this is a change likely to affect your performance. See Example A.12 on page 266 for the SQL used to generate this EXPLAIN output.

Example 7-2 Access path change

```
ACC  M    IX
QNO QBK PLN# MTH CREATOR  TNAME       TABNO TYP COL CREATOR  IXNAME   XONLY
-- --- ---- --- -------- ---------- ------ --- --- -------- -------- -----
===BEFORE===
47   1    1   0 PAOLOR3  MYTABLES        1 I     1 PAOLOR3  INDEX1   N
47   1    2   3                           0       0                   N
===AFTER===
47   1    1   0 PAOLOR3  MYTABLES        1 I     2 PAOLOR3  INDEX2   N
47   1    2   3                           0       0                   N
```

7.3 DB2 hint not applied

As discussed in 10.9, “Optimization hints” on page 209, an optimization hint can be used to suggest a desired access path. Keep in mind that DB2 might not honor the request. If DB2 uses the hint you provided, it returns SQLCODE +394 from the rebind. If your hints are not found or invalid, DB2 issues SQLCODE +395.

It is important to confirm that SQLCODE +394 is returned for all packages where a hint was specified. You should incorporate this check as part of your normal bind procedures. The expectation should be that ALL hints must be honored. If they are not, they should be removed.

An example of this situation is shown in Example 7-3.

Example 7-3 DB2 hint not applied

```
DSNX105I  -DB9A REBIND SQL WARNING
           USING PAOLOR3 AUTHORITY
           PLAN=(NOT APPLICABLE)
           DBRM=PGM05
           STATEMENT=47
           SQLCODE=395
           SQLSTATE=01628
           TOKENS=13
DSNT222I  -DB9A DSNTBBP2 REBIND WARNING
           FOR PACKAGE = DB9A.DFINTEST2.PGM05.
           USE OF OPTHINT RESULTS IN
           0 STATEMENTS WHERE OPTHINT FULLY APPLIED
           1 STATEMENTS WHERE OPTHINT NOT APPLIED
             OR PARTIALLY APPLIED
           0 STATEMENTS WHERE OPTHINT IS NOT FOUND
```

7.4 Authorization errors

As with other DB2 commands, various authorization errors are possible if the user lacks sufficient authority to complete the bind/rebind operation or to execute the plan or package. Chapter 9, “Security considerations” on page 173 discusses this aspect in detail. In this section, we discuss some of the common authorization errors and how to resolve them.

Important: The examples covered in this section deal with authorization IDs. When using roles in a trusted context, similar considerations apply. For an introduction to roles, see 9.6, “Packages using roles in a trusted context” on page 188. For details, see *Securing DB2 and Implementing MLS on z/OS,* SG24-6480-01.

7.4.1 BIND/REBIND PACKAGE failure: Access to underlying objects

The package owner must have authority to access the underlying objects. For example, if your program reads the EMP table, you must have SELECT authority for that table. Similarly, if your program updates the DEPT table, the package owner must have UPDATE authority for that table. Note that it is the authority of the package owner that is checked, not the primary or secondary authorization ID issuing the bind command.

An example of this situation is shown in Example 7-4.

Example 7-4 BIND/REBIND PACKAGE failure - access to underlying objects

```
DSNX200I  -DB9A REBIND SQL ERROR
           USING PAOLOR9 AUTHORITY
           PLAN=(NOT APPLICABLE)
           DBRM=PGM05
           STATEMENT=47
           SQLCODE=-551
           SQLSTATE=42501
           TOKENS=PAOLOR9.SELECT.PAOLOR3.MYTABLES
           CSECT NAME=DSNXOSC
           RDS CODE=-100

DSNT233I  -DB9A UNSUCCESSFUL REBIND FOR
           PACKAGE =
DB9A.DFINTEST2.PGM05.(2008-11-08-19.50.08.741500)
```

In this example, the owner (PAOLOR9) is not authorized to select from table PAOLOR3.MYTABLES.

7.4.2 BIND/REBIND PACKAGE failure: Access to collection

The package owner must also have the ability to create packages in the specified collection.

An example of this situation is shown in Example 7-5.

Example 7-5 BIND/REBIND PACKAGE failure - access to collection

```
DSNT235I  -DB9A DSNTBCM2 BIND AUTHORIZATION ERROR
           USING PAOLOR9 AUTHORITY
           PACKAGE =
DB9A.DFINTEST3.PGM05.(2008-11-08-19.50.08.741500)
```

```
           PRIVILEGE = CREATE IN
DSNT233I  -DB9A UNSUCCESSFUL BIND FOR
           PACKAGE =
DB9A.DFINTEST3.PGM05.(2008-11-08-19.50.08.741500)
```

In this example, the owner (PAOLOR9) does not have the authority to create a package in collection DFINTEST3.

7.4.3 Package or plan bind failure: Unable to add

We assume here the value of the DSNZPARM BINDNV is set to BINDADD. In this case, the owner must also have the ability to add a package via the BINDADD privilege. See 5.4.1, “BINDNV in DSNZPARM” on page 79 for details on this setting.

An example of this situation is shown in Example 7-6.

Example 7-6 Package or plan bind failure - unable to add

```
DSNT235I  -DB9A DSNTBCM2 BIND AUTHORIZATION ERROR
           USING PAOLOR9 AUTHORITY
           PACKAGE =
DB9A.DFINTEST3.PGM05.(2008-11-08-19.50.08.741500)
           PRIVILEGE = BINDADD
DSNT233I  -DB9A UNSUCCESSFUL BIND FOR
           PACKAGE =
DB9A.DFINTEST3.PGM05.(2008-11-08-19.50.08.741500)
```

In this example, the owner PAOLOR9 does not have the BINDADD privilege. Depending on other authorizations, the owner might be able to replace existing package but not create new ones.

7.4.4 BIND/REBIND PLAN failure: Access to specific package

In order to include a package in a plan, the plan owner must have execute authority on the package being included. The results depend on whether you specify VALIDATE(BIND) or VALIDATE(RUN).

An example of this situation when VALIDATE(BIND) is used is shown in Example 7-7. Here the bind plan is unsuccessful.

Example 7-7 BIND PLAN failure - access to specific package

```
DSNT241I  -DB9A BIND AUTHORIZATION ERROR ON PLAN PLAN9
           AUTHORIZATION-ID PAOLOR9 NOT AUTHORIZED TO EXECUTE
           PACKAGE DFINTEST2.PGM05
DSNT201I  -DB9A  BIND FOR PLAN PLAN9    NOT SUCCESSFUL
```

In this example, the owner (PAOLOR9) is not authorized to execute package DFINTEST2.PGM05 and the plan bind is unsuccessful.

An example of this situation when VALIDATE(RUN) is used is shown in Example 7-8. Here, the plan bind is successful even when the package has NOT been added. This plan then receives an error at run-time if the package authorization is still missing.

Example 7-8 Plan bind failure - access to specific package

```
DSNT241I  -DB9A BIND AUTHORIZATION WARNING ON PLAN PLAN9
           AUTHORIZATION-ID PAOLOR9 NOT AUTHORIZED TO EXECUTE
           PACKAGE DFINTEST2.PGM05
DSNT252I  -DB9A DSNTBCM1 BIND OPTIONS FOR PLAN PLAN9
           ACTION        ADD
           OWNER         PAOLOR9
           VALIDATE      RUN
           ISOLATION     CS
           ACQUIRE       USE
           RELEASE       COMMIT
           EXPLAIN       NO
           DYNAMICRULES  RUN
DSNT253I  -DB9A DSNTBCM1 BIND OPTIONS FOR PLAN PLAN9
           NODEFER       PREPARE
           CACHESIZE     0256
           QUALIFIER     PAOLOR9
           CURRENTSERVER
           CURRENTDATA   NO
           DEGREE        1
           SQLRULES      DB2
           DISCONNECT    EXPLICIT
           REOPT         NONE
           KEEPDYNAMIC   NO
           IMMEDWRITE    NO
           DBPROTOCOL    DRDA
           OPTHINT
           ENCODING      EBCDIC(00037)
           ROUNDING      HALFUP
           PATH
DSNT200I  -DB9A  BIND FOR PLAN PLAN9    SUCCESSFUL
```

In this example, the owner (PAOLOR9) is not authorized to execute package DFINTEST2.PGM05 but the plan bind is successful.

7.4.5 BIND/REBIND PLAN failure: Access to collid.* package

In order to include a package in a plan, the plan owner must have execute authority on all local packages in the collection which are being included via the *collid.** parameter and those explicitly specified.

An example of this situation is shown in Example 7-9.

Example 7-9 Plan bind failure - access to collid. package*

```
DSNT241I  -DB9A BIND AUTHORIZATION ERROR ON PLAN PLAN9
           AUTHORIZATION-ID PAOLOR9 NOT AUTHORIZED TO EXECUTE
           PACKAGE DFINTEST2.*
DSNT201I  -DB9A  BIND FOR PLAN PLAN9    NOT SUCCESSFUL
```

In this example, the owner (PAOLOR9) is not authorized to execute package DFINTEST2.* (i.e., all packages in collection DFINTEST2) and the bind plan is not successful.

7.4.6 Plan execution failure or unable to connect

This is a runtime error obtained when the user cannot connect to the DB2 subsystem using the specified plan or is not authorized for the plan.

An example of this situation is shown in Example 7-10.

Example 7-10 Plan execution failure - unable to connect

```
DSNT408I SQLCODE = -922, ERROR:  AUTHORIZATION FAILURE: PLAN ACCESS ERROR.
   REASON 00F30034
DSNT418I SQLSTATE   = 42505 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNAET03 SQL PROCEDURE DETECTING ERROR
```

In this example, the connection to DB2 using the plan is not successful. Check that you are attempting to connect to the proper DB2 subsystem and are authorized to use the plan. One common way in which such authorizations get lost is when you do not specify the RETAIN keyword of the BIND command.

A failure to connect results in other errors such as SQLCODES -923, -924 or -927.

7.4.7 Package execution failure: Unable to set CURRENT SQLID

In order to issue a SET CURRENT SQLID command within the package, the authorization ID executing the package must have the SQLID as one of the secondary authorization ID. Note that the authorization checked is that of the ID executing the SQL, not the owner of the package, regardless of what option (RUN versus BIND) is used for DYNAMICRULES.

An example of this situation is shown in Example 7-11.

Example 7-11 Package execution failure - unable to set CURRENT SQLID

```
NON ZERO RETURN CODE FROM SET CURRENT SQLID
SQLCODE = -0553
SQLTATE = 42503
SQLERRM =   XYZ
```

In this example, the package issued a SET CURRENT SQLID = ‘XYZ’ and XYZ is not one of the secondary authorization IDs for the user executing the package.

7.4.8 Result of a SET CURRENT PACKAGESET to a missing PKLIST entry

Note: When a package issues a SET CURRENT PACKAGESET statement, DB2 makes no attempt to verify that the collection ID referenced is part of the PKLIST for the current plan. It always returns SQLCODE 0.

As shown in Example 7-12, an attempt to set it to XYX (an invalid collection) is successful but the first SQL statement executed after it attempts to look for the package in the specified collection. When this is unsuccessful, an SQLCODE -805 with a reason code of 02 (package missing in the plan) is returned.

Example 7-12 Package execution failure - unable to use CURRENT PACKAGESET

```
NON ZERO RETURN CODE FROM OPEN
SQLCODE = -0805
SQLTATE = 51002
SQLERRM =   DB9A.XYZ.PGM08.186953A31C08EA53 PLAN1 02
```

In this example, PGM08 is present in the plan but not in the specific collection XYX, and DB2 returns the error, which can be somewhat misleading at first.

7.4.9 Package or plan execution failure: Disabled in this environment

This situation occurs when the plan exists but cannot be executed in the specific environment. For example, an attempt to execute plan in a batch environment that has been enabled only for CICS can result in this error.

An example of this situation is shown in Example 7-13.

Example 7-13 Plan execution failure - disabled in this environment

```
NON ZERO RETURN CODE FROM OPEN
SQLCODE = -0807
SQLTATE = 23509
SQLERRM =   TEMP.PGM08 BATCH
```

In this example, package PGM08 has been explicitly disabled for batch access as indicated by the error message.

7.5 Dynamic SQL authorization errors

Dynamic SQL presents a complete new set of challenges from an authorization perspective. The DYNAMICRULES option, along with the runtime environment of the package (whether the package is run stand-alone or under the control of a stored procedure or user-defined function) determines the authorization ID used to check the authorization.

See 9.5, “Dynamic SQL statements in stored procedures” on page 185 for details.

7.6 Inoperative packages

There are two flags in SYSPACKAGE that indicate the usability of a package.

- VALID:

  Indicates whether the package is valid.

- OPERATIVE:

  Indicates whether the package can be allocated.

Packages can become invalid (VALID = N) when any object that the package depends on is altered or dropped. For example, a package can become invalid when an index or table is dropped, or when a table column is altered. The SYSIBM.SYSPACKDEP catalog table records package dependencies. Depending on the change, different copies of the packages can be affected differently. In the case of a dropped table, all of the copies of the package would be invalidated, whereas, in the case of a dropped index, only certain copies that depend on the dropped index might be invalidated.

These considerations are discussed in detail in 8.5, “Schema changes and their effect on packages” on page 167.

In Example 7-14, we show an example of the error that you receive when you attempt to execute an inoperative package.

Example 7-14 Package execution failure - inoperative package

```
NON ZERO RETURN CODE FROM OPEN
SQLCODE = -0904
SQLCA   = SQLCA      h      00E30305 00000801 TEMP.PGM23.186B2D4F1756929C
                     57011
SQLERRM =   00E30305 00000801 TEMP.PGM23.186B2D4F1756929C
```

The example shows SQLCODE -904 for the package TEMP.PGM23.

A query to list all invalid or inoperative packages is provided in A.1, “Identifying invalid or inoperative packages” on page 261.

Part 3

Managing packages

In this part of the book, we provide considerations for managing an environment with packages, focusing on the DBAs, system, and application. We begin with various steps involved in administering the package, such as managing your collections, plan_tables, and so on. We then cover the security aspects that must be in place.

One other important area is the management of the access paths, because it is possible for access paths to change (and cause severe performance degradation in rare cases). We discuss this issue and provide recommendations on how you can control these changes. In particular, we discuss the newly introduced *package stability* feature.

We then look at the major DSNZPARMs influencing package performance and describe the tracing functions available with DB2 Instrumentation Facility Component (IFC).

This part includes the following chapters:

- Chapter 8, “Administration of packages” on page 161.
- Chapter 9, “Security considerations” on page 173.
- Chapter 10, “Access path management” on page 199.
- Chapter 11, “Performance considerations” on page 223

8

Administration of packages

In this chapter we discuss how to administer packages, how to manage the impact of schema and environment changes, and how to change and distribute packages between test and production environments.

We cover the following topics:

- Package administration
- Package proliferation
- Deprecating packages
- Package and PLAN_TABLE maintenance
- Schema changes and their effect on packages

8.1 Package administration

In this section, we discuss the details of package administration. This includes strategies to reduce the maintenance, as well as how to keep your catalog and EXPLAIN tables well maintained. We also talk about the impact of schema changes on your packages and how to keep your system available with minimum downtime.

8.2 Package proliferation

One of the first decisions you probably made when converting to packages was whether you would use multiple versions of a package in the same collection. If you have chosen to implement versioning of your packages, one issue you might face is a proliferation of many packages, both in your test and production systems.

8.2.1 Development environment

Package proliferation can occur using VERSION(AUTO) in the development environment, which would be the case where each time a program is recompiled, it is bound into a new package. During development, this can create many packages that are rarely if ever used. In the environment where your developers are initially testing their code, it is a good idea to avoid specifying VERSION(AUTO) in the precompile step. When you begin to migrate the code, it is a better time to recompile and use VERSION(AUTO) to migrate the code to the rest of the testing regions. In this environment, it is still possible to generate many packages that will soon be obsolete.

Tip: In your initial development region, do not specify VERSION(AUTO), even if you plan to use it in your shop. After the initial code has been migrated and it is not changing many times a day, you can begin to use VERSION(AUTO).

8.2.2 Production environment

One method of moving programs into production is to use VERSION(AUTO). Refer to 5.3, “Contrasting collection, version, and consistency CONTOKEN” on page 76 for more information. By using VERSION(AUTO), you can BIND your package into the production collection before moving the load module into the production load library. When you have moved the new load module into production, DB2 determines the CONTOKEN from the new load module and selects the correct package to execute. If you need to back out the program, you simply restore the old load module and the next time DB2 executes the package, DB2 selects the old package to execute. This process can cause a build-up of old packages in the DB2 catalog, and also in the extended EXPLAIN tables that DB2 uses to document access paths.

You need to periodically clean up the packages that are no longer in use from both the catalog and the EXPLAIN tables. One of the drawbacks to maintaining multiple versions of a package in the same collection is that there is not an easy method to determine which of your old packages are obsolete and can be freed.

8.2.3 Benefits of cleaning up your packages and PLAN_TABLE

The benefits of cleaning up your packages and EXPLAIN tables are as follows:

- Reduce the size of the DB2 directory
- Reduce the size of the DB2 catalog
- Reduce the size of the EXPLAIN tables listed
- Faster problem determination

We discuss each of these benefits here.

Reducing the size of the DB2 directory

By removing old packages, you reduce the size of your SPT01 and SCT02 table spaces in the DB2 directory. This improves the speed of your utilities against the DB2 directory. DB2 performs faster because it has fewer entries to look through when attempting to allocate the package.

Reducing the size of the DB2 catalog

By removing old packages, you reduce the size of the DSNDB06.SYSPKAGE table space, which includes tables: SYSPACKAGE, SYSPACKDEP, SYSPACKSTMT, SYSPACKLIST, SYSPACKAUTH and SYSPKSYSTEM. This helps your queries against the catalog run faster as well as the improving the elapsed time of your utilities against the DB2 catalog.

Reducing the size of your EXPLAIN tables

Removing rows in your EXPLAIN tables reduces the size of the tables. The performance of your queries against the EXPLAIN tables might be better, depending on the access path. It improves the speed of your utilities against these tables.

Faster problem determination

By reducing the number of entries in SYSPACKAGE and related catalog tables, as well as the PLAN_TABLE and extended EXPLAIN tables, you have less data to look at while performing problem determination, making your job easier and allowing you to perform it faster.

8.2.4 VERSION parameter

There are three ways to use the VERSION parameter in the DB2 precompiler or coprocessor:

- Do not code the VERSION parameter.
- Code a specific value, such as release number - VERSION(V1R1).
- Code VERSION(AUTO).

Do not code the VERSION parameter

If you do not use the VERSION parameter, each time you bind a package, the new package replaces the old package. In this case, you only wind up with at most one package in a collection for a program. In terms of maintaining packages, this is the simplest option. The unique index on SYSPACKAGE(LOCATION,COLLID,NAME,VERSION) avoids duplicates.

Code a specific version

If you use the VERSION(UserValue) parameter of the precompile, each time you bind a package, you can create a new VERSION. Inside a given collection, you might have multiple versions of a single package. The most common use of this flavor of the VERSION parameter is to use it for release levels. In this case, you would only have one package per release, but you could still have the same package in your collection for different releases. To make

package maintenance easier, you could have all modules in your application recompiled for each new release, but this might be impractical in many instances.

Code VERSION(AUTO)

If you use VERSION(AUTO), each time you bind a new copy of your program, you create a new version of the package in the collection. This might lead to many copies of a single package in one collection. If you are using VERSION(AUTO) you need to have a maintenance strategy that periodically cleans up the obsolete packages. In addition, where ever you use EXPLAIN(YES), you also generate rows in your EXPLAIN tables, which also need to be maintained. VERSION(AUTO) is the recommended setting for this parameter. This builds a new version of the package each time you precompile your code. The version equals a store clock representation of a timestamp value assigned at precompile time.

8.2.5 How to decide what packages to keep

There are several criteria that you must consider to decide which packages need to be freed and which rows in your EXPLAIN tables need to be deleted:

- Packages that have no matching load module can be freed. If the load module does not exist, then the package cannot be executed.
- When determining rows to be deleted, consider these criteria:
  - No matching load module
  - Keeping x number of versions
  - Obsolete releases of software

Set limits on the number of package versions maintained in a given collection. It is likely that users find DB2 has more package versions than the source code repository does of the LOAD, DBRM, and SOURCE. Do not allow DB2 to get slowed down with obsolete data.

A query to list multiple packages in a collection is shown in Example 8-1.

Example 8-1 Multiple versions for a package in a collection

```
SELECT
 SUBSTR(SYSPACK.COLLID,1,18)  AS COLLECTION
,SUBSTR(SYSPACK.NAME,1,8)     AS PACKAGE
,SUBSTR(SYSPACK.VERSION,1,26) AS VERSION
,BINDTIME                     AS BINDTIME
FROM SYSIBM.SYSPACKAGE SYSPACK
WHERE
  EXISTS
    (SELECT 1
       FROM SYSIBM.SYSPACKAGE SYSPACK1
       WHERE SYSPACK.COLLID     = SYSPACK1.COLLID
         AND SYSPACK.NAME       = SYSPACK1.NAME
         AND SYSPACK1.BINDTIME < CURRENT TIMESTAMP - 3 MONTHS
     )
  AND EXISTS
    (SELECT SYSPACK2.COLLID, SYSPACK2.NAME, COUNT(*)
     FROM SYSIBM.SYSPACKAGE SYSPACK2
     WHERE SYSPACK.COLLID = SYSPACK2.COLLID
       AND SYSPACK.NAME   = SYSPACK2.NAME
     GROUP BY SYSPACK2.COLLID, SYSPACK2.NAME
     HAVING COUNT(*) > 3
    )
```

```
   ORDER BY COLLECTION, PACKAGE,BINDTIME
WITH UR

COLLECTION       PACKAGE  VERSION                    BINDTIME
---------------+--------+--------------------------+--------------------------
DSNTWR91         DSNTWR   2008-05-12-14.51.41.727543 2008-05-12-09.51.46.178025
DSNTWR91         DSNTWR   2008-02-26-17.56.37.779396 2008-02-26-11.56.42.534767
DSNTWR91         DSNTWR   2007-04-23-18.08.51.467741 2007-04-23-13.08.58.034436
```

- If you install software that uses DB2, the vendors (or you) can add versions to the packages for a given release. That way, when you upgrade the release, you should be able to free all of the old release's packages (or deprecate them as shown in 8.3, "Deprecating packages" on page 165).

8.3 Deprecating packages

After you have identified the packages that you would like to remove, here is a safe procedure for removing the packages while providing for rapid fallback in case the package should be needed later. To use this procedure, you must *disable ALL* environments that are available in DB2. The next example shows the eight environments for DB2 9 for z/OS. If a future release adds more environments, you have to revisit this procedure to make it applicable to the current release. The steps you need to take to deprecate packages are:

- Verify the list of obsolete packages against the accounting data to check that the package was not active during the previous quarter.
- Disable the package for *all* environments. The command for DB2 9 for z/OS is:

  ```
  REBIND PACKAGE(LOCATION.COLLID.NAME)
  DISABLE(BATCH,CICS,DB2CALL,DLIBATCH,IMSBMP,IMSMPP,REMOTE,RRSAF)
  ```

  If an application receives an SQLCODE -807, simply rebind the package enabling the environment it runs in. REBIND ENABLE(BATCH) for example. The first part of the error message shows you the package that has been disabled. The second part shows you the associated plan. The error message you receive looks like:

  ```
  DSNT408I SQLCODE = -807, ERROR: ACCESS DENIED: PACKAGE PGM01 IS NOT ENABLED
  FOR ACCESS FROM BATCH
  DSNT418I SQLSTATE = 23509 SQLSTATE RETURN CODE
  DSNT408I SQLCODE = -923, ERROR: CONNECTION NOT ESTABLISHED: DB2 ACCESS,
  REASON 00E3001B, TYPE 00000800, NAME FINTEST1
  DSNT418I SQLSTATE = 57015 SQLSTATE RETURN CODE
  ```

- Update any queries you run to generate REBIND commands to exclude packages not enabled for any environment by adding a subselect not exists to SYSPKSYSTEM. This prevents packages that have been deprecated from being rebound by standard maintenance queries. See Example 8-2 for a query to list all packages not disabled for all environments.

Example 8-2 Sample query to list all packages that have not been completely disabled

```
SELECT DISTINCT
       SUBSTR(A.COLLID,1,8)   AS COLLID,
       SUBSTR(A.NAME,1,8)     AS PACKAGE,
       SUBSTR(A.VERSION,1,44) AS VERSION
FROM SYSIBM.SYSPACKAGE A
WHERE NOT EXISTS
 (SELECT COUNT(*) FROM SYSIBM.SYSPKSYSTEM B
```

```
WHERE A.LOCATION = B.LOCATION
  AND A.COLLID   = B.COLLID
  AND A.NAME     = B.NAME
  AND A.CONTOKEN = B.CONTOKEN
  AND B.ENABLE = 'N'
  HAVING COUNT(*) = 8);
```

- Free obsolete packages after they have been disabled for a specified number of days.

8.4 Package and PLAN_TABLE maintenance

You also want to periodically clean up rows in your PLAN_TABLE and extended EXPLAIN tables. You should consider the data backup and recovery requirements for the application.

Retention criteria

The retention criteria for PLAN and EXPLAIN tables are as follows:

- Keep a specified number of binds for a given package in the PLAN_TABLE.
- Rows in PLAN_TABLE that are older than a specified number of days old can be candidates for deletion, if they are not restricted by another of your criteria.
- Rows in PLAN_TABLE that have OPTHINT populated should not be deleted because they might be used for access path selection in the future.
- Tie package clean-up to application releases. Remember to not remove active OPTHINT rows or rows that might be needed for previous versions when using package stability.
- Using naming conventions for plans and collections tied to vendor software can help in freeing outdated packages and assist in quickly reverting to a previous release through the use of version names in the collections and plans.
- Consider using aliases for the vendor products DBRM libraries, LOAD libraries, and so on. If a version scheme is used in naming patterns, it can quickly identify all packages related to a release and use this output to generate FREE PACKAGE statements for those old packages. See Example 8-3 for finding packages bound to '%P115%' collections associated to P%1151% plans. Where 1151 represents release 11.5 of a given vendor product.

Example 8-3 Sample query to list all packages related to a release

```
SELECT
  PACK.LOCATION
, PACK.COLLID
, PACK.NAME
, PACK.VERSION
FROM SYSIBM.SYSPACKAGE  PACK,
     SYSIBM.SYSPACKLIST PACKLIST
WHERE PACKLIST.PLANNAME LIKE 'P%1151%'
  AND PACK.COLLID = PACKLIST.COLLID
AND PACK.COLLID LIKE '%1151%';
```

- Be aware of scripts and commands that FREE packages with no WHERE clauses or use wildcards.

- Free any obsolete packages monthly, weekly or tied to each maintenance change. Keep only the package versions needed to run your business. Follow the 8.3, “Deprecating packages” on page 165 to safely manage the freeing of package versions.
- Do not forget trigger packages and stored procedure packages.

Keep only the necessary rows in your PLAN_TABLE to determine performance changes from various code releases. Clean up old versions of EXPLAIN output. Consider updating the REMARKS column of PLAN_TABLE to include any notes about the performance or version to assist in your performance tuning and historical analysis.See Appendix A.16, “Finding obsolete PLAN_TABLE entries” on page 270.

8.5 Schema changes and their effect on packages

There are two flags in SYSPACKAGE which indicate the usability of a package.

- VALID

  Whether the package is valid. If the package is marked as VALID, the package is eligible for execution. If the package is marked NOT VALID, the package must be either explicitly or implicitly be rebound. before it can be executed.

- OPERATIVE

  Whether the package can be allocated. If the package is marked as NOT OPERATIVE, the package must be explicitly rebound before it can be executed.

Packages can become invalid when any object that the package depends on is altered or dropped. For example, a package can become invalid when an index or table is dropped, or when a table column is altered. The SYSIBM.SYSPACKDEP catalog table records package dependencies. Depending on the change, different copies of the packages can be affected differently. In the case of a dropped table, all of the copies of the package would be invalidated, whereas in the case of a dropped index, only certain copies that depend on the dropped index might be invalidated.

When an object is changed that causes a package to be invalidated (see “Schema invalidation events” on page 168 for events that cause package invalidation), the following events occur in SYSPACKAGE:

- Prior to the change, the package is both valid and operative (VALID=Y, OPERATIVE=Y).
- After the change, the package is invalid and operative (VALID=N, OPERATIVE=Y).
- If using AUTOBIND (set via a DSNZPARM) or when an explicit bind is attempted and the bind fails:
  - The package is marked invalid and inoperative (VALID=N, OPERATIVE=N).
  - The object has to be created or the source code must be changed to correct the problem. BINDs (not REBINDs) pick up the source code changes.
  - The package is marked valid and operative if bind is successful (VALID=Y,OPERATIVE=Y).

It is important to note that an invalid (but still operative) package can be corrected by autobind, while a package that is inoperative requires an explicit BIND or REBIND. This is depicted in Figure 8-1.

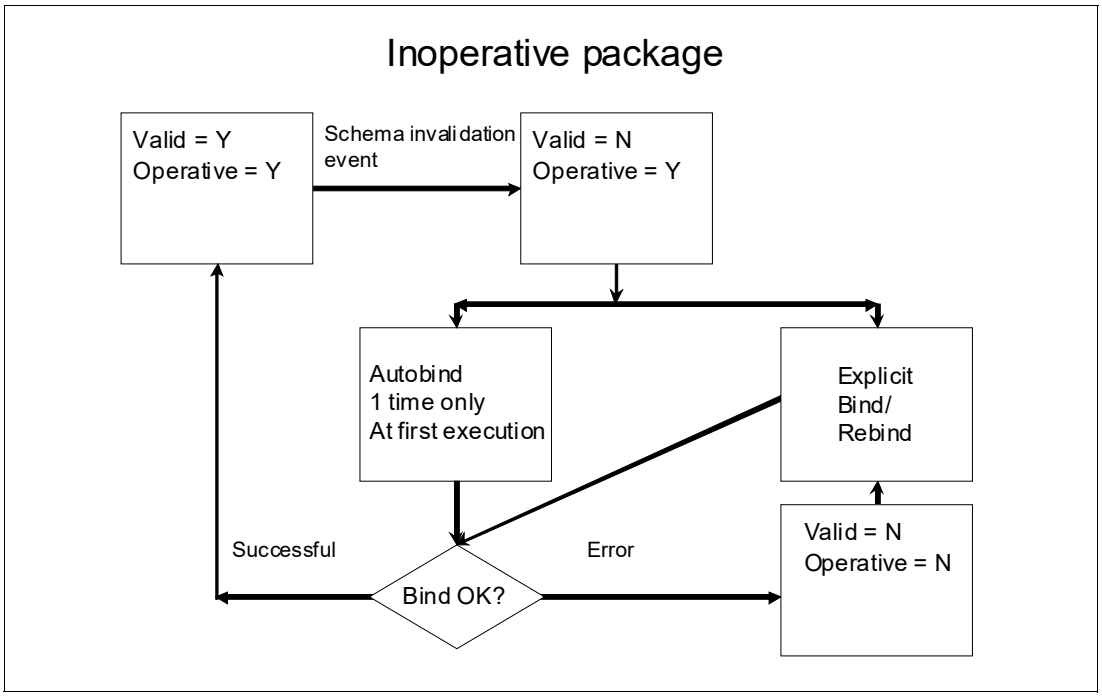

Figure 8-1 VALID and OPERATIVE packages

8.5.1 Schema invalidation events

During the course of schema changes to the database, packages might be invalidated. Each release of DB2 continues to make improvements to online schema evolution, which might reduce the number of changes that invalidate packages. Online schema changes are evolutionary. What causes a package to be invalidated changes from release to release. Here, we list the online schema changes in DB2 9 for z/OS that can invalidate a package:

- Dropping a table, index, or other object, even if you later recreate the object
- Dropping an object that a package depends on
- Revoking an authorization to use an object
- Changing the SBCS CCSID attribute of a table space
- Altering a table, if any one of the following conditions is true:
  - The table is a created temporary table or a materialized query table.
  - The table is changed to add or drop a materialized query definition.
  - The AUDIT attribute of the table is changed.
  - A DATE, TIME, or TIMESTAMP column is added and its default value for added rows is CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, respectively.
  - A security label is added.
  - The length attribute of a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY column has changed.
  - The column data type, precision, scale, or subtype is changed.
  - The column is renamed.
  - The table is partitioned and a partition is added or one of the existing partitions is changed or rotated
  - A table other than an auxiliary table is renamed.

- Altering an index, if one of the following conditions is true:
  - A column is added to the index.
  - The index is altered to be **PADDED**.
  - The index is a partitioning index on a table that uses index-controlled partitioning, and one or more of the limit key values are altered.
  - The index is altered to **REGENERATE**.
- If an external procedure is altered, all packages that refer to the procedure.
- When a version of an SQL procedure is altered to change any option that is specified for the active version, see *DB2 Version 9.1 for z/OS SQL Reference* SC18-9854, Chapter 5 for a full description of when procedures are rebound.
- The creation of an INSTEAD OF trigger causes dependent packages, plans, and statements in the dynamic statement cache to be marked invalid if the view definition is not read-only.
- A revoke or cascaded revoke of any privilege or role that was exercised to create a plan or package makes the plan or package invalid when the revokee no longer holds the privilege from any other source.

Our discussion here deals with what can be done to increase availability when packages are marked invalid.

8.5.2 Invalid and inoperative packages

When a schema change occurs that affects a package to the point that it is no longer executable, the package is marked with an “N” in the VALID column of SYSPACKAGE. The next time that the package is called to be executed, DB2 determines that it is invalid and issues an automatic REBIND and if successful, it loads it into the EDM pool.

If there are issues where DB2 cannot complete the REBIND successfully, DB2 marks the package with an “N” in the OPERATIVE column of SYSPACKAGE. After this has happened, DB2 no longer attempts to do an automatic rebind of the package. The only way to get the package to be executable again is to issue either an explicit BIND or REBIND statement.

Limiting the impact of invalid packages

You might want to consider if you want automatic rebinds happening at certain points of the day. You cannot prevent them from occurring, but you can reduce the number of automatic binds by checking for invalid or inoperative packages (and plans) at a time when the impact is minimal. You can scan your SYSPACKAGE table to determine if you have invalid (VALID =”N”) or inoperative (OPERATIVE = “N”) packages in your system. Example 8-4 shows a query to retrieve all invalid or inoperative packages in your subsystem.

Example 8-4 Query to determine invalid or inoperative package

```
SELECT
 SUBSTR(COLLID,1,18) AS COLLECTION
,SUBSTR(NAME,1,8)    AS PACKAGE
,HEX(CONTOKEN)       AS CONTOKEN
,SUBSTR(OWNER,1,8)   AS OWNER
,SUBSTR(CREATOR,1,8) AS CREATOR
,VALID               AS VALID
,OPERATIVE           AS OPERATIVE
FROM SYSIBM.SYSPACKAGE
WHERE VALID = 'N' OR OPERATIVE = 'N'
```

```
ORDER BY COLLECTION, NAME, VERSION
WITH UR;
```

Note: The queries demonstrated in the section use the SUBSTR function to reduce the size of the output. The columns on which we apply the SUBSTR function are all larger than the lengths we show here.

You can use the query in Example 8-5 to format a REBIND statement for each of the invalid or inoperative packages and run this REBIND command as part of a maintenance job at a slow time in your shop, if you have one. One example of this is to run the query and the associated REBINDs before your online system comes up for the day, or at another slow time in your processing. The idea is to get the explicit REBINDs done at a time when it has the least impact to your system. If your online system executes a package marked invalid and for some reason, the BIND does not complete successfully the first time, the package will not successfully execute until the explicit REBIND has been successfully run.

Important: Limit the impact of automatic rebinds by explicitly rebinding invalid or inoperative packages at a time when it has the least impact.

Example 8-5 SQL to generate REBIND commands for invalid and inoperative packages

```
SELECT
  'REBIND PACKAGE(' CONCAT
   STRIP(COLLID)    CONCAT
  '.'               CONCAT
   STRIP(NAME)      CONCAT
  '.('              CONCAT
   STRIP(VERSION)   CONCAT
  '))'
FROM SYSIBM.SYSPACKAGE
WHERE (VALID = 'N' OR OPERATIVE = 'N')
  AND VERSION <> ''
UNION ALL

SELECT
  'REBIND PACKAGE(' CONCAT
   STRIP(COLLID)    CONCAT
  '.'               CONCAT
   STRIP(NAME)      CONCAT
  ')'
FROM SYSIBM.SYSPACKAGE
WHERE (VALID = 'N' OR OPERATIVE = 'N')
AND VERSION = ''
ORDER BY 1
```

Output from this SQL is listed in Example 8-6.

Example 8-6 Sample output from SQL to generate REBIND command

```
REBIND PACKAGE(SYSIBMTS.DSN50019)
REBIND PACKAGE(SYSPROC.NODIFF.(V1))
```

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks publications

For information about ordering these publications, see “How to get IBM Redbooks publications” on page 172. Note that some of the documents referenced here might be available in softcopy only:

- *DB2 Packages: Implementation and Use,* GG24-4001-00
- *DB2 9 for z/OS Performance Topics*, SG24-7473
- *DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7604
- *DB2 9 for z/OS Performance Topics*, SG24-7473
- *Squeezing the Most Out of Dynamic SQL with DB2 for z/OS and OS/390*, SG24-6418
- *DB2 for z/OS: Considerations on Small and Large Packages,* REDP-4424
- *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability,* SG24-7134
- *A Deep Blue View of DB2 Performance: IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS*, SG24-7224
- *Securing DB2 and Implementing MLS on z/OS,* SG24-6480-01

Other publications

These publications are also relevant as further information sources:

- *DB2 Version 9.1 for z/OS Reference for Remote DRDA Requesters and Servers,* SC18-9853-02
- *DB2 Version 9.1 for z/OS Installation Guide*, GC18-9846-05
- *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844-03
- *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841-03
- *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851-04
- *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854-05
- *DB2 Version 9.1 for z/OS Internationalization Guide*, SC19-1161-01
- *CICS Transaction Server for z/OS V3.1 CICS DB2 Guide,* SC34-6457-03
- *DB2 Version 9.5 for Linux, UNIX, and Windows Call Level Interface Guide and Reference, Volume 2,* SC23-5845-01
- *DB2 for Linux, Unix and Windows: Developing Java Applications*, SC23-5853
- *Considering SQLJ for Your DB2 V8 Java Applications*, by Connie Tsui, available from: http://www.ibm.com/developerworks/db2/library/techarticle/0302tsui/0302tsui.html
- *Enterprise COBOL for z/OS Compiler and Runtime Migration Guide Version 4 Release 1,* GC23-8527-00

Online resources

These Web sites are also relevant as further information sources:

- DB2 for z/OS home page:

  http://www.ibm.com/software/data/db2/zos/index.html
- The Open Group:

  http://www.opengroup.org/bookstore/catalog/c066.htm
- developerWorks, Technical resources for IBM Information Management software:

  http://www.ibm.com/developerworks/db2/
- Information Center for IBM Data Studio:

  http://publib.boulder.ibm.com/infocenter/dstudio/

How to get IBM Redbooks publications

You can search for, view, or download IBM Redbooks publications, IBM Redpaper publications, Technotes, draft publications, and Additional materials, as well as ordering hardcopy IBM Redbooks publications, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads:

ibm.com/support

IBM Global Services:

ibm.com/services

9

Security considerations

In this chapter, we discuss the authorization needed to create, modify, and execute packages. We begin with an overview of the authorization scheme for packages that includes explicit and implicit privileges.

We then cover the ability to limit the execution of packages in specific environments only. Applications dealing with dynamic SQL present a new set of challenges. We cover these next. Finally, we discuss the newly introduced concept of roles in a trusted context to overcome some of the limitations of existing schemes dealing with authorization IDs.

We cover the following topics:

- Package authorization overview
- ENABLE/DISABLE
- Controlling security for dynamic SQL applications
- Privileges required to execute stored procedures
- Dynamic SQL statements in stored procedures
- Packages using roles in a trusted context

9.1 Package authorization overview

In the following sections we discuss data access control, authorities, and privileges.

9.1.1 DB2 data access control

You can enable or disable data access control within DB2. Access to data can originate from a user through an interactive terminal session, an application program that is running in batch mode, or an IMS or CICS transaction. With the variety of access originators, the term process is used to represent all access to data. For example, within a DB2 subsystem, a process can be a primary authorization ID, one or more secondary IDs, a role, or an SQL ID. A process can gain access to DB2 data through several routines. As shown in Figure 9-1, DB2 provides different ways for you to control access from all but the data set protection route.

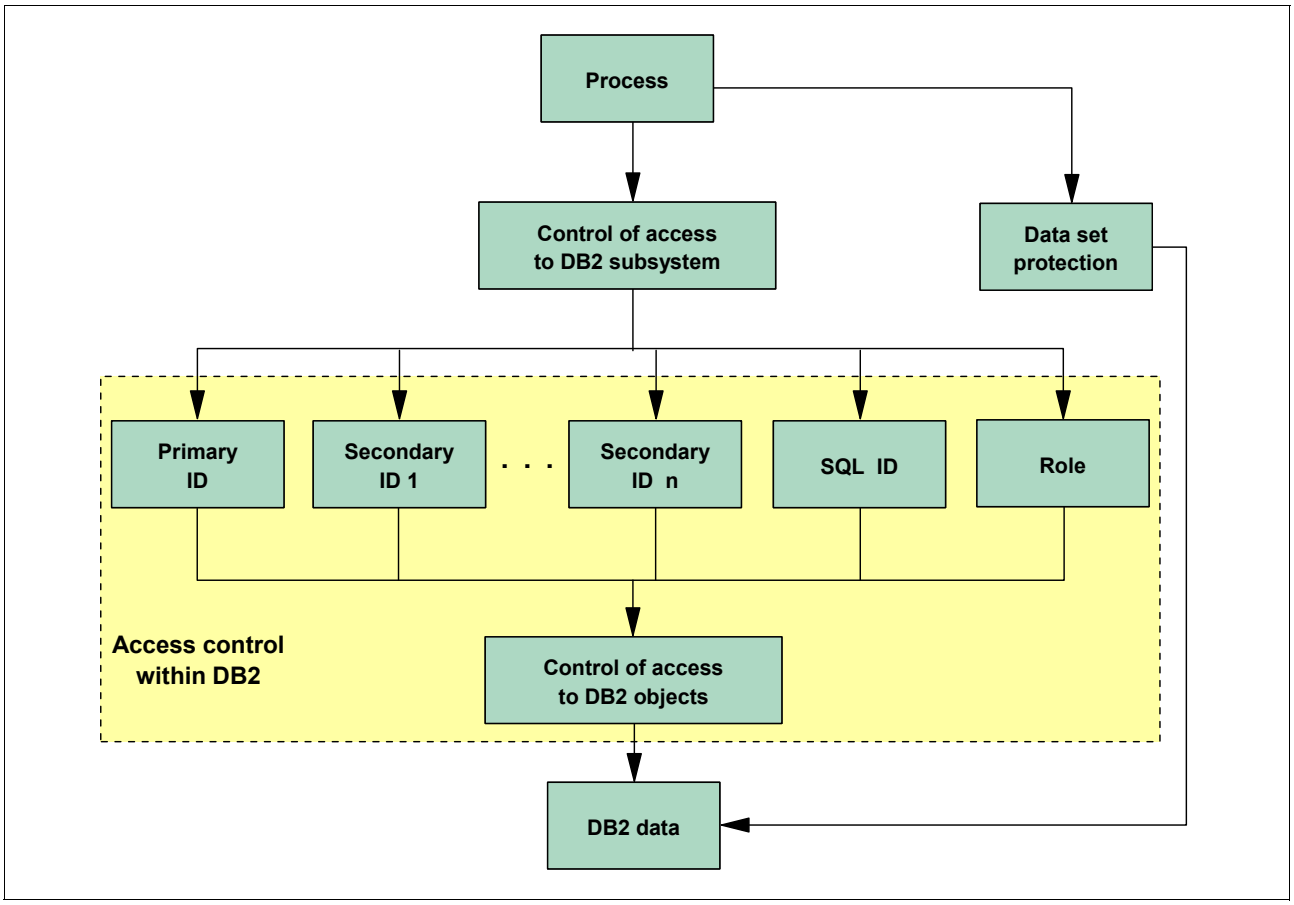

Figure 9-1 DB2 data access control

One of the ways that DB2 controls access to data is by using authorization IDs or roles. DB2 relies on IDs or roles to determine whether to allow or prohibit certain processes. DB2 assigns privileges and authorities to IDs or roles so that the owning users can take actions on objects. In this sense, it is an ID or a role, not a user, that owns an object. In other words, DB2 does not base access control on a specific user or person who need access. For example, if you allow other users to use your IDs, DB2 recognizes only the IDs, not the people or programs that use them.

This section summarizes the authorizations required to administer and use packages. We begin by discussing the explicit privileges which can exist at the package level. We then discuss the administrative authorities that bundle such privileges. Finally, we discuss various operations related to packages and the minimum authority needed to perform them.

Note: If a BIND or REBIND was done by a role in a trusted context, all subsequent BINDs or REBINDs must also be done by a role in a trusted context. For discussion of roles, see 9.6, “Packages using roles in a trusted context” on page 188.

While the distinction between explicit privileges (see 9.1.2, “Explicit privileges ”) and administrative authorities (see 9.1.3, “Administrative authorities ”) might seem arbitrary, the first set is more closely related to packages, while the latter is broader in nature.

Important: The DSNZPARM BINDNV controls what authority is needed to add a new package. If you specify a value of BIND, a user with BIND authority on a package can create a new VERSION. If you specify BINDADD, only users with BINDADD authority can add new VERSIONs of a package.

In this chapter, we assume that an explicit BINDADD privilege is necessary and the BINDNV value is set to BINDADD.

9.1.2 Explicit privileges

Next, we briefly describe the explicit privileges related to packages. We discuss package privileges, collection privileges, and system privileges.

Package privileges

The following three privileges are recorded in the SYSIBM.SYSPACKAUTH table:

- BIND:

  To bind and rebind a package

- COPY:

  To copy a package

- EXECUTE:

  To execute a package specified in the PKLIST of a plan

These privileges apply to every version of the package. The package owner, or whoever holds the bind authorizations with the WITH GRANT OPTION clause, can grant them on a specific package, for example:

```
GRANT BIND ON PACKAGE COL1.PROGA....
GRANT COPY ON PACKAGE COL1.PROGA.....
GRANT EXECUTE ON PACKAGE COL1.PROGA..
```

Users with SYSADM, SYSCTRL, and PACKADM WITH GRANT OPTION authority can grant access to all packages in a collection by specifying an asterisk (*) for the package_id. This eliminates the process of having to grant again when a new package is added to the collection:

```
GRANT BIND ON PACKAGE COL1.* ....
GRANT COPY ON PACKAGE COL1.* ....
GRANT EXECUTE ON PACKAGE COL1.* ....
```

SYSCTRL cannot specify the EXECUTE privilege in this case because it does not have the EXECUTE privilege over a package.

Collection privileges

Because a collection is considered a DB2 resource, you must have the following privilege in order to bind a new package into the collection:

- CREATE IN:

  To create packages in the designated collections.

CREATE IN privileges are recorded in SYSIBM.SYSRESAUTH.

System privileges

BINDAGENT is designed only for operations related to binding, so it does not have EXECUTE privilege. BINDADD includes packages as well as plans. These privileges are recorded in SYSIBM.SYSUSERAUTH:

- BINDAGENT:

  To bind, rebind, copy, or free a package or plan *on behalf of the owner who granted the BINDAGENT privilege*.

The BINDAGENT privilege is only valid for the relationship between the GRANTOR and the GRANTEE authorization IDs. The GRANTEE authorization ID (which can be any user ID, or anyone connected to this user ID as a secondary authorization ID) can specify the GRANTOR authorization ID as an OWNER in a BIND PACKAGE command.

9.1.3 Administrative authorities

SYSCTRL and PACKADM are administrative authorities that include package privileges. We briefly describe each authority:

- SYSCTRL:

  SYSCTRL has almost as much control of DB2 as SYSADM, except for access to user data. It is designed for the user who wants administrative but not data access privileges. (However, SYSCTRL can access the DB2 catalog.) SYSCTRL also has other authorities not related to packages.

  SYSCTRL does *not* have the authority to:

  - Execute data manipulation language (DML) statements on user tables or views
  - Execute packages
  - Execute plans
  - Set the current SQLID to a value that is not one of its primary or secondary authorization IDs
  - Grant DBADM, SYSADM, and PACKADM authorizations.

- PACKADM:

  PACKADM privileges are recorded in the SYSIBM.SYSRESAUTH table. It is designed to be a package administrator in a given collection. It has all explicit package privileges (BIND, COPY, EXECUTE) and the CREATE IN privilege, for a specific collection, or in all collections if an asterisk is specified for the collection when granted:

  ```
  GRANT PACKADM ON COLLECTION FIN_PROD_BACTH TO user1
  ```

Now that we have discussed the package privileges that are included in the administrative authorities, let us look at the same issue from a different perspective, to see which authority is sufficient for a specific privilege (such as BINDADD, CREATE IN, COPY, and so on).

We consider the authority for three separate groups of privileges:

- Who has BINDADD?
- Who has BIND, COPY, EXECUTE?
- Who has CREATE IN?

Table 9-1 summarizes the BINDADD privileges held by the administrative authorities. While the ID which is assigned the PACKADM authority does not automatically get the BINDADD privilege, it is advisable in most cases for the same ID to be granted BINDADD privilege.

Table 9-1 Who has BINDADD privilege?

	BINDADD
SYSADM	Y
	All packages in all collections
SYSCTRL	Y
	All packages in all collections
PACKADM	N
	N/A

Table 9-2 summarizes the BIND, COPY, and EXECUTE privileges held by the administrative authorities.

Table 9-2 Who has BIND, COPY, and EXECUTE privileges?

	BIND	**COPY**	**EXECUTE**
SYSADM	Y	Y	Y
	All packages in **all** collections	All packages in **all** collections	All packages in **all** collections
SYSCTRL	Y	Y	N
	All packages in **all** collections	All packages in **all** collections	N/A
PACKADM	Y	Y	Y
	All packages in **specified** collection	All packages in **specified** collection	All packages in **specified** collection

Table 9-3 summarizes the CREATE IN privileges held by the administrative authorities.

Table 9-3 Who has CREATE IN privileges?

	CREATE IN
SYSADM	Y
	All collections
SYSCTRL	Y
	All collections

	CREATE IN
PACKADM	Y
	Specified collection

9.1.4 Privileges required for package-related operations

Table 9-4 summarizes the privileges—explicit, administrative authority, implicit— required for package-related operations.

Notes:

- Users (authorization IDs or roles executing the process) must have at least *one* of the three privileges.
- If a BIND or REBIND was done by a role in a trusted context, all subsequent BINDs or REBINDs must also be done by a role in a trusted context. For a discussion of roles, see 9.6, “Packages using roles in a trusted context” on page 188.
- Implicit privileges are those acquired without an explicit grant being issued, for example, by virtue of package ownership.

Table 9-4 Privileges required for package-related operations

		You need at least one of these three possibilities:		
Operation that you are trying to perform	**Which authority is checked**	**Explicit privilege**	**Administrative authority**	**Implicit privilege**
Binding a new package	Owner of the package	Privileges to execute SQL statements in the package	SYSADM	N/A
		BINDADD	SYSADM SYSCTRL	
		CREATE IN for the specified collection	SYSADM SYSCTRL PACKADM	
	Binder if different from owner	BINDAGENT	SYSADM SYSCTRL	
	Note: If the binder has SYSADM or SYSCTRL authorities, the owner does not require BINDADD and CREATE IN authorities. For VALIDATE(RUN), the owner does not need privileges to execute SQL statements in the package. When the binder is different from the owner, the possible authorization ID values for the owner depend on the authorizations of the binder: ► If the binder has BINDAGENT authority, the owner must be the authorization ID that granted BINDAGENT authority to the binder. ► If the binder has SYSADM or SYSCTRL authorities, the owner can be any authorization ID. ► For all other cases, the owner must be one of the secondary authorization ID of the binder.			
Rebinding or bind replacing a package	Primary or any secondary authorization ID of the binder	BIND or BINDAGENT	SYSADM SYSCTRL PACKADM	Package owner
	Note: When the owner changes, the new owner needs BIND authority and privileges to execute SQL statements in the package. If the binder has SYSADM or SYSCTRL, BIND is not required for the new owner.			

		You need at least one of these three possibilities:		
Operation that you are trying to perform	**Which authority is checked**	**Explicit privilege**	**Administrative authority**	**Implicit privilege**
Freeing a package	Primary or any secondary authorization ID of the user	BINDAGENT	SYSADM SYSCTRL PACKADM	Package owner
Copying a package	Primary or any secondary authorization ID of the binder or target package owner	COPY on the source package, or BINDAGENT from the owner of the source package. Same privileges required as for binding a new package	SYSADM SYSCTRL PACKADM	Source package owner
Executing a plan	Primary or any secondary authorization ID of the user	EXECUTE on plan	SYSADM	Plan owner
	Note: Plan owner must have EXECUTE on packages listed in PKLIST.			
Granting EXECUTE privilege on specific package	Primary or any secondary authorization ID of the grantor	EXECUTE on package with GRANT OPTION	SYSADM PACKADM with GRANT OPTION on this collection or all collections	Package owner
Granting EXECUTE privilege on all packages in a collection (COL.*)	Primary or any secondary authorization ID of the grantor	N/A	SYSADM PACKADM with GRANT OPTION on this collection or all collections	N/A

In addition to package-related authorizations, the plan owner must have EXECUTE privilege on packages listed in the PKLIST. This privilege is checked at bind time, regardless of the VALIDATE option, for each package list entry. For VALIDATE(RUN), if the plan owner's privileges are not resolved at bind time, authorization ID checking is done at each execution of the plan.

If you use a wild card, such as COL.*, to specify packages in the PKLIST, the plan owner should have EXECUTE authority on COL.*. For VALIDATE(RUN), authorization checking can be postponed to runtime. At runtime, the plan owner should have authorizations either on each package that is going to be executed or on COL.*.

Important: We strongly recommend that the plan owner has the EXECUTE privilege on each package list entry before binding the plan to avoid authorization checking at execution time.

9.2 ENABLE/DISABLE

In this section, we discuss how we can prevent packages intended for a secure environment from being used in other environments.

Consider the following simple scenario. A CICS program accesses a security control table using the user ID of the person signed on, to verify that the person possesses authority to read sensitive payroll information. If authorized, it lets the user proceed, otherwise it displays a “not authorized” message. In the CICS environment, the signing on by the user is required and the host variable is always set to the signed on user ID by the application. In this environment, this application is believed to be quite secure and execute authority is granted to a wide audience, perhaps to PUBLIC.

Now consider an enterprising programmer who copies the precompiler output of this program and makes one minor change. Instead of setting the host variable to the user ID, it is now set to a super-ID (who does have such privileges). In essence, the user is now pretending to be someone they are not. They can then create a new load module and adjust the consistency token to match the consistency token of the CICS application. By executing this new application in batch by pointing to the newly-created LOADLIB, the user could gain access to the sensitive payroll information which that would otherwise not get.

You can prevent exploitation of this situation by enabling the execution of the package only in authorized environments (such as CICS). When enabled in this manner, any attempt to execute this package outside of the intended environment results in an error. This is illustrated in 7.4.9, “Package or plan execution failure: Disabled in this environment” on page 157.

This control is accomplished by using the keyword ENABLE or DISABLE during the BIND or REBIND. For remote applications, you can also control access by location name and additionally, by LUNAME. See 2.4.2, “ENABLE and DISABLE in BIND PACKAGE and BIND PLAN” on page 29 for details.

9.3 Controlling security for dynamic SQL applications

One of the primary reasons cited for avoiding dynamic SQL is the potential security exposure and the extra burden of maintaining authorizations. In this section, we discuss the security aspects of using dynamic SQL. We examine the options and possible solutions to limit the security exposure in a server-based, client/server and Web-enabled environment.

9.3.1 Possible security exposures with dynamic SQL

Consider an application program that contains the following *embedded static SQL statement:*

```
UPDATE      DSN8710.EMP
SET         SALARY = SALARY + :RAISE
WHERE       EMPNO = :EMPNO
```

A user running this program needs to have only an *execute authority on the plan or package* that contains this program. No explicit authorization to update the EMP table directly is needed and typically not granted. This means that the user can issue this statement from *within* the application but cannot do so from *outside* the application (such as through SPUFI or QMF).

Now consider a similar program that contains a similar *embedded dynamic SQL statement*:

```
UPDATE      DSN8710.EMP
SET         SALARY = SALARY + ?
WHERE       EMPNO = ?
```

In this case, the user needs *execute authority on the plan or package* (as before). Assuming we use the default bind option of DYNAMICRULES(RUN), the user requires an *explicit authorization to issue the update on the specific table*. This has two implications:

- All authorizations for users on the objects they access via dynamic SQL must be maintained, creating *extra work* for the DBA not needed when using static SQL
- Such users can bypass the application and issue the statements directly *outside of the application* (such as via SPUFI or QMF, if they are authorized to use the SPUFI or QMF plan).

In the rest of the section, we discuss alternatives that address these concerns. The most comprehensive method of dealing with dynamic SQL is available via roles which is discussed in 9.6, “Packages using roles in a trusted context” on page 188.

9.3.2 Using DYNAMICRULES(BIND)

In Chapter 2, “BIND options” on page 17, we discuss the various options for this bind parameter and their meaning. In general, using DYNAMICRULES(BIND) makes dynamic SQL appears like static SQL from an authorization perspective. A program bound with DYNAMICRULES(BIND) uses the authorization of the package or plan owner to determine whether or not the operation is permitted. For the previous example, if the owner of the package has authority to update the EMP table, so does the user executing the package.

In any environment where the user cannot issue any free-form SQL (that is, all generated SQL is controlled by the application), this option is easy to implement and virtually eliminates the security exposure. When a user is allowed to issue any free-form SQL, care must be taken to make sure that the package is bound by a user who has *just the sufficient authority*. For example, if the package is bound by a user with the SYSADM authority, *all* dynamic SQL are authorized—equivalent to granting access to all objects to PUBLIC, which is probably not the intent. We discuss this further in 9.3.5, “Controlling dynamic SQL security” on page 182. Trusted contexts can also help with this situation, as explained in 9.6, “Packages using roles in a trusted context” on page 188.

9.3.3 Using stored procedures to overcome security issues

If a stored procedure containing only static SQL is used to replace a distributed application, the only authorization that a user needs is the ability to execute the stored procedure (EXECUTE ON PROCEDURE) and the ability to execute the stored procedure package (EXECUTE authorization on the package). This means no grants to the base objects (tables) have to be issued, nor can the user access the base objects outside of the application. However, this option is only possible if the entire business logic can be encapsulated in the stored procedure.

9.3.4 Impact on ODBC/JDBC applications

Like applications using embedded SQL, applications using JDBC or ODBC need DB2 packages to run. In this section, we describe how such ODBC/JDBC packages are different from an application using embedded dynamic SQL and why DYNAMICRULES(BIND) might not provide the granularity you wanted, as well as how to overcome this limitation.

Here are some major differences between ODBC/JDBC packages and packages generated from an embedding application:

- In the embedded approach, the packages are tied to the application and are not usable outside the programs for which they are created.
- In ODBC or JDBC applications, the packages are generic and can be used by *any* C, C++ (for ODBC) or Java (for JDBC) programs.

For this reason, binding DB2 ODBC or JDBC packages with the DYNAMICRULES(BIND) option does not provide the same security level as obtained with embedded SQL. More precisely, when using this bind option, you must be aware of the following considerations:

- Users of an ODBC/JDBC application inherit the DB2 authorizations of the package owner, each time they use the ODBC or JDBC interface.
- As a consequence, they can create their own ODBC/JDBC programs, or use any ODBC/JDBC enabled tool to do any data manipulations that the ODBC/JDBC package owner is authorized to do.
- Moreover, if two applications use the same ODBC/JDBC packages, users of one application can use programs or tools to manipulate data belonging to the other application.

To solve the latter problem, you can isolate your ODBC applications by creating a set of ODBC/JDBC packages for each application:

- At bind time, choose a collection name that represents the application, and choose an bind owner who has the right authorizations on the application objects.
- At runtime, use the ODBC keyword COLLECTIONID (on z/OS) or CURRENTPACKAGESET (on the client) to point to the correct packages.

A more detailed analysis of the possible security issues in ODBC/JDBC applications is discussed next in 9.3.5, “Controlling dynamic SQL security”.

9.3.5 Controlling dynamic SQL security

Table 9-5 summarizes the possible security issues when an application containing dynamic SQL statements is *directly run by the end users*, without passing through an application server. More precisely, the analysis is based on the following three assumptions:

- The application does not call stored procedures or user-defined functions. The security implication of this are dealt with in 9.3.3, “Using stored procedures to overcome security issues” on page 181 and 9.5, “Dynamic SQL statements in stored procedures” on page 185.
- End users are directly connected to DB2 to run the SQL processor program. They are consequently granted the authorization to EXECUTE the DB2 package or plan.
- The SQL processor implements either of these interfaces:
  - A generic interface allowing any free form SQL statements without any programmed control. Packages as JDBC, ODBC, QMF and SPUFI belong to this category.
  - A restricted interface that controls the SQL statements submitted to the database.

Note:

- If a BIND or REBIND was done by a role in a trusted context, all subsequent BINDs or REBINDs must also be done by a role in a trusted context. For discussion of roles, see 9.6, “Packages using roles in a trusted context” on page 188.

Table 9-5 Controlling dynamic SQL security

Does the user have DB2 authorization to objects accessed?	DYNAMIC-RULES bind option	Does the SQL processor allow free-form SQL?	Within SQL processor	Outside SQL processor
YES	RUN	YES	Access to authorized objects and any SQL	
		NO	Access to authorized objects and SQL controlled by SQL processor	Access to authorized objects and any SQL allowed
	BIND	YES	Access to objects bind owner is authorized for and any SQL allowed	Access to authorized objects and any SQL allowed
		NO	Access to objects bind owner is authorized for and SQL controlled by SQL processor	Access to authorized objects and any SQL allowed
NO	RUN	YES	No access - meaningless	
		NO		
	BIND	YES	Access to objects bind owner is authorized for and any SQL allowed	No access
		NO	Access to objects bind owner is authorized for and SQL controlled by SQL processor	

Assuming that a secure application is one that allows the user to operate only within the application, but not outside, and that the application controls what the user is allowed to do. From Table 9-5, the following conclusions can be drawn:

- If the SQL processor does not use a generic interface such as ODBC or JDBC and does not use any application such as QMF or SPUFI that allows free-form SQL, a high level of security can be obtained by the DYNAMICRULES(BIND) option if the user is not given any explicit authorization on application tables. The security model is similar to the one for static SQL.
- If the SQL processor makes use of a generic interface such as ODBC or JDBC or allows free form SQL, there is *no* secure implementation based on any combination of authorizations and bind options. This is one of the reasons for incorporating roles in a trusted context, which is discussed in 9.6, “Packages using roles in a trusted context” on page 188.

9.4 Privileges required to execute stored procedures

After a stored procedure has been created, it is most likely to be executed by a number of DB2 users. Two types of authorizations are required:

- Authorization to execute the CALL statement:

  The privileges required to execute the CALL depend on several factors, including the way the CALL is invoked. The CALL can be dynamic or static:

  - The dynamic invocation of a stored procedure is whenever the CALL is executed with a host variable, as shown here:

    CALL :*host-variable*

  - The static invocation of a stored procedure is whenever the CALL is executed with a procedure name, as shown here:

    CALL *procedure-name*

- Authorization to execute the stored procedure package and any dependent packages:

  The privilege required to EXECUTE the package is independent of the type of CALL.

In places where we mention “authorization ID” in this chapter, we are referring to the value of CURRENT SQLID. This is usually the authorization ID of the process, meaning the authorization ID being passed from the client in the case of a distributed call, or the authorization ID associated with a CICS transaction or batch job if the stored procedure is called from CICS or batch. The authorization ID might be changed if the calling application issues a SET CURRENT SQLID statement prior to issuing the CALL to the stored procedure.

With DB2 9, the GRANT EXECUTE statement allows for the privilege on a stored procedure to be granted to a ROLE. ROLE privileges or executing stored procedures are considered in addition to the value of CURRENT SQLID for batch, local and remote applications. Trusted contexts cannot be defined for CICS and IMS.

9.4.1 Privileges to execute a stored procedure called dynamically

For static SQL programs that use the syntax CALL host variable (ODBC applications use this form of the CALL statement), the authorization ID of the plan/package which contains the CALL statement or an assigned role[1] must have one of the following possibilities:

- The EXECUTE privilege on the stored procedure
- Ownership of the stored procedure
- SYSADM authority

9.4.2 Privileges to execute a stored procedure called statically

For static SQL programs that use the syntax CALL procedure, the owner (an authorization ID or a role) of the plan or package that contains the CALL statement must have one of the following possibilities:

- The EXECUTE privilege on the stored procedure
- Ownership of the stored procedure
- SYSADM authority

It does not matter whether the current authorization ID at execution time has the EXECUTE privilege on the stored procedure. As long as the authorization ID has EXECUTE authority on the plan or package of the calling application, it is able to execute any CALL statements within the calling application. This privilege is checked at the time the plan or package for the calling application is bound, unless VALIDATE(RUN) is used.

[1] The DYNAMICRULES behavior for the plan or package that contains the CALL statement determines both the authorization ID and the privilege set that are held by that authorization ID.

9.4.3 Authorization to execute the stored procedure packages

DB2 checks the following authorization IDs in the order in which they are listed for the required authorization to execute the stored procedure package (in each case an owner is either an authorization ID or a role):

1. The owner (the definer) of the stored procedure.
2. The owner of the plan that contains the statement that invokes the stored procedure package if the application is local or if the application is distributed and DB2 for z/OS is both the requester and the server.
3. The owner of the package containing the statement that invokes the stored procedure package if the application is distributed and DB2 for z/OS is the server but not the requestor, or the application uses Recoverable Resources Management Services attachment facility (RRSAF).
4. The owner of the package that contains the statement that invokes the package if the application is distributed and DB2 for z/OS is the server but not the requester.
5. The authorization ID as determined by the value of the DYNAMICRULES bind option for the plan or package that contains the CALL statement if the CALL statement is in the form of CALL host variable. Prior to DB2 9 for z/OS, when DYNAMICRULES(RUN) was specified, then DB2 checked the CURRENT SQLID, the primary authorization ID (if different) and its secondary authorization IDs. DB2 9 for z/OS also includes the user's role in effect.

The privilege required to run the stored procedure package and any packages that are used under the stored procedure is any of the following possibilities:

- The EXECUTE privilege on the package
- Ownership of the package
- PACKADM authority for the package's collection
- SYSADM authority

A PKLIST entry is not required for the stored procedure package.

In case of stored procedures invoking triggers and UDF, additional authorizations are required. See *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

9.5 Dynamic SQL statements in stored procedures

We know that stored procedures can be called by dynamic SQL programs, and can execute their work using static SQL within the procedures, and so derive the security strengths of the static SQL model. You can grant execute privilege on the procedure, rather than access privileges on the tables that are accessed in the procedures. An ODBC or JDBC application can issue a dynamic CALL statement, and invoke a static stored procedure to run under the authority of the package owner for that stored procedure.

Stored procedures with dynamic SQL are also good for security reasons, but they require a bit more effort to plan the security configuration.

All of the security topics we have discussed so far are applicable to stored procedures that are created with and contain static SQL. Stored procedures that contain dynamic SQL are influenced by the DYNAMICRULES option in effect at the time that the stored procedure package is bound.

Tip: In order to decide attributes such as default QUALIFIER or authorization ID used, you need to be aware of two things:

- What environment is this statement run in?
- What is the value of DYNAMICRULES?

The run-time environment, in combination with DYNAMICRULES option, determines what values apply at run-time for dynamic SQL attributes such as authorization ID for authorization checking, and qualifier for unqualified objects, as well as some other attributes. The set of attribute values is called the *dynamic SQL* statement behavior.

Statement behaviors

These are the four dynamic SQL statement behaviors:

- Run behavior
- Bind behavior
- Define behavior
- Invoke behavior

Each behavior represents a different set of attribute values that impact how authorizations are handled for dynamic SQL.

DYNAMICRULES

The authorization processing for dynamic SQL in a stored procedure is impacted by the value of the DYNAMICRULES parameter when the stored procedure package was bound. There are six possible options for the DYNAMICRULES parameter:

- BIND
- RUN
- DEFINEBIND
- DEFINERUN
- INVOKEBIND
- INVOKERUN

If you bind the package for the stored procedure with DYNAMICRULES(BIND), then the dynamic SQL in the stored procedure is also authorized against the package owner for the dynamic SQL program.

For each of the other values, the authorization for dynamic SQL in a stored procedure is checked against an authorization ID[2] other than the package owner.

Table 9-6 shows how DYNAMICRULES and the run-time environment affect dynamic SQL statement behavior when the statement is in a package that is invoked from a stored procedure (or user-defined function).

Table 9-6 How is runtime behavior determined?

DYNAMICRULES value	Stored procedure or user-defined function environment	Authorization ID
BIND	Bind behavior	Plan or package owner
RUN	Run behavior	Current SQLID

[2] This also applies to role as well as authorization ID, see 9.6, “Packages using roles in a trusted context” on page 188.

DYNAMICRULES value	Stored procedure or user-defined function environment	Authorization ID
DEFINEBIND	Define behavior	Owner of user-defined function or stored procedure
DEFINERUN	Define behavior	Owner of user-defined function or stored procedure
INVOKEBIND	Invoke behavior	Authorization ID of invoker
INVOKERUN	Invoke behavior	Authorization ID of invoker

The DYNAMICRULES option, along with the runtime environment of a package (whether the package is run stand-alone or under the control of a stored procedure or user-defined function), determines: the authorization ID used to check authorization, the qualifier for unqualified objects, the source of application programming options for SQL syntax, and whether or not the SQL statements can include DDL/DCL such as GRANT, REVOKE, ALTER, and CREATE statements.

Table 9-7 shows the implications of each dynamic SQL statement behavior.

Table 9-7 What the runtime behavior means

Dynamic SQL attribute	Bind behavior	Run behavior	Define behavior	Invoke behavior
Authorization ID	Plan or package owner	Current SQLID, secondary authorization IDs and role in effect are checked	Owner of user-defined function or stored procedure	Authorization ID of invoker
Default qualifier for unqualified objects	Bind OWNER or QUALIFIER value	CURRENT SCHEMA, which defaults to CURRENT SQLID if not explicitly set	Owner of user-defined function or stored procedure	Authorization ID of invoker
CURRENT SQLID	Not applicable	Applies	Not applicable	Not applicable
Source for application programming options	Determined by DSNHDECP parameter DYNRULS	Install panel DSNTIPF	Determined by DSNHDECP parameter DYNRULS	Determined by DSNHDECP parameter DYNRULS
Can execute GRANT, REVOKE, ALTER, DROP, RENAME	No	Yes	No	No

Use the value appropriate for your environment. In a DB2 for z/OS server-only environment, DYNAMICRULES(BIND) makes embedded dynamic SQL behave similarly to embedded static SQL, and is probably the best option for most users if the users are not allowed to use “free-form SQL”. In a distributed environment, binding multiple packages using different levels of authorization might provide the best granularity. Figure 9-2 shows how complex the choices can be when invoking a stored procedure.

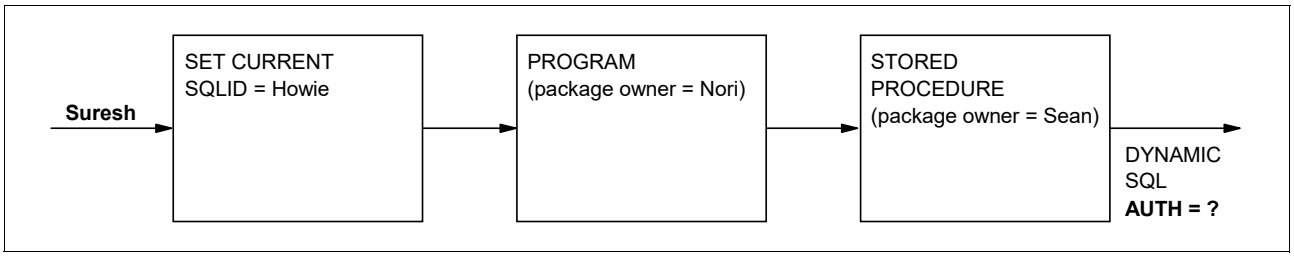

Figure 9-2 Security implications of dynamic SQL in a stored procedure

Consider a user, Suresh, using a CURRENT SQLID of Howie, who executes a package bound by Nori. This package calls a stored procedure created and bound by Sean. Whose authority is checked at runtime? Depending on the option chosen, the authorization ID used to determine whether or not the execution of dynamic SQL within this stored procedure is permitted could be:

- Suresh (invoker)
- Howie (current SQLID)
- Nori (owner of package) or
- Sean (owner of stored procedure)

Due to the variety of options available, it is difficult to make a general recommendation that applies to all situations. See *DB2 Version 9.1 for z/OS9 Administration Guide,* SC18-9840 for more details on the DYNAMICRULES option of the BIND command to help you determine which value is appropriate for your application.

9.6 Packages using roles in a trusted context

We first discuss some of the security challenges we face in today's DB2 environment. Then we introduce the concept of roles within a trusted context. We discuss how the client drivers can exploit this new feature and then summarize how best practices can help us overcome the challenges and secure the DB2 environment.

For details of this topic, see *Securing DB2 and Implementing MLS on z/OS,* SG24-6480-01.

Let us begin with a simple example.

A DBA has been authorized to implement a schema change over the weekend. We stipulate that the authority is granted provided the DBA connects from a specific IP address (home PC), using a specified encryption level and executes a specific change management tool. In addition, the authority is made available only between 1 am and 3 am on Sunday morning. Only if all of these conditions are met, the DBA has the authority to implement the schema change. Notice how this is different from the environment without roles and trusted contexts. The DBA otherwise needs to possess a SYSADM or DBADM privilege all the time and make any other changes from any other connection using any application.

This business need leads us naturally to the concept of a trusted context where we can meet each of the requirements we specified previously.

9.6.1 DB2 security challenges

We now provide a summary of existing challenges that lead us to use the newly provided DB2 9 trusted context and role capabilities.

Trust all connection requests?

DB2 receives requests from many external entities. Currently, you have the option to set a system parameter that indicates to DB2 that all connections are to be trusted. Trusted means that users do not have to be authenticated to RACF with credentials. It is unlikely that all connection types, such as DRDA, RRS, TSO, and batch, from all sources would fit into this category.

Application server user ID/password

Most existing application servers connect to DB2 using a single user ID to initiate requests to DB2. This ID must have all administrative privileges and the privileges required to execute the business transactions. If someone steals that user ID and password, that person can exercise those privileges from another place in the network. This is a significant exposure.

It also means diminished accountability. For example, there is no ability to separate access performed by the middleware server for its own purposes from those performed on behalf of users.

Here we describe some characteristics of the common, three-tier architecture:

- The middle layer (sometimes called the middleware layer) authenticates users running client applications.
- The middle layer also manages interactions with the database server (DB2).
- The middle layer's user ID and password are used for authentication purposes.
- The database privileges associated with that authorization ID are checked when accessing the database, including all access on behalf of all users.
- The middle layer's user ID must be granted privileges on all the resources that might be accessed by user requests.

Figure 9-3 illustrates the three-tier architecture.

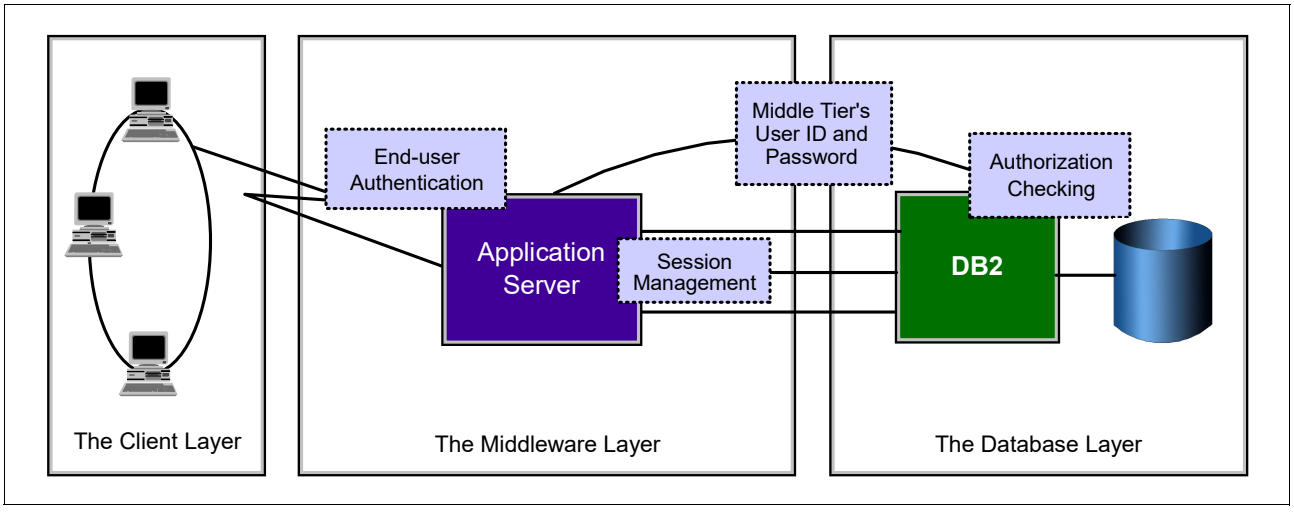

Figure 9-3 Three-tier architecture

Dynamic SQL auditability

Looking at Figure 9-3, now we discuss the reasons that led to this configuration and the resulting weak auditing capability.

The middleware layer's authorization ID is used to access DB2 so that customers do not have to grant dynamic SQL privileges directly to their users. DB2 only authenticates and knows about a single authorization ID, which makes all the requests, when, in fact, there are multiple users initiating requests through the application. A consequence of this approach is diminished accountability; there is no ability in DB2 to separate access performed by the middleware layer's authorization ID server for its own purposes from those performed on behalf of users.

Business user authentication is performed by the middleware component and not in DB2, which means a loss of control over user access to the database.

Another reason for using a three-tiered approach is connection pooling in the application server. It takes time to simply create a new connection to the database server and re-authenticate the user at the database server. This can be a problem for middleware servers that do not have access to user credentials, such as IBM Lotus® Domino®.

Securing DBA privileges

We describe some challenges and new opportunities for securing DBA privileges.

Shared SYSADM ID

In some shops, certain DBAs have powerful user IDs that hold administrative privileges such as SYSADM or DBADM. Although they might only use certain privileges infrequently, they typically hold these privileges all the time. This increases the risk of adversely affecting DB2 objects or table data when an error or mistake occurs. Such a situation is potentially open to malicious activity or fraud.

Suppose a team of DBAs share a generic user ID that holds SYSADM or DBADM privileges. When a DBA leaves the team for a new internal or external job, there is no cascading effect. However, by using a shared ID there is no individual accountability, which is a Sarbanes-Oxley requirement.

Additionally, many customers are concerned about DBA access to sensitive data.

Dual responsibilities

A DBA might have a dual job function, where they maintain databases in both development and production using the same user ID in both environments. In this scenario, a mistake can occur when making changes if a DBA thinks they are connected to development, when they are really connected to production.

Full-time access to sensitive/private data by DBA

When DBAs create tables, they have full access to the contents of the tables all of the time.

DBADM can create view for another ID, but cannot drop/alter

An authorization ID, with DBADM privilege, under certain circumstances can create views for others. However, this same authorization ID is not able to drop or grant privileges on these views to others. Because views do not have an underlying database, the DBADM privilege cannot be recognized for views.

Without reserving a RACF group, a table can be dropped

When the creator of a table is not a user ID, and a RACF group has not been set up with same name as the creator, it is possible for someone else to create the group, connect themselves to it, and drop the table. Another user ID can do this with basic access to DB2 and without any grants received on that table.

Privileges granted can be exercised from anywhere

Before trusted contexts and roles, privileges held by an authorization ID were universal, irrespective of context. This can weaken security because a privilege can be used for purposes other than those originally intended. For example, if an authorization ID is granted SELECT on a payroll table, the authorization ID can exercise that privilege regardless of how it gains access to the table.

9.6.2 Roles and trusted contexts

Roles in a trusted context were created to overcome the limitations discussed previously. A trusted context is a new object used to control users and applications accessing DB2. A role is a new object which can be granted privileges or own objects but the role is associated with a DB2 process only when the application is associated with a trusted context. It is NOT a group. For example, this allows a user to have SYSADM privileges only when in trusted context, but not outside of this context.

To create a trusted context, you specify a system ID and connection attributes necessary to associate a trusted context to a connection. This can include:

- IP address or host name of a remote application
- Job name of a local application
- Encryption requirements

You then specify an optional list of users who can be associated with the trusted connection and authentication requirements for these users.

An application can be associated with a trusted context using one of the following connections:

- DDF
- RRS attach
- DSN ASUSER
- ROLE jobname

After the trusted connection is established, you can switch the user associated with the connection without requiring credentials.

9.6.3 Impact of roles on plan and package ownership

When BIND or REBIND are issued in a trusted context specified with ROLE AS OBJECT OWNER, ownership is determined as follows:

- If OWNER is not specified, the role associated with the binder becomes the owner.
- If OWNER is specified, the role specified in the OWNER option becomes the owner (in a trusted context, the OWNER specified must be a role).

Note the following plan and package ownership considerations:

- For a package to be bound remotely with a role as the owner of the package at the remote DB2, the trusted context at the remote DB2 must be specified as WITH ROLE AS OBJECT OWNER.
- BIND/REBIND on the package must be performed in a trusted context.
- If OWNER is specified for a remote BIND across a trusted connection, OWNER can be a role or an authorization ID. Outbound authorization ID translation is not performed for the OWNER.
- If the plan owner is a role and the application uses a package bound at a remote DB2 server, the plan owner privilege to execute the package is not considered at the remote DB2 server.

  The package owner or the user executing the package (as determined by the value of DYNAMICRULES) at the DB2 server must have the EXECUTE privilege on the package at the server.

9.6.4 Trusted context sample scenarios

In this section we provide samples of trusted context and role utilization.

Remote execution of a package

Example 9-1 is an example that fits the 3 tier scenario depicted in Figure 9-3 on page 189.

The trusted context is named LOAN_APP and it is reachable from 9.10.30.237, the IP address of the Windows server.

The system authorization ID that is allowed to establish a connection through the trusted context is LOAN_FUNC_ID.

SAMTA is the business user at a Windows PC.

Via the trusted connection, SAMTA acquires the role ROLE1 and is allowed to execute package DSN9CC13.CALCINT.

Example 9-1 Package remote execution

```
CREATE ROLE ROLE1;

GRANT EXECUTE ON PACKAGE DSN9CC13.CALCINT TO ROLE ROLE1;

CREATE TRUSTED CONTEXT LOAN_APP
BASED UPON CONNECTION USING SYSTEM AUTHID LOAN_FUNC_ID
ATTRIBUTES (ADDRESS '9.10.30.237')
WITH USE FOR SAMTA ROLE ROLE1
ENABLE;
```

Controlling BINDing packages in a batch production environment

Suppose that new packages need to be bound for the LOAN application during a 2 hour window on the weekend. Example 9-2 shows how you can limit:

- The source of the BIND command (for collection LOAN_COLL)
- Who can BIND
- When a BIND can be done

Example 9-2 Limiting batch BINDs

```
CREATE ROLE PROD_BATCH_BINDER;

GRANT PACKADM ON COLLECTION LOAN_COLL TO ROLE PROD_BATCH_BINDER;

CREATE TRUSTED CONTEXT FOR_BINDING_PRODUCTION_PACKAGES_IN_BATCH
BASED UPON CONNECTION USING SYSTEM AUTHID USRT001
ATTRIBUTES (JOBNAME 'BINDPRD')
DEFAULT ROLE PROD_BATCH_BINDER
WITH ROLE AS OBJECT OWNER
DISABLE;
```

A ROLE is used and the ROLE will be the OWNER of the packages.

During the weekend, at the beginning of the 2 hour implementation window, a USER with SYSADM executes the following:

```
ALTER TRUSTED CONTEXT FOR_BINDING_PRODUCTION_PACKAGES_IN_BATCH ALTER ENABLE ;
```

When USRT001 submit a batch job with the name BINDPRD, a trusted context is established, USRT001 acquires the PROD_BINDER ROLE, and is able to BIND packages into the LOAN_COLL collection.

Binding into this collection can only happen via local job submitted by this user. For example BINDING into this collection cannot be done from the “BIND PACKAGE” function available via the “DB2I (DB2 Interactive) facility” (panel in TSO), nor can it be done from a remote location.

In this example all packages bound via this trusted context will be owned by ROLE PROD_BATCH_BINDER

After the completion of the implementation, a user with SYSADM executes the following command:

```
ALTER TRUSTED CONTEXT FOR_BINDING_PRODUCTION_PACKAGES_IN_BATCH ALTER DISABLE;
```

By having roles own packages tighter controls can be implemented, packages can only be bound when the appropriate trusted context is enabled.

Limiting package BINDs to TSO interface

Example 9-3 shows how to limit BINDing packages for the payroll applications. Such tasks will only be allowed via the TSO DB2I interface. A ROLE is used and the ROLE is the OWNER of the packages

Example 9-3 Allow package BINDs only via TSO

```
CREATE ROLE PROD_TSO_BINDER;

GRANT PACKADM ON COLLECTION PAYROLL_COLL TO ROLE PROD_TSO_BINDER;

CREATE TRUSTED CONTEXT FOR_BINDING_PACKAGES_IN_PRODUCTION
BASED UPON CONNECTION USING SYSTEM AUTHID USRT001
ATTRIBUTES (JOBNAME 'USRT001')
DEFAULT ROLE PROD_TSO_BINDER WITH ROLE AS OBJECT OWNER
ENABLE;
```

1. The user USRT001 logs on to TSO.
2. The trusted connection is established because both the SYSTEM AUTHID and the TSO JOB attributes match the attributes for trusted context:

   FOR_BINDING_PACKAGES_IN_PRODUCTION

   This trusted connection exists for the duration of the logon session.
3. USRT001 acquires the role PROD_TSO_BINDER. USRT001 uses the DSN command processor and runs multiple BIND commands.
4. The BIND commands gets executed and role PROD_TSO_BINDER becomes the owner of the packages.

Packages for the PAYROLL_COLL collection can only be bound using ID USRT001 via TSO.

Controlling execution of an application's batch packages

In Example 9-4 we grant the privilege to run all packages in collection FOREIGN_EXC to role OFFLINE_ROLE.

Example 9-4 Allow a window for package BINDs via batch

```
GRANT EXECUTE ON PACKAGE FOREIGN_EXC.* TO ROLE OFFLINE_ROLE;

CREATE TRUSTED CONTEXT OVERNIGHT_BATCH
BASED UPON CONNECTION USING SYSTEM AUTHID OFFLN01
ATTRIBUTES (JOBNAME 'FXPART1')
DEFAULT ROLE OFFLINE_ROLE
DISABLE;
```

1. The offline window begins, and the context is ENABLED.
2. User OFFLN01 submits a batch job called FXPART1, a trusted context is established, user OFFLN01 acquires the ROLE OFFLINE_ROLE and is able to run the steps that execute the relevant packages within the FOREIGN_EXC collection.
3. The offline window closes, and the context is DISABLED.

User OFFLN1 can only execute the FOREIGN_EXC.* packages from this interface and only during the time the trusted context is enabled.

Bind package locally and remotely with role ownership

If a role needs to own a package locally and remotely, a trusted context can be defined at the local and the remote DB2 with role ROLELAR and the ROLE AS OBJECT OWNER clause. The role ROLELAR needs to have the necessary privileges to bind the package. The local communications database is set up to request a trusted connection to the remote location.

At the local DB2, the trusted context can be defined as in Example 9-5.

Example 9-5 Package owner is a local ROLE within the trusted context

```
CREATE ROLE ROLELAR;

CREATE TRUSTED CONTEXT LCLCTX
BASED UPON CONNECTION USING SYSTEM AUTHID USERAPPL
ATTRIBUTES (JOBNAME 'USERAPPL')
DEFAULT ROLE ROLELAR WITH ROLE AS OBJECT OWNER
ENABLE;
```

At the remote DB2 such a trusted context could be defined as in Example 9-6.

Example 9-6 Package owner is a remote ROLE within the trusted context

```
CREATE ROLE ROLELAR;

CREATE TRUSTED CONTEXT REMOTECTX
BASED UPON CONNECTION USING SYSTEM AUTHID USERAPPL
ATTRIBUTES (ADDRESS '9.30.123.196')
DEFAULT ROLE ROLELAR WITH ROLE AS OBJECT OWNER
ENABLE;
```

1. The user USERAPPL logs on to TSO, uses the DSN command processor, and runs the following BIND commands:

   ```
   BIND PACKAGE(SVLREMOTE.TEST02) MEMBER(TEST02) ACTION(REP) QUALIFIER(USERAPPL)
   BIND PACKAGE(SVLLABS.TEST02) MEMBER(TEST02) ACTION(REP) QUALIFIER(USERAPPL)
   BIND PLAN(TESTPL2) PKLIST(*.*.TEST02) ACTION(REP) QUALIFIER(USERAPPL)
   ```

2. The DSN command processor connects to the local DB2.
3. DB2 determines SVLREMOTE is a remote location and performs the following:
   a. The DB2 requester sends a request to establish a trusted connection using the authentication information for USERAPPL.
   b. Assuming USERAPPL is authorized, the DB2 remote server finds a matching trusted context with primary authorization ID USERAPPL and address 9.30.123.196 and the connection is established as trusted. The role ROLELAR gets associated with the trusted connection.
   c. The BIND command gets executed and role ROLELAR becomes the owner of the package at the remote location.
4. DB2 determines SVLLABS is a local location and performs the following:
   a. Assuming USERAPPL is authorized, DB2 finds a matching trusted context, LCLCTX, with primary authorization ID USERAPPL and jobname USERAPPL and establishes the connection as trusted. The role ROLELAR gets associated with the trusted connection.
   b. The BIND command gets executed and role ROLELAR becomes the owner of the package and plan.

Client exploitation

CLI and JDBC client driver APIs support trusted contexts.

Using JDBC, you can issue the commands listed in Example 9-7.

Example 9-7 Setting up trusted content with JDBC

```
// Create a DB2ConnectionPoolDataSource instance
      com.ibm.db2.jcc.DB2ConnectionPoolDataSource dataSource = new
com.ibm.db2.jcc.DB2ConnectionPoolDataSource();
      // Set properties for this instance
      dataSource.setDatabaseName("DB9A");
      dataSource.setServerName("wtsc63.itso.ibm.com");
      dataSource.setDriverType(4);
      dataSource.setPortNumber(12347);

      // Set other properties using
      // properties.put("property", "value");
      java.util.Properties props = new java.util.Properties();
```

```
// Supply the user ID and password for the connection
String user = "PAOLOR1";
String password = "abcdefgh";

// Call getDB2TrustedPooledConnection to get the trusted connection
// instance and the cookie for the connection
Object[] objects = dataSource.getDB2TrustedPooledConnection(user, password, props);
javax.sql.PooledConnection pooledcon = (javax.sql.PooledConnection) objects[0];
byte[] cookie = (byte[]) objects[1];

// get connection
Connection con = pooledcon.getConnection();
```

In the WebSphere Application Server environment, you set the datasource property to:

`propagateClientIdentityUsingTrustedContext`

9.6.5 Summary

The enhancements described in this section strengthen DB2 security in the z/OS environment.

Roles and trusted contexts provide an opportunity for system administrators to control access to enterprise objects at a level of abstraction that is close to the structure of their enterprise.

These new database entities and the concept of trusted connections provide considerable benefits:

- Manageability:
  - Allows database administrators to manage and control access from network-attached middleware servers.
  - Reusing a connection without the need for DB2 to authenticate the new user provides the ability to switch without a password. This eliminates the need to manage multiple passwords at the client.
  - The role object ownership in a trusted context eliminates the object dependency on authorization IDs.
  - Authorization IDs already acquire the necessary database privileges that are associated with either an assigned role or a default role specific to the trusted context, simplifying management of dynamic SQL privileges.
  - Allows an authorization ID to acquire a special set of privileges that are not available to it outside the trusted context.
- User accountability without compromising performance:
  - Knowing the user's identity provides improved data access auditing capability. Therefore, the middleware server can distinguish the transactions performed by the middleware server for its own purpose from those performed on behalf of some user.
  - The role object ownership in a trusted context gives the ability to customers to eliminate shared user IDs and have each person audited and accountable with their own authorization IDs.

- Improved security:
  - With trusted context, administrators can configure their system in such a way that the SYSTEM authorization ID has all the privileges only when used from the trusted context. This eliminates the concern about the theft of the middle tier SYSTEM authorization ID. Even if someone learns the SYSTEM authorization ID credentials, they would not be able to penetrate the DB2 server unless they also had the ability to penetrate the trusted middle tier server as well.
  - Roles holding privileges can only be exercised within a trusted connection.
  - With roles in a trusted context, privileges granted to authorization IDs can be restricted, which improves security, allowing the authorization IDs to perform only the approved activities.

Roles in a trusted context provide you the ability to control where, when, and how local and remote requesters communicate with DB2.

10

Access path management

In this chapter we discuss tools and techniques to obtain a desirable access path and to stabilize it after a rebind.

We cover the following topics:

- Introduction
- Using Optimization Service Center to view access paths
- Guidelines for a rebind strategy
- Change control for access paths
- Simulating possible access path changes
- VERSION(AUTO) fallback
- Using a try-out collection when promoting with a blank VERSION value
- REOPT (NONE / ALWAYS / ONCE / AUTO)
- Optimization hints
- SQL tricks
- Materialized Query Tables (MQT)
- Catalog statistics manipulation
- Package stability
- Planning and managing fallback for access paths

10.1 Introduction

Optimization of an access path for an SQL statement is a broad subject and is clearly beyond the scope of this book. Proper design and coding techniques and proper RUNSTATS strategy are essential prerequisites for a good access path. In this chapter, we explain how you can determine if any changes in access path have occurred and decide if they can cause poor performance. We cover various techniques to stabilize the access path.

As such, the focus is on stabilization, not on optimization.

You can obtain information about DB2 access paths by using the following methods:

Use Optimization Service Center for DB2 for z/OS

You can display and analyze information on access paths chosen by DB2. Optimization Service Center (OSC) displays the access path information in both graphical and text formats. The tool provides you with an easy-to-use interface to the EXPLAIN output and allows you to invoke EXPLAIN for dynamic SQL statements. You can also access the catalog statistics for the referenced objects of an access path. In addition, the tool allows you to archive EXPLAIN output from previous SQL statements to analyze changes in your SQL environment.

Refer to 11.3.5, “Optimization Service Center” on page 237 for more information.

Run IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS accounting reports

Another way to track performance is with the IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS (abbreviated to OMEGAMON PE in this book) accounting reports. The accounting report, short layout, ordered by PLANNAME, lists the primary performance figures. Check the plans that contain SQL statements whose access paths you tried to influence. If the elapsed time, TCB time, or number of GETPAGE requests increases sharply without a corresponding increase in the SQL activity, then there could be a problem. You can use OMEGAMON PE to track events after your changes have been implemented, providing immediate feedback on the effects of your changes.

Specify the EXPLAIN bind option

Specify the EXPLAIN(YES) bind option. You can also use the EXPLAIN option when you bind or rebind a plan or package. Compare the new plan or package for the statement to the old one. Investigate any changes in access path in the new plan or package; they could represent performance improvements or degradations. For example, if the new one has a table space scan or a non-matching index space scan, but the old one did not, this statement probably should be investigated.

A PLAN_TABLE must exist for the owner of the package or the current SQLID issuing the EXPLAIN command dynamically. This object does not have to be a table but can be an alias to a local table. This allows a common set of tables to be used by multiple owners. In the development environment, for example, instead of each programmer having individual PLAN_TABLE, you can create an alias to a common PLAN_TABLE which you manage.

The advantages of this method are as follows:

- Fewer objects to manage.
- One common place to obtain information for all packages.
- When the format changes (as it commonly does with a new version of DB2), it is easier to migrate to the new format.

The disadvantages of this method are as follows:

- You need to manage the space consumed regularly, because the misuse of space could prevent other users from being able to BIND with EPXLAIN(YES).
- You might lose the granularity desired. For example, user1 could accidentally update/delete rows for user2.

10.2 Using Optimization Service Center to view access paths

In this section, we provide a very quick overview of Optimization Service Center, which can help us perform reactive and proactive tuning of SQL. We also use it in 11.4, “Investigating a query from a package” on page 238. For a complete description of this product, see *IBM DB2 9 for z/OS: New Tools for Query Optimization,* SG24-7421.

10.2.1 Overview of Optimization Service Center features

IBM Optimization Service Center for DB2 for z/OS is a Windows workstation tool designed to ease the workload of DBAs by providing a rich set of autonomic tools that help optimize query performance and workloads. Optimization Service Center is built on Eclipse, an open source platform for the construction of powerful software development tools and rich desktop applications.

A sample access path graph created by Optimization Service Center is shown in Figure 10-1.

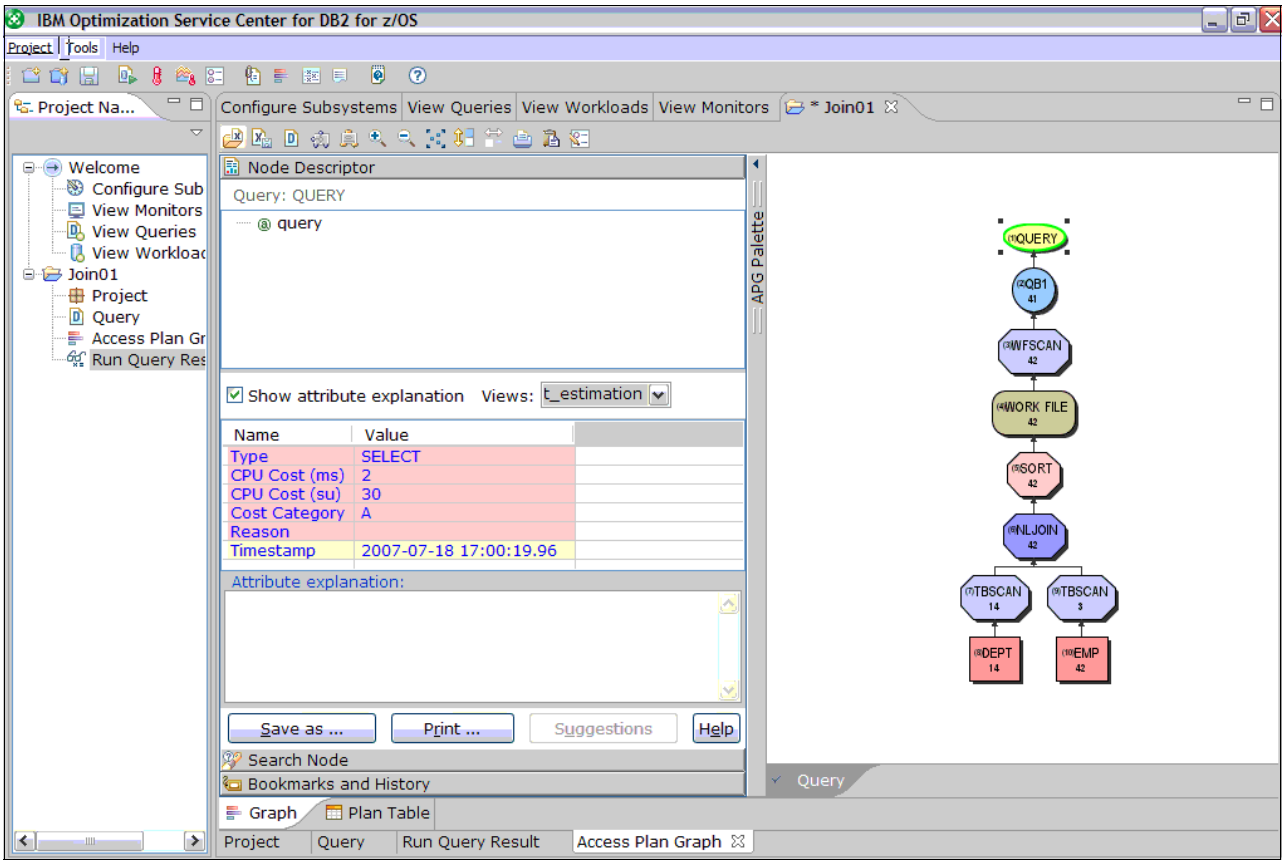

Figure 10-1 Sample access path graph generated by Optimization Service Center

10.2.2 Reactive tuning

You can use Optimization Service Center to identify and analyze problem SQL statements and to receive expert advice about statistics that you can gather to improve the performance of known problematic and poorly performing SQL statements on a DB2 subsystem.

10.2.3 Proactive tuning

You can use Optimization Service Center to identify and analyze groups of statements and receive expert advice about statistics that you can gather to improve the performance of entire SQL workloads. An SQL workload is any group of statements that run on your DB2 subsystem. You define a workload by creating a set of criteria that identifies the group of statements that make up the workload. For example, you can create a workload to capture all of the statements in a particular application, plan, or package. This can be done prior to installing the application in a production environment.

10.2.4 Advanced tuning

Optimization Service Center optimization tools allow the experienced DBA to understand, analyze, format, and optimize the SQL statements that run on a DB2 subsystem. In particular, you can view a graphical depiction of the access paths and graphically create hints.

10.3 Guidelines for a rebind strategy

It is important to develop a reasonable rebind strategy in your shop. Let us consider the advantages and disadvantages of rebinding. Rebinding allows you take advantage of improvements in the optimizer. This is most likely to occur when a new version of DB2 is installed but minor changes which can affect your access paths can also result from regular service stream maintenance. In rare cases, you have the potential to cause a performance regression. It also allows you to benefit from the most current RUNSTATS information to choose the optimal access path for the current state of the data.

Important: If you think you can keep access paths “frozen,” consider the fact that routine business changes require you to install a changed program, and this has the potential to cause a performance regression as well.

The key issue is not *“whether to rebind”* but *“when to rebind”* !

We suggest the following guidelines for your rebind strategy:

- Rebind each program at least once after a major upgrade. This includes major application changes as well as changes to system software and DB2 maintenance.
- Do NOT issue a mass rebind of all your programs unless:
  - You have simulated (see 10.4, “Change control for access paths” on page 203) and analyzed the potential impact
  - You h the ave a change control procedure in place for access paths (see 10.5.1, “Using a try-out collection in a production environment” on page 204)
  - You are using package stability function on all your packages, in which case you might be able to afford a more aggressive rebind strategy, because fallback can be managed with minimal disruption of service.

- Rebind gradually (for example, per application group or by a group of small number of packages) to limit the impact and scope of analysis needed.
- Use the techniques mentioned in this chapter to avoid the performance regression for specific queries.
- Use the *package stability* function (10.13, “Package stability” on page 216) to switch back quickly to an earlier access path.

10.4 Change control for access paths

In the DB2 development environment, various types of changes including system software upgrades, changes to database structures, and all application software changes are typically tightly controlled. However, the one prominent exception to this rigorous change control is the impact of production binds and rebinds. This can result in unpredictable performance changes. Any inefficient access paths are typically dealt with in a reactive mode.

This is undesirable and a more proactive approach is clearly needed.

Implementing strict change control procedures is not easy. Manually comparing each change to the access path for a large production environment is not practical but home-grown automation or vendor tools can make this task easier.

In Appendix A, “Useful queries” on page 259, we provide a query to detect that a change has taken place. You can refine this query to fit your needs to focus on what is important in your environment.

For example, you might have a list of critical programs or critical tables which affect your service level agreements. Some of the things you could look for are:

- Change in table order
- Change in indexes used
- Change in list prefetch
- Change in join method

10.5 Simulating possible access path changes

Various techniques enable you to test or try out a new package without changing the package that is currently used. These techniques can be used whether an updated program is being moved into a new environment, or an unchanged program is being tested to see if the access paths changes as a result of a rebind.

Some of the scenarios where they might be useful are:

- To determine the impact of SQL changes
- To evaluate the impact of a new index
- To evaluate the impact of updated statistics in the DB2 Catalog
- After maintenance has been installed on DB2 that might affect optimization

You can use the try-out collection in the production environment (as discussed in 10.5.1, “Using a try-out collection in a production environment” on page 204) or in a new environment with no data (as discussed in 10.5.2, “Using a try-out collection in a DEFINE(NO) environment” on page 204). In both cases, the original package is untouched and remains accessible and you do not intend to execute the package, you simply want to simulate the access paths.

Note the following considerations:

- There is no impact on the package in a production environment.
- Examination of performance issues can be completed in advance of going live with new access paths.

10.5.1 Using a try-out collection in a production environment

This technique can be used, for example, to evaluate the potential impact of a rebind. The existing package is bound into a try-out collection using the COPY option of BIND PACKAGE command. This new package can then be used to evaluate the impact of a rebind. By using the access path change control procedures outlined in 10.4, “Change control for access paths” on page 203, we can obtain an early warning and take appropriate steps.

Assume that a new index has been created that could improve the access path of the ACCNTINQ package. To test out the impact of this index, the FIN_PROD_CICS.ACCNTINQ package is bound into the FIN_PROD_TRYOUT collection (using its DBRM as input for the BIND PACKAGE with COPY option). Re-optimization occurs during bind time and might include a new access path that uses the new index. By examining the access path, you can determine if it is likely to provide acceptable performance in the production environment.

10.5.2 Using a try-out collection in a DEFINE(NO) environment

Using this technique, you first create a complete production-like environment with one important difference: You create the table spaces and indexes with DEFINE(NO) option. This creates the structures in the catalog but does not define any datasets. You can then update the relevant catalog statistics to reflect the production (or simulated) environment and bind the package in this new environment to a try-out collection as before. From this you can determine if the access paths are as expected and take appropriate steps.

10.6 VERSION(AUTO) fallback

When using versioning using the VERSION(AUTO) option, you have an easy built-in method for fallback with minimal extra work. Because the package search uses the CONTOKEN from the load module to locate the package, you can retain several versions of the package in your production environment, with only one currently accessible based on the version of the load module being used. If fallback to a previous version is necessary, you simply copy the load module corresponding to the previous version of the package in the LOADLIB and the previous version of the package is automatically used.

No change to the DB2 environment, including BIND, REBIND or FREE of packages is necessary.

The only drawback to this method is that you need to clean up obsolete versions of the package periodically. We discuss the process to do this in 8.2, “Package proliferation” on page 162.

10.7 Using a try-out collection when promoting with a blank VERSION value

In this section we expand on the idea of a try-out collection and incorporate it as part of our normal migration process into the production environment.

W assume that you are not using VERSION(AUTO) when binding packages. If you do, see 10.6, “VERSION(AUTO) fallback” on page 204 for the recommended fallback method.

“Promotion to a production environment using a blank VERSION value” on page 86 provided a step-by-step process. We incorporate the try-out collection scheme outlined earlier in 10.5.1, “Using a try-out collection in a production environment” on page 204 into this process.

The basic idea is to create a “fallback” collection for each collection in your target environment. This collection is generally placed after the real production collection in the PKLIST of your plans.

For example, instead of a production plan with a PKLIST of:

```
PKLIST(FIN_PROD_CICS.*, PAY_PROD_CICS.*, and so on)
```

You would use:

```
PKLIST(FIN_PROD_CICS.*, FIN_PROD_CICS_FALLBACK,
       PAY_PROD_CICS.*, PAY_PROD_CICS_FALLABACK, ...)
```

This mirrors the concatenation of LOADLIBs, which is PROD.LOAD followed by FALLBACK.LOAD. Typically, the FALLBACK collections do not contain packages and the FALLBACK.LOAD library is also empty.

Assuming that you are migrating from System Test to Production, the migration process consists of these steps (steps 3 and 4 are the additional steps that incorporate the try-out collection):

1. Use the COPY option of BIND PACKAGE to create a new package in the FALLBACK.* collection from the PROD.* collection. (At this time, the current load modules are copied to a fallback library).
2. You expect the access paths to remain the same as PROD, but due to changes in catalog statistics or other environmental factors, the access paths might be different. You should evaluate these access paths and deal with any regression. Because you have not affected the PROD collection, this analysis is not time-critical.
3. Use the COPY option of BIND PACKAGE to create a new package from System Test to a Try-out collection in the production environment.
4. You expect the access paths to remain the same as System Test, but because the catalog statistics and other environmental factors are different, the access paths might be different. You should evaluate these access paths and deal with any regression. Because you have not affected the PROD collection, this analysis is not time-critical.

   Note that this is also the “sign-off” where you approve the migration to the PROD platform.
5. Use the COPY option of BIND PACKAGE to create a new package from System Test to the Prod collection. Because you completed the due diligence outlined in steps 3 and 4, you have some assurance that the access paths are good, but due to the timing, you need to verify the access paths are likely to provide acceptable performance (again).
6. Migrate the load module from System Test to Prod. The new package is automatically picked up using the new CONTOKEN.

At this stage you have completed the migration process. If the performance in the production environment is good, you can free the packages in the FALLBACK.* collection and delete the corresponding load module from FALLBACK.LOAD at some convenient point.

If you discover performance issues, all you need to do is delete the load module form the PROD.LOAD library. As a result, the module in the FALLBACK.LOAD library is the one that is used and the package in the FALLBACK collection is the one executed.

To summarize: this procedure allows you to:

- Use try-out collections to verify access paths
- Provide an easy means to fallback even when not using VERSION(AUTO)

10.8 REOPT (NONE / ALWAYS / ONCE / AUTO)

Data is not always evenly distributed and an access path that suits some data values might not suit another. For example, an index on the city name might be useful when the number of qualifying rows is small (such as for Adak, Alaska) but might not be useful when the number of qualifying rows is large (such as for Los Angeles, California). Static SQL with host variables is not aware of this non uniform distribution of data.

As a result, an access path choice made at bind time (based on host variables, not actual data values) might need to be reconsidered based on the data provided when the SQL statement is actually executed.

For dynamic SQL, a similar problem exists. If an SQL statement is prepared and executed for every single set of data values, the cost of optimization is high. If an SQL statement is prepared using parameter markers, the access path is frozen and used for a second query by the same authorization ID regardless of whether or not the data values are appropriate for the chosen index.

The REOPT option on the BIND PACKAGE command addresses this situation. It cannot resolve the issue completely because there is no perfect solution. You can use this option to:

- Freeze the access path when the SQL statement is bound or prepared and:
  - Optimizations costs are minimized.
  - Runtime data values can sometimes cause poor performance.
- Reoptimize the access path every time the SQL statement is executed and:
  - Runtime data values always assist with selection of the most appropriate access path.
  - Optimizations costs can be too high for a frequently-executed statement.

DB2 provides a range of REOPT options that you can choose based on circumstances. Note that a package can contain a combination of static and dynamic SQL.

10.8.1 REOPT options for static SQL

Only two options are available for static SQL:

- REOPT(NONE)

  DB2 does not determine an access path at runtime. It uses the access path selected during the bind process. This is the default and the right option for most static SQL applications.

- REOPT(ALWAYS)

  DB2 determines the access path again at runtime each time the SQL statement is run. DB2 determines access paths at both bind time and runtime for statements that contain one or more of the following variables:

  – Host variables
  – Special registers

  This option can be useful for queries accessing data with non-uniform distribution as well as for queries which are generic searches: for example, an application which allows a user to enter any or all of the search criteria like customer number, account number, ship date, amount, destination city, source city, and so on. In the case of generic searches, REOPT(ALWAYS) can detect which search criteria were entered by the user and create an access path best suited for those criteria instead of a common access path generated at bind time.

  If you have a program with many SQL statements, the REOPT(ALWAYS) applies to all SQL statements in the program because this option is not available at the package level. To minimize the overhead of re-preparing all the other statements which do not require re-optimization, you should consider isolating the statements needing re-optimization into a separate subroutine which can then be bound as a separate package.

10.8.2 REOPT options for dynamic SQL

Multiple options are provided for dynamic SQL:

- REOPT(NONE)

  DB2 does not determine an access path again after the statement has been prepared.

- REOPT(ALWAYS)

  DB2 determines the access path again at runtime each time the SQL statement is run. DB2 determines access paths at both bind time and runtime for statements that contain one or more of the following variables:

  – Parameter markers
  – Special registers

The earlier discussion on when this option is appropriate for static SQL also applies here.

- REOPT(ONCE)

  Determines the access path for any dynamic statement only once, at the first runtime or at the first time the statement is opened. This access path is used until the prepared statement is invalidated or removed from the dynamic statement cache and needs to be prepared again.

- REOPT(AUTO)

  Autonomically determines if a new access path needs to be generated to further optimize the performance for each execution. DB2 determines the access path at the first runtime and then at each subsequent execution, DB2 evaluates whether a new access path is needed. If so, DB2 generates a new access path and uses that access path until another new access path is generated. For a cached dynamic statement referencing parameter markers, a new path can be generated at any execution, based on how the parameter marker value changes.

There are restrictions that apply to dynamic SQL and the REOPT options:

- You cannot use REOPT(ALWAYS) with KEEPDYNAMIC(YES)
- You cannot use both REOPT(ONCE) and NODEFER(PREPARE)

- If a dynamic statement in a plan or package that is bound with REOPT(ONCE) runs when dynamic statement caching is turned off, the statement runs as if REOPT(ONCE) is not specified.
- If you specify the bind option REOPT(ALWAYS) or REOPT(ONCE), DB2 sets the bind option DEFER(PREPARE) automatically. However, when you specify REOPT(ONCE), DB2 determines the access path for a statement only once (at the first runtime).
- REOPT(AUTO) only applies to dynamic statements that can be cached. If dynamic statement caching is turned off and DB2 executes a statement that is bound with REOPT(AUTO), no reoptimization occurs.

10.8.3 Dynamic SQL and REOPT(ONCE)

The REOPT(ONCE) bind option tries to combine the benefits of REOPT(ALWAYS) and dynamic statement caching. The idea of REOPT(ONCE) is to re-optimize the access path only once (using the first set of input variable values) no matter how many times the same statement is executed. The access path chosen is based on the set of input variable values is stored in the dynamic statement cache and used for all later executions until the prepared statement is invalidated or removed from the dynamic statement cache and needs to be prepared again. This solution is based on the assumption that the chosen set of host variable values at runtime is better than the default one chosen by the optimizer at bind time.

REOPT(ONCE) can be useful in some specific cases and REOPT(AUTO) can provide a more general solution. Here is a specific example of where it could be useful:

- A program is used to track shipments for a given user and accesses a large table using the customer name, date, originating city, destination city and some other search criteria. The program issues dynamic SQL. This program is used by two types of users:
  - Retail users who have only a handful of shipments in the system (customer type = RETAIL)
  - Commercial users who have a very large number of shipments in the system (customer type = COMMERCIAL)

  We assume the authorization ID used by each of these types of customers is different.

  Assuming an index exists on customer name in addition to other indexes on other search columns, an access path using this index is generated the first time a query from a retail customer is run. The access path is ideal and is re-used for all queries from subsequent retail customers while commercial customers might use a different access path.

  Using REOPT(NONE) would re-use the same access path for both types of users leading to long runtimes for commercial users while using REOPT(ALWAYS) would consume system resources. REOPT(ONCE) prepares the access path once for each type of user.

10.8.4 Dynamic SQL and REOPT(AUTO)

For frequently called SQL statements that take very little time to execute, re-optimization using different input host variable values at each execution time is expensive, and it might affect the overall performance of applications.

Consider using the REOPT(AUTO) bind option to achieve a better balance between the costs of re-optimization and the costs of processing a statement. You might use the REOPT(AUTO) bind options for many statements for which you could choose either the REOPT(ALWAYS) or REOPT(NONE) bind options, and especially in the following situations:

- The statement must be a dynamic SQL statement that can be cached.

- Queries are such that a comparison can alternate between a wide range or a narrow range of unknown values.
- The SQL statement sometimes takes a relatively long time to execute, depending on the values of referenced parameter markers, especially when parameter markers refer to non-uniform data that is joined to other tables.

In the foregoing cases, the performance gain from a new access path that is chosen based on the input variable values for each might or might not be greater than the performance cost of re-optimization when the statement runs.

When dynamic SQL queries with predicates that contain parameter markers are automatically reoptimized, the newly-generated access path replaces the current one and can be cached in the statement cache.

REOPT(AUTO) can improve execution costs in a query such as:

```
SELECT ... FROM T1 WHERE C1 IN (?, ?, ?) ...
```

Here T1 is a large table that uses different host variables for successive executions with host variable values that provide good filtering. If you specify REOPT(AUTO), DB2 automatically determines if a new access path is required to further optimize the performance of a statement for each execution.

REOPT(AUTO) decides at OPEN (DEFER PREPARE), depending on the content of host variables, whether to re-prepare a statement. The first optimization is the same as REOPT(ONCE). Based on host variable contents, REOPT(AUTO) can:

- Re-prepare a statement and insert into the global cache (full PREPARE)
- Remain in AUTO mode, checking at each OPEN (short PREPARE),
- Change to OPTIMUM mode, with no more checking at OPEN

Emphasis should be placed on the fact that re-optimization checking, due to REOPT(AUTO) results in extra overhead at statement execution regardless of whether the statement is re-optimized. You should consider this cost against the potential total cost of the statement execution. If the statement execution cost is relatively high, then REOPT(AUTO) might be a good candidate. For statements for which REOPT(AUTO) might result in frequent re-optimization, note that the current implementation has a 20% limitation of re-optimization of the total executions on an individual statement. This means that, if every statement execution would result in re-optimization because of the different disparate host variable values, re-optimization does not necessarily occur due to this limit. For this kind of processing, use REOPT(ALWAYS) to avoid this limitation.

Refer to *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851, for details.

Note: The REOPT(AUTO) functionality is available in DB2 9 New Function Mode (NFM). APARs PK49348 (OPEN) and PK47318 (UK31630) provide additional improvements to REOPT(AUTO).

10.9 Optimization hints

Experienced users can tell DB2 how to process a query by giving hints to DB2. The process of giving hints to DB2 is relatively simple but determining what those hints should be is not. Always test and analyze the results of any query that uses optimization hints. Giving optimization hints to DB2 is useful in the following situations for some specific queries:

- You want consistency of response times across rebinds and across code migrations. When a plan or package is rebound, the access path is reformulated. If the database or application has changed, or if DB2 has new function that causes it to choose a different access path, it is handy to have the ability to suggest an old access path if the new one does not perform as well. For this reason, it is even more important to save the access paths of your critical queries before migrating to a new version of DB2.
- You want to temporarily bypass the access path chosen by DB2.

Figure 10-2 illustrates how a hint is created and applied for static SQL.

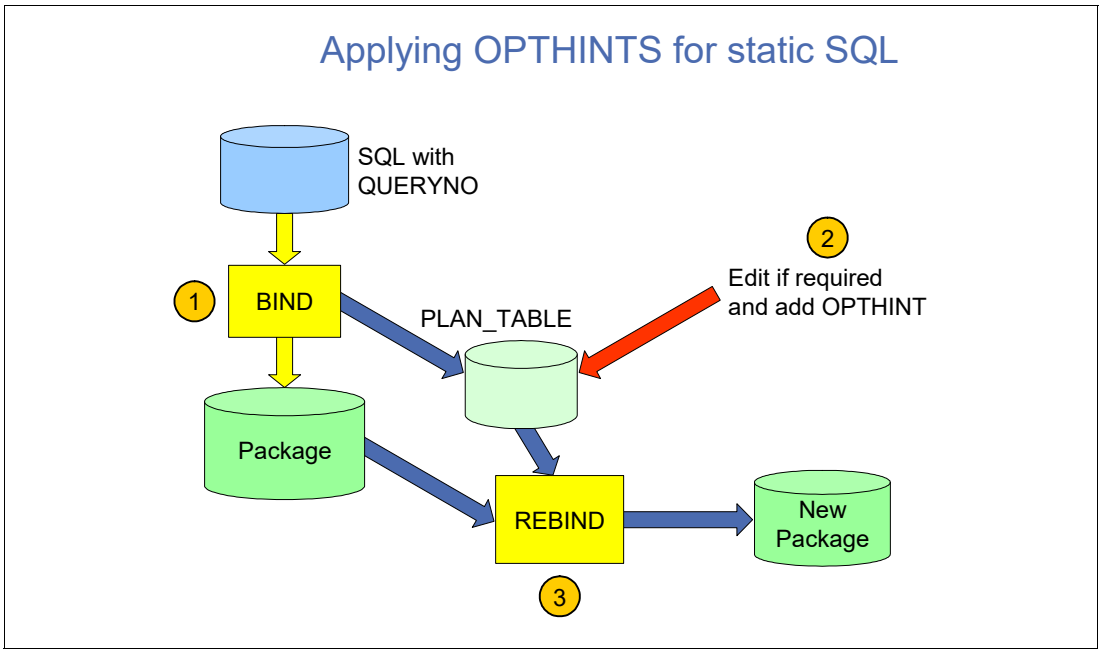

Figure 10-2 How optimization hints are applied

10.9.1 Enabling optimization hints for the system

On the subsystem where the application is bound or where dynamic queries are prepared, specify YES for the DSNZPARM OPTHINTS. If this is not set, all optimization hints are ignored by DB2. After being authorized, there is no means to limit the use to an authorized group of individuals; any user authorized to bind a package can apply a hint.

10.9.2 Planning to use optimization hints

Before you can give hints to DB2, make sure your PLAN_TABLE is of the correct format. Refer to *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851, for details on the current format of the PLAN_TABLE and the CREATE statement you need to issue.

For best performance, create an ascending index on the following columns of PLAN_TABLE:

- QUERYNO
- APPLNAME
- PROGNAME
- VERSION
- COLLID
- OPTHINT

The DB2 sample library, in member DSNTESC, contains an appropriate CREATE INDEX statement that you can modify.

You should also consider building an infrastructure that manages your hint environment and possible actions when hints are expected but not found or not honored by DB2.

10.9.3 Why QUERYNO is critical

DB2 does not require you to assign a query number to use optimization hints. If you do not assign a query number, DB2 uses the statement number (indicated by STMTNOI or STMTNO in SYSPACKAGE). However, query numbers are critical in the long run, especially for static SQL.

Dynamic SQL

The query number for dynamic applications is the statement number in the application where the prepare occurs. For some applications, such as DSNTEP2, the same statement in the application prepares each dynamic statement, resulting in the same query number for each dynamic statement. Assigning a query number to each statement that uses optimization hints eliminates ambiguity as to which rows in the PLAN_TABLE are associated with each query.

Static SQL

If you change a application source code or copybooks that have static statements, the statement number might change, causing rows in the PLAN_TABLE to be out of sync with the modified application. Statements that use the QUERYNO clause are not dependent on the automatically generated statement number. You can move those statements around in the source code without affecting the relationship between rows in the PLAN_TABLE and the statements that use those rows in the application.

Important: In an application containing embedded static SQL (such as COBOL), *any* program change (for example, addition of a few comment lines at the top) is likely to affect the statement number. We strongly recommend that QUERYNO is mandatory for queries using hints.

10.9.4 Case 1: Preventing a change during REBIND PACKAGE

By making an access path into a hint, you can tell DB2 to use that same access path when you rebind.

The following scenario assumes that DB2 uses an access path that you like for a particular query and that PLAN_TABLE contains a row for that desired access path. By making that access path into a hint, DB2 can use your hint when you rebind, essentially “freezing the access path”:

1. Determine the query number that currently exists for that query in the PLAN_TABLE. In order to correlate the query number with the query in the application, modify the statement in the application to use the QUERYNO clause. (If you want to use some kind of numbering convention for queries that use access path hints, you can change the query number in PLAN_TABLE. The important thing is to have the query in the application use a query number that is unique for that application and that matches the QUERYNO value in the PLAN_TABLE.) We assume you have used QUERYNO=100 in this example.
2. Make the PLAN_TABLE rows for that query (QUERYNO=100) into a hint by updating the OPTHINT column with the name you want to call the hint. In this case, the name is GOOD_OLD_PATH.

3. Tell DB2 to use the hint, and verify in the PLAN_TABLE that DB2 used the hint.
   a. For dynamic SQL statements in the program, follow these steps:

      Execute the SET CURRENT OPTIMIZATION HINT statement in the program to tell DB2 to use GOOD_OLD_PATH. For example: SET CURRENT OPTIMIZATION HINT = 'GOOD_OLD_PATH'. If you do not explicitly set the CURRENT OPTIMIZATION HINT special register, the value that you specify for the bind option OPTHINT is used. If you execute the SET CURRENT OPTIMIZATION HINT statement statically, rebind the plan or package to pick up the SET CURRENT OPTIMIZATION HINT statement.

      Execute the EXPLAIN statement on the SQL statements for which you have instructed DB2 to use GOOD_OLD_PATH. This step adds rows to the PLAN_TABLE for those statements. The rows contain a value of GOOD_OLD_PATH in the HINT_USED column. If DB2 uses all of the hints that you provided, it returns SQLCODE +394 from the PREPARE of the EXPLAIN statement and from the PREPARE of SQL statements that use the hints. If any of your hints are invalid or not used, DB2 issues SQLCODE +395. If the statement executes successfully, DB2 returns SQLCODE 0.

      Important: If the dynamic statement cache is enabled, DB2 does not use the hint to optimize a dynamic SQL statement.

   b. For static SQL statements in the program, follow these steps:

      Rebind the plan or package that contains the statements. Specify bind options EXPLAIN(YES) and OPTHINT('GOOD_OLD_PATH') to add rows for those statements in the PLAN_TABLE that contain a value of GOOD_OLD_PATH in the HINT_USED column. If DB2 uses the hint you provided, it returns SQLCODE +394 from the rebind. If your hints are invalid or not used, DB2 issues SQLCODE +395.
4. Select from PLAN_TABLE to see what was used for the last rebind. It should show the GOOD_OLD_PATH hint, indicated by a value in OPTHINT and it should also show that DB2 used that hint, indicated by GOOD_OLD_PATH in the HINT_USED column.

10.9.5 Case 2: Modifying an existing access path

The following scenario assumes that DB2 uses an undesirable access path (OLDPATH) and you know of a better access path (TRY_NEW_PATH):

1. Put the old access path in the PLAN_TABLE and associate it with a query number.
2. Make the PLAN_TABLE rows into a hint by updating the OPTHINT column with the name you want to call the hint. In this case, the name is TRY_NEW_PATH.
3. Change the access path so that PLAN_TABLE reflects the rows associated with the desirable new access path TRY_NEW_PATH.
4. Tell DB2 to look for the TRY_NEW_PATH hint for this query:
   - If dynamic, issue: SET CURRENT OPTIMIZATION HINT = 'TRY_NEW_PATH';
   - If static, rebind the package or plan with OPTHINT set to TRY_NEW_PATH.
5. Use EXPLAIN on the query, package, or plan to check the results.
6. Examine the PLAN_TABLE to see what was used for the last rebind. It should show the TRY_NEW_PATH as the value in OPTHINT and it should also show that DB2 used that hint, indicated by the value TRY_NEW_PATH in the HINT_USED column of the PLAN_TABLE.

10.9.6 How DB2 locates the PLAN_TABLE rows for a hint

For a PLAN_TABLE row, the QUERYNO, APPLNAME, PROGNAME, VERSION, and COLLID values must match the corresponding values for an SQL statement before the SQL statement can use that row. In addition:

- If the SQL statement is executed dynamically, the OPTHINT value for that row must match the value in the CURRENT OPTIMIZATION HINT special register.
- If the SQL statement is executed statically, the OPTHINT value for the row must match the value of bind option OPTHINT for the package or plan that contains the SQL statement.

If no PLAN_TABLE rows meet these conditions, DB2 determines the access path for the SQL statement without using hints.

10.9.7 How DB2 validates the hint

DB2 provides a valid access path. DB2 validates the PLAN_TABLE rows in which the value in the OPTHINT column matches the specified hint. If the access path that you specify is clearly inferior, DB2 invalidates all hints for that query block. In that event, DB2 determines the access path as it normally does. DB2 validates the information only in the following PLAN_TABLE columns:

- METHOD
- CREATOR and TNAME
- TABNO
- ACCESSTYPE
- ACCESSCREATOR and ACCESSNAME
- SORTN_JOIN and SORTC_JOIN
- PREFETCH
- PAGE_RANGE
- PARALLELISM_MODE
- ACCESS_DEGREE and JOIN_DEGREE
- WHEN_OPTIMIZE
- PRIMARY_ACCESSTYPE

10.9.8 Limitations on optimization hints

Optimization hints cannot force or undo query transformations, such as subquery transformation to join or materialization or merge of a view or table expression.

Important: If you include a hint in one release of DB2 for a query that is not transformed in that release, but in a later release of DB2, the query is transformed, DB2 does not use the hint in the later release.

10.9.9 Best practices when using optimization hints

We recommend that you follow these guidelines to manage your optimization hints:

- Use optimization hints only when necessary.
- Develop an infrastructure (or use vendor tools) to mange your hints. Specifically, make sure you have procedures in place to apply the desired hint consistently and alert you if the suggested hint was not applied by DB2.

- Make sure you retain all PLAN_TABLE rows that relate to hints. See A.14, “Showing packages that use OPTHINTs” on page 269, and A.16, “Finding obsolete PLAN_TABLE entries” on page 270 for an example.
- Periodically re-evaluate whether the hint is still needed and remove if they are unwarranted. See 10.4, “Change control for access paths” on page 203 for methods to simulate access paths.

10.10 SQL tricks

If DB2 chooses an unsatisfactory access path, you might consider using some “special tricks” to rewrite your SQL statement to get a different access path.

Important: The access path selection “tricks” that are described here cause significant performance degradation if they are not carefully implemented and monitored. In case of a query re-write, the “trick” might become ineffective in a future release of DB2.

Some of the examples of such methods of influencing the access path are as follows:

- Add OPTIMIZE FOR n ROWS where n is artificially small or large
- Define a table as VOLATILE to encourage index usage
- Use CARDINALITY clause of a user-defined function
- Use the CARDINALITY MULTIPLIER clause of a user-defined function
- Reduce the number of matching columns for an index by making predicates non-indexable (for example, col = :hv + 0)
- Add fake redundant predicates to favor one access path over another

Refer to *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851, for details.

10.11 Materialized Query Tables (MQT)

A materialized query table (MQT) contains pre-computed data. The pre-computed data is the result of a query, that is a fullselect associated with the table, specified as part of the CREATE/ALTER TABLE statement. The source for a materialized query table can be base tables, views, table expressions, or user-defined table functions.

If you use MQTs, the costly computations do not have to be performed every single time you run a query against your warehouse tables. The query to build the MQT is executed only once. From then on, the data is available in the MQT, and can be used instead of the underlying tables.

MQTs can either be accessed directly via SQL, or can be chosen by the optimizer (through automatic query rewrite). In this case DB2 optimizer determines whether or not to use the MQT, and is the real power of the feature, as users no longer have to be aware of the existence of the MQT.

Automatic QUERY REWRITE

The process of recognizing whether an MQT can be used in answering a query and rewriting the query accordingly, is called automatic query rewrite. Two new special registers, CURRENT REFRESH AGE and CURRENT MAINTAINED TABLETYPES FOR OPTIMIZATION, control whether or not an MQT is considered by automatic query rewrite for a dynamically prepared query. The default value for both parameters is specified on panel DSNTIP8 during installation time. The default value for CURRENT REFRESH AGE ends up as DSNZPARM parameter REFSHAGE, and the one for CURRENTMAINTAINED TABLE TYPES FOR OPTIMIZATION goes into DSNZPARM parameter MAINTYPE.

Important: Automatic query rewrite is only supported for dynamically prepared queries that are read-only. It is not supported for statically bound queries, including REOPT(VARS) packages.

You can, however, use an MQT in either a statically bound query, or a dynamically prepared query to improve the response time, if appropriate, by coding them directly in the SQL statement (thus “bypassing” automatic query rewrite done by the optimizer).

To determine whether an MQT is used instead of a base table, you can use the EXPLAIN statement. The EXPLAIN output shows the plan for the rewritten query. When used, the name of the MQT is shown in the TNAME column, and a TABLE_TYPE of 'M' is used.

10.12 Catalog statistics manipulation

If you have the proper authority, you can influence access path selection by using an SQL UPDATE or INSERT statement to change statistical values in the DB2 catalog. However, doing so is not generally recommended except as a last resort.

Important: If you update the catalog statistics, make sure that you update all related statistics. For example, if the cardinality of a table is increased, the statistics for the associated indexes almost certainly need to change also.

Although updating catalog statistics can help a your specific query, other queries can be affected adversely. Also, the UPDATE statements must be repeated after RUNSTATS resets the catalog values. You should be very careful if you attempt to update statistics. If you update catalog statistics for a table space or index manually, and you are using dynamic statement caching, you need to invalidate statements in the cache that involve those table spaces or indexes. To invalidate statements in the dynamic statement cache without updating catalog statistics or generating reports, you can run the RUNSTATS utility with the REPORT NO and UPDATE NONE options on the table space or the index that the query is dependent on.

In some cases, it is useful to copy statistics for an entire database from production to development to preview access path behavior. This could be done with a vendor or home grown tool.

10.13 Package stability

This function has been made available through the maintenance stream after general availability of DB2 9 for z/OS via APAR PK52523 (PTF UK31993). The support applies to packages—not plans—and includes non-native SQL procedures, external procedures, and trigger packages. It helps in those situations where a static SQL REBIND causes worse performance of queries because of changes to access paths. Regression of access paths typically occurs when:

- Migrating to a new release of DB2
- Applications are bound or rebound after maintenance is applied to DB2
- Changes in data distribution are reflected in the DB2 Catalog by updated statistics values
- There are schema changes to the application database objects

Important: Some IBM manuals refer to this feature as “Plan stability” because it deals with the stability of access paths (an “access plan”). In this book, we refer to it as package stability because this option is available for packages only.

For transactions known to be critical, users have had techniques like access path hints (implemented via the OPTHINT option of BIND and REBIND) to revert to old access paths. However access path hints are not simple to create or maintain, and identifying the source of such regressions can be time-consuming. Package stability provides a fast and relatively inexpensive way to revert to a previous “good” access path. This improves availability and allows time for more complete problem determination

During REBIND PACKAGE with active PLANMGMT, the various copies of the package are managed in the DB2 directory. Extra rows in SYSPACKDEP, denoted by different values for DTYPE, indicate the availability of additional package copies.

Important: DB2 stores all of the details about a package in the Directory, but only the *active* copy is externalized in the Catalog. If an index or other database object is dropped that causes VALID=’N’ on the original or previous copy of the package, this is not visible in the Catalog until a REBIND with the SWITCH option activates that copy of the package.

Controlling the new PLANMGMT option for REBIND PACKAGE

The option can be controlled at two levels:

- Subsystem level via a new DSNZPARM (PLANMGMT)
- BIND level with new options for REBIND

Controlling globally at subsystem level

DB2 9 introduces a new system parameter called PLANMGMT for specifying the default setting of PLANMGMT option of REBIND PACKAGE.

Possible settings are: OFF, BASIC, and EXTENDED. The default value of this parameter is OFF. To use package stability, set the value to BASIC or EXTENDED.

Controlling at package level during REBIND

The new REBIND PACKAGE option is called PLANMGMT.

You can affect packages selectively with three possible settings: OFF, BASIC, and EXTENDED.

- PLANMGMT(OFF):

  This is the default behavior. There is no change to the existing behavior. A package continues to have one active copy.

- PLANMGMT(BASIC):

  The package has one active (“current”) copy, and one additional (“previous”) copy is preserved.

  At each REBIND:

  - Any previous copy is discarded.
  - The current copy becomes the previous copy.
  - The incoming copy becomes the current copy.

Important: As a result of this, if you issue two or more rebinds, after migration to a new version, you can end up wiping out the access path for packages from previous version which you might wish to preserve.

- PLANMGMT(EXTENDED):

  It retains up to three copies of a package: one active copy and two additional old copies (PREVIOUS and ORIGINAL) are preserved.

  At each REBIND:

  - Any previous copy is discarded
  - If there is no original copy, the current copy is saved as the original copy
  - The current copy becomes the previous copy
  - The incoming copy becomes the current copy

Important: Unlike the case when you use PLANMGMT(BASIC), the original copy is the one that existed from the “beginning”, *it is saved once and never overwritten* (it could be the copy you wish to preserve from a prior version).

Using the new SWITCH option for REBIND PACKAGE

When regression occurs, a new REBIND PACKAGE option called SWITCH allows you to switch back to using the old package copy. This switch, unlike a normal REBIND operation that generates a new access path, preserves the earlier access path.

- SWITCH(PREVIOUS):

  Provides a means of falling back to the last used copy:

  - The previous copy is activated.
  - It switches from the current to the previous copy.

- SWITCH(ORIGINAL):

  Provides a means of falling back to the oldest known package:

  - The original copy is activated.
  - The current copy moves to previous, and the original copy becomes current.

Figure 10-3 shows how REBIND works when using PLANMGMT=BASIC.

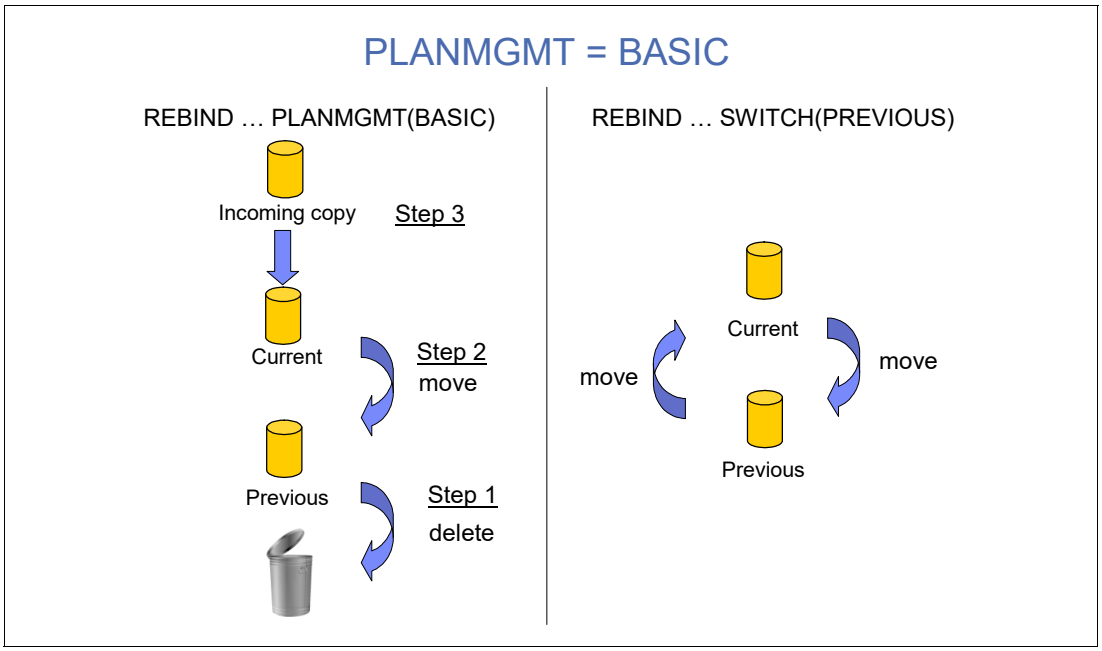

Figure 10-3 PLANMGMT=BASIC

Figure 10-4 shows how REBIND works when using PLANMGMT=EXTENDED. After the first REBIND, the old current package is copied to previous and original.

REBIND... SWITCH(PREVIOUS) works the same way as with PLANMGMT=BASIC and now you have the additional option of REBIND... SWITCH(ORIGINAL).

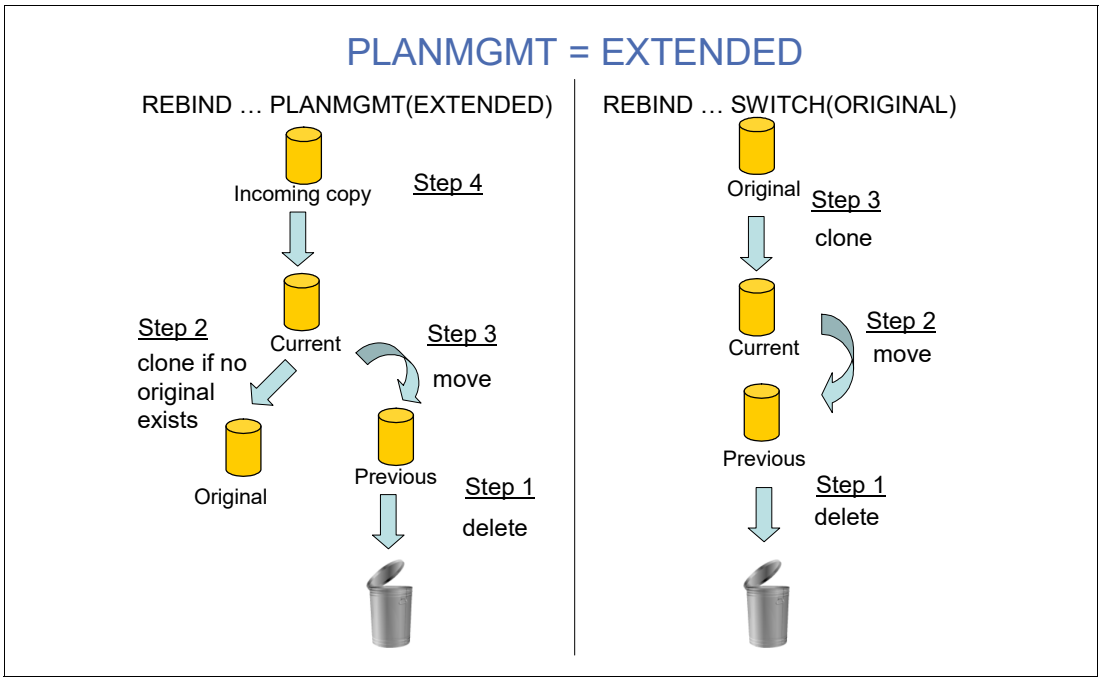

Figure 10-4 PLANMGMT=EXTENDED

Detecting number of copies saved for a package

You can run the query shown in Example 10-1 to see how many copies of each package you have.

Example 10-1 Number of saved copies by version for package stability

```
SELECT SP.COLLID, SP.NAME, SP.VERSION, SP.CONTOKEN
      COUNT(DISTINCT SPD.DTYPE) AS COPY_COUNT
FROM SYSIBM.SYSPACKAGE SP, SYSIBM.SYSPACKDEP SPD
WHERE SP.COLLID = SPD.DCOLLID
AND SP.NAME = SPD.DNAME
AND SP.CONTOKEN = SPD.DCONTOKEN
AND DTYPE IN (‘’, ‘O’, ‘P’)
GROUP BY SP.COLLID, SP.NAME, SP.VERSION, SP.CONTOKEN
```

Based on the value of copycount, you can determine the option you have deployed:

- COPY_COUNT=1: OFF (DTYPE = blank)
- COPY_COUNT=2: BASIC (DTYPE = blank and P)
- COPY_COUNT=3: EXTENDED (DTYPE = blank, P, and O)

Deleting old PACKAGE copies

A new FREE PACKAGE option called PLANMGMT SCOPE allows you to free older copies that are no longer necessary.

- PLANMGMT SCOPE(ALL):

  To free the entire package including all copies. This is the default.

- PLANMGMT SCOPE(INACTIVE):

  To free all old copies only (that is, original and previous, if any).

The existing FREE PACKAGE command and DROP TRIGGER SQL statement drops the specified package and trigger as well as all associated current, previous and original copies. In other words, it behaves like SCOPE(ALL).

For details, refer to *DB2 Version 9.1 for z/OS Installation Guide*, GC18-9846-02 and other current DB2 product documentation.

Visibility of original and previous copies

Keep in mind that Package Stability retains a copy of not just the access paths but the entire package, however the catalog visibility for the original and previous copies is limited. The package details are stored in SPT01 for all copies of the package but the externalization of this in SYSPACKAGE occurs only for the active copy.

The implications of this are as follows:

1. SYSPACKAGE shows only the current copy, but the other copies, along with their bind parameters, are retained by DB2. For example, the current copy could be using isolation level CS and the previous copy using isolation level UR. After a REBIND PACKAGE... SWITCH, the UR copy of the package becomes active.
2. SYSPACKDEP shows all copies of the package. If an index used by a non-current copy is dropped, that copy is marked invalid, although the current entry in SYSPACKAGE might remain valid. If REBIND PACKAGE... SWITCH to the invalid copy is issued, the switch occurs but the current copy might now be invalid. Even if an auto rebind makes the current copy valid, the access path might not be what you originally intended.

Important: We strongly recommend that you develop special procedures for packages using package stability. An invalidation of *any* copy of such packages should be dealt with as if it is the current copy. You should also verify that the package you switched to after a REBIND... SWITCH is valid.

10.14 Planning and managing fallback for access paths

In this chapter, we have discussed various proactive and reactive techniques to manage the fallback to a previous access path of the package in case of severe performance regression. Refer back to the following sections as necessary:

- 10.7, “Using a try-out collection when promoting with a blank VERSION value” on page 205
- 10.6, “VERSION(AUTO) fallback” on page 204
- 10.9, “Optimization hints” on page 209
- 10.13, “Package stability” on page 216

Each of these techniques can be useful to plan and manage your fallback strategy for access paths. In this section, we compare these methods, as summarized in Table 10-1.

Table 10-1 Planning and managing fallback for access paths

Method	Advantages	Disadvantages
Using try-out collections in the promotion process	► Seamless fallback ► Easy to manage	► Normal promotion can causes disruption of service (package cannot be in use) ► Limited to the number of collections defineds
VERSION(AUTO)	► Seamless fallback ► Easy to manage ► Not limited to specific number of versions ► Versions are easily identifiable ► Fallback using appropriate LOADLIB ► Normal promotion causes no disruption of service (previous version can be in use)	► Need to clean up obsolete versions ► Authorization is version independent (for example, user authorized to execute a read-only version could execute update-intent version) with proper LOADLIB ► Need to retain all requited LOADLIB versions
Optimization hints	► No preparation needed up-front ► Might be useful if a very small number of issues or no critical SLAs	► Reactive in nature ► Difficult to respond quickly to access path regressions ► Need to build infrastructure to manage hints ► With query rewrite, hints might be disregarded in the future
Package stability	► Minimal planning needed (bind with PLANMGMT option) ► Easy to manage	► Increases in SPT01 size ► Increases catalog size ► Limited to “original” and “fallback” copies of package ► Non-active copies not very visible in catalog ► Bind options might be different for earlier copies

To summarize:

- VERSION(AUTO) is the most flexible and recommended method.
- Package stability should be considered for critical programs.
- Optimization hints should be used only when necessary.
- Try-out collections in the promotion process should not be the last resort.

11

Performance considerations

In this chapter we discuss topics related to performance of packages. First we introduce the BIND option and package complexity from the performance point of view. Next we consider package lists, then the available monitoring functions. After that, we show how to investigate queries from a package, and finally we deal with an issue relating to large package size.

We cover the following topics:

- Binding parameters that impact performance
- Package monitoring
- Investigating a query from a package
- Considerations for packages with a large number of sections
- Tracing packages

11.1 Binding parameters that impact performance

In this section we discuss some of the important BIND features that impact performance.

11.1.1 CURRENTDATA

CURRENTDATA(NO) serves two purposes:

- It promotes lock avoidance, but only for applications using isolation level cursor stability, ISOLATION(CS).
- It promotes data blocking for distributed applications, and it is the only option available when issuing a remote bind package.

For most cases, CURRENTDATA(NO) in conjunction with ISOLATION(CS) is the appropriate choice for maximum concurrency and performance. For a detailed description of the impact of CURRENTDATA, see *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*, SG24-7134.

11.1.2 RELEASE(COMMIT)

This option tells DB2 whether to release resources at each COMMIT or when the plan is deallocated.

“Resources” are generally considered to be locks, but in fact some transient DB2 structures used to improve performance are also released (destroyed).

RELEASE(COMMIT) releases resources at each COMMIT. It also causes structures to be destroyed. This switches off dynamic prefetch, and eliminates Insert Procs (IPROCs), Update Procs (UPROCs) and Select Procs (SPROCs). If processing continues, DB2 will:

- Reacquire database object locks as they are needed
- Rebuild statistics used by index look-aside and sequential detection
- Rebuild IPROCs, SPROCs, and UPROCs if they are needed

For this reason, issuing COMMIT too frequently causes overhead beyond additional CPU for the COMMIT processing itself. It is recommended that online applications generally use RELEASE(COMMIT) because there is usually only a single COMMIT.

For batch applications that process large data volumes, RELEASE(DEALLOCATE) can be appropriate, although the retained table space locks could prevent a successful online REORG utility from completing.

The things impacted by the RELEASE parameter are:

- When table space level locks are released
- When plans and packages are released from the EDM pool
- When storage acquired by the thread is released
- When sequential detection information is reset
- When index look-aside information is reset
- When IPROCs, SPROCs, and UPROCs are destroyed

There are two values that you can specify for the RELEASE parameter:

- **COMMIT**: Release resources when the program commits
- **DEALLOCATE**: Release resources when the program ends

A comparison of the pros and cons of release deallocate is shown in Table 11-1. When you consider the list of resources that are affected, it should be clear that the use of RELEASE(DEALLOCATE) is certainly not a free lunch.

Best practice is to use RELEASE(COMMIT) and set the commit frequency to run 2 to 15 seconds between commits. The trade-off is between lock wait and efficiency for this process. In this range you can generally satisfy online users and run nearly as fast.

Using RELEASE(DEALLOCATE) tells DB2 that you want to keep the resources around so they do not need to be reacquired after a commit. This is good if you have a long-running batch job with frequent commits, or want to get the best performance out of thread reuse in an online environment with over 10 transactions per second and less than 5 SQL statements per transaction.

Table 11-1 Pros and cons of RELEASE(DEALLOCATE)

Pros	**Cons**
Reduce CPU for simple transaction	Increased virtual storage usage
Reduces table space lock activity	Increased EDM pool usage
Reduces locks propagated to CF in data sharing	Increased chance of contention issues

The advantages of the use of RELEASE(DEALLOCATE) are as follows:

- It reduces CPU impact for simple transactions. Acquiring the locks and plan tables can represent a large portion of the CPU cost of a simple transaction. Saving this can mean a significant performance boost for simple, short running transactions.
- It reduces table space (TS) lock activity.
- It reduces the number of TS locks propagated to the CF in a data sharing environment. The caveat is if this is the only work in the subsystem for the TS. Commit protocol 2 resolves most issues.
- For batch with many commits, RELEASE(DEALLOCATE) avoids resetting sequential detection, index look-aside, and xPROCs at commit, as well as destroying the PT in the EDMPOOL. The best answer is to commit less frequently.

However, the downsides of using RELEASE(DEALLOCATE) should be considered as well, and in many big installations, they often outweigh the benefits. The disadvantages of using RELEASE(DEALLOCATE) are as follows:

- Virtual storage capacity is impacted. The threads accumulate ever more storage for statements that are not being used. The storage for unused statements can be left around until deallocation time. For these threads, storage contraction is not effective (nor is this the case when full system storage contraction needs to be performed). The storage impact of RELEASE(DEALLOCATE) is often an underestimated problem that can have severe consequences. There ia also CPU cost for managing the storage.
- The plans and packages that are bound with RELEASE(DEALLOCATE) stay around much longer and cause growth in the EDM pool's storage consumption. This is especially problematic in a CICS environment with heavy thread reuse when using a single plan.

- The chances of getting into resource contention increase. Because RELEASE(DEALLOCATE) only releases table space locks at deallocation time for static SQL, and dynamic SQL when full caching is active, you might receive time-outs when:
  - (Re)binding plans and packages
  - Executing DDL against any of the objects referenced by any of those plans and packages
  - Trying to perform a mass delete on segmented table space
  - Lock escalation has occurred
  - Executing an SQL LOCK TABLE statement
  - Running an on-line utility

Recommendation: Use RELEASE(COMMIT), unless you really need the small amount of extra performance that RELEASE(DEALLOCATE) can provide, and your system can tolerate the side effects of the use of RELEASE(DEALLOCATE) as discussed previously.

A measurement done by the DB2 performance department in the IBM Silicon Valley Laboratory comparing workload using RELEASE(DEALLOCATE) versus RELEASE(COMMIT) using the DB2 Version 8 data sharing environment showed only a 5% Internal Throughput Rate (ITR) and 7% CPU time difference. The gap used to be much wider in Version 7, 17% for both ITR and CPU time, mainly due to the extra XES contention in the old locking protocol (protocol level 1). This shows that you must be very close to some capacity limit that is difficult to remove to be able to justify the “inconveniences” caused by the use of RELEASE(DEALLOCATE).

It is worth noting that the RELEASE attribute is ignored in a distributed environment where DB2 for z/OS work is done on behalf of a DRDA client. In a distributed environment, RELEASE(COMMIT) is always used. Also note that the RELEASE parameter does not apply to page and row locks. In almost all circumstances, they are released at commit time.

Impact on sequential detection and index look-aside

Some of the mechanisms that DB2 uses to improve performance are dynamic prefetch and index look-aside. Beginning with DB2 9 for z/OS, DB2 highly favors dynamic prefetch over sequential prefetch. DB2 uses an algorithm which looks at the last 8 pages that have been read for a table into the bufferpool. If 5 out of the 8 page reads are sequential, DB2 turns on dynamic prefetch. This process is referred to as sequential detection. DB2 then retrieves a large amount of pages asynchronously. The number of pages is dictated by the size of the bufferpool, but in DB2 9 for z/OS it has been increased to allow for 64 pages to be read asynchronously. The resulting performance on high performance disk subsystems is more than 200 MB/sec for sequential read.

Impact on SPROC/IPROC/UPROCs

DB2 looks at programs as they are running to determine if they are running the same SELECT, INSERT or UPDATE statements. If they are issuing the same type of statement repetitively, DB2 might decide that it is faster to spend some CPU cycles to build a structure which processes these statements more efficiently. These structures are called IPROCS (insert procedures), SPROCS (select procedures) and UPROCS (update procedures). When using RELEASE(COMMIT), each time you commit, these structures are destroyed. In a batch application, if you commit too frequently and use RELEASE(COMMIT), DB2 might spend the CPU cycles to build these procedures and destroy them before they become beneficial in reducing CPU time of the job. You can decrease the number of commits or bind the program with RELEASE(DEALLOCATE) to allow for these procedures to help you lower your CPU usage.

Impact on high volume transaction environments

In a high volume transaction environment (more than 10 tps) and light transactions (under 5 SQL statements), reusing threads can significantly improve performance. In DB2, one of the ways that you reduce resource consumption when reusing threads is by using RELEASE(DEALLOCATE) on your plans and packages. This minimizes the amount of overhead DB2 performs at transaction allocation/deallocation time.See *DB2 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851 and *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*, SG24-7134 for details.

11.1.3 DEGREE

The DEGREE parameter tells DB2 whether it is permissible to use parallelism for this package. If you specify 1, the package does not use parallelism. If you specify ANY, then DB2 might choose to use a parallel access when the package is bound. DEGREE should be specified as 1 until you can show the elapsed time reduction of running parallel. DEGREE(ANY) can require 50% to 70% more storage for the cursor tables and plan tables in the EDM pool than if bound with DEGREE(1).

In a zIIP environment, parallel can reduce cost by shifting workload from general purpose processors to zIIP.

For more details, refer to “Enabling parallel processing” in *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851.

11.2 Package list organization

In order to maximize throughput in a DB2 system, prior DB2 documentation suggested that you limit the of the number of packages and package lists in a plan. The details of package list search are shown in 3.2, “Package list search ” on page 41. There is overhead in searching for packages and not finding them. However, it does not appear that this is as big an issue today as in the past.

11.2.1 Performance results

The following are results from a basic test that we ran. The program executed a simple select statement, SELECT ‘1’ FROM SYSIBM.SYSDUMMY1 followed by a COMMIT. The program ran these 2 statements 5,000 times. We ran the COMMIT to make DB2 destroy and rebuild the package table (PT) in SPT01 when using RELEASE(COMMIT). We ran six different tests. We used plans with 1, 20, and 999 PKLIST entries, each using RELEASE(COMMIT) and RELEASE(DEALLOCATE). In the tests with multiple package lists, the program we executed was the last package in the package list. A sample BIND statement that we used is shown in Example 11-1.

Example 11-1 BIND sample used for our tests

```
BIND PLAN(PLAN999)  +
PKLIST(COLL001.*  , +
       COLL002.*  , +
       COLL003.*  , +
       ... ,
       TEMP.PGM999)
```

In our test, we did not have any packages in the collections that were ahead of the package that we were executing.

The results of this test are shown in Table 11-2.

Table 11-2 Performance data using multiple package lists and 5000 executions

# of collections	Elapsed seconds		Class 1 CPU seconds		Class 2 CPU seconds	
	COMMIT	DEALLOC	COMMIT	DEALLOC	COMMIT	DEALLOC
1	0.159	0.107	0.149	0.099	0.146	0.096
20	0.217	0.106	0.171	0.099	0.168	0.096
999	4.468	0.623	1.323	0.115	1.319	0.115
999 WITH HOLD	0.181		0.163		0.160	

Tip: If you use a large package list, use very few, short SQL statements, your package is at the bottom of the package list, and you use RELEASE(COMMIT), you might incur significant CPU increases. In this case, you can move the package higher in the package list, use RELEASE(DEALLOCATE) and execute more SQL in the package so that the overhead is not as significant, or you can use the alternative that we outline in the following discussion, to open a dummy cursor WITH HOLD.

The results show that the fewer package lists you have in your plan, the better the program performs. They indicate that it makes sense to keep your package lists as small as possible, but the additional CPU is not earth shattering. When creating your package list, keep the most frequently executed packages towards the front of the PKLIST.

However, if you use a package list with a large number of collection.* entries and you commit frequently and your package is at the end of the list, you could incur significant CPU increases. This is due to the entry in the package table (PT) of the EDM pool being rebuilt after each commit. You can use RELEASE (DEALLOCATE) in this situation to reduce the overhead of PT allocation.

If you are connecting using ODBC or JDBC, you use plan DISTSERV. In order to limit the number of collections that your plan use, limit the size of your CURRENT PACKAGE PATH to the necessary collections. When running with a named plan, keep the most frequently used packages at the beginning, or use package lists and keep the most frequently used package lists at the front of the PKLIST specification.

Open a dummy cursor WITH HOLD

If you need to run your program with RELEASE(COMMIT), you can use a technique which opens a dummy cursor at the beginning of a program and keep it open throughout the execution of the program. You can use this technique to improve performance if the following three items are true:

- You run with RELEASE(COMMIT)
- You have a large package list
- Your frequently executed packages are towards the end of the package list

DB2 does not remove the entries from SPT01 if there are any cursors open “with hold” at commit processing. You can code a dummy cursor to select from SYSDUMMY1 WITH HOLD. At the beginning of your program, open the cursor and leave it open for the duration of the program. The results from this test are listed in the last row of Table 11-2.

While the performance is not as good as if you had used a smaller package list, it is significantly better than running the same program without opening the dummy cursor. The elapsed time was reduced from 4.468 seconds to 0.181 seconds.

11.2.2 Impact on the EDM pool

The environmental descriptor manager (EDM) pool contains active and skeleton application plans and packages. You should estimate the space that is needed for plans, packages, and cached prepared statements.

In DB2 9 for z/OS, the EDM pool has been separated into five pools:

- EDM pool above the 2 GB bar
- EDM pool below the 2 GB bar
- Skeleton pool
- Database descriptor (DBD) pool
- Statement pool

The EDM pool is currently split between being above and below the 2 GB bar. This includes the pools used for dynamic statement cache. The EDM pool space for database descriptors, skeleton cursor tables, and skeleton package table are all above the 2 GB bar.

The important parts of the EDM pool for packages are the EDM pool area where DB2 stores the cursor tables and plan tables as well as the statement cache.

See *DB2 Version 9.1 for z/OS Installation Guide, GC*18-9846 for details on how to size the EDM pools.

11.3 Package monitoring

This section looks at performance monitoring from two points of view: monitoring the access path of the SQL statements in the package, and monitoring package performance.

11.3.1 Access path

BIND PACKAGE supports the EXPLAIN option to monitor the access path selected by DB2. When you bind a package with EXPLAIN(YES), DB2 inserts the information into the package_owner.PLAN_TABLE. There are additional EXPLAIN tables that DB2 populates if they exist. The DSN_STATEMNT_TABLE can be useful to determine if a package has changed access paths. Example 11-2 shows the sample query to retrieve important columns from the PLAN_TABLE.

Example 11-2 Sample query to display PLAN_TABLE

```
SELECT
 SUBSTR(A.COLLID,1,8)           AS COLLID
,SUBSTR(A.PROGNAME,1,8)         AS PROGNAME
--,SUBSTR(A.VERSION,1,26)       AS VERSION
,SUBSTR(DIGITS(A.QUERYNO),5,5)  AS QUERY_NUMB
,SUBSTR(DIGITS(A.QBLOCKNO),3,3) AS QRY_BLK
,SUBSTR(DIGITS(A.PLANNO),3,3)   AS PLN_NO
,SUBSTR(DIGITS(A.METHOD),3,3)   AS MTH
--,SUBSTR(A.CREATOR,1,8)        AS TABLE_CREATOR
,SUBSTR(A.TNAME,1,18)           AS TABLE_NAME
```

```
,SUBSTR(DIGITS(A.TABNO),3,3)    AS TAB_NO
,A.ACCESSTYPE                   AS A_C_C_T_Y_P
--,SUBSTR(A.ACCESSCREATOR,1,8)  AS INDEX_CREATOR
,SUBSTR(A.ACCESSNAME,1,8)       AS INDEX_NAME
,A.MATCHCOLS                    AS M_C_O_L
,A.INDEXONLY                    AS I_X_O_N_L_Y
,A.PREFETCH                     AS P_R_E_F_C_H
,(SORTN_UNIQ    CONCAT
 SORTN_JOIN    CONCAT
 SORTN_ORDERBY CONCAT
 SORTN_GROUPBY)                 AS UJOGN
,(SORTC_UNIQ    CONCAT
 SORTC_JOIN    CONCAT
 SORTC_ORDERBY CONCAT
 SORTC_GROUPBY)                 AS UJOGC
FROM PLAN_TABLE A
WHERE A.PROGNAME = 'PGM10'
-- get the latest package from the PLAN_TABLE
AND A.BIND_TIME IN(SELECT MAX(B.BIND_TIME) FROM PLAN_TABLE B
                   WHERE A.PROGNAME = B.PROGNAME)
ORDER BY COLLID,PROGNAME,QUERYNO,QBLOCKNO,PLANNO
```

Figure 11-1 shows the output from the PLAN_TABLE for PGM10. The queries against SYSTABLES and SYSINDEXES use table scans, indicated by the “R” in the ACCESSTYPE column. The query against SYSTABLESPACE uses a one column matching index scan, indicated by the “I”: in ACCESSTYPE and a “1” in MATCHCOLS (MCOLS on the report). The query against SYSPACKAGE uses a non-matching index scan, indicated by as ACCESSTYPE of “I” and a “0” in MATCHCOLS.

```
                                                A               I  P
                                                C             M X  R
                                                C             C O  E  N U  C U
                                                T             O N  F  E J  O J
COLL       QUERY QRY PL      TABLE         TAB Y  INDEX      L L  C  W O  M O
 ID  PGM   NUMB  BLK NO  MTH NAME          NO  P  NAME       S Y  H     G  P G
---- ----- ----- --- --- --- ------------- --- -- -------- ---- -- -- ---- ----
TEMP PGM10 00001 001 001 000 SYSTABLES     001 R             0 N  S  NNNN NNNN
TEMP PGM10 00001 001 002 003               000               0 N     NNNN NNYN
TEMP PGM10 00002 001 001 000 SYSTABLESPACE 001 I  DSNDSX01   1 Y     NNNN NNNN
TEMP PGM10 00002 001 002 003               000               0 N     NNNN NNYN
TEMP PGM10 00003 001 001 000 SYSCOLUMNS    001 R             0 N  S  NNNN NNNN
TEMP PGM10 00003 001 002 003               000               0 N     NNNN NNYN
TEMP PGM10 00004 001 001 000 SYSPACKAGE    001 I  DSNKKX02   0 Y  S  NNNN NNNN
TEMP PGM10 00004 001 002 003               000               0 N     NNNN NNYN
```

Figure 11-1 Sample output from PLAN_TABLE

11.3.2 DSN_STATEMNT_TABLE

The DSN_STATEMNT_TABLE can be used to determine if a package has changed access paths. See A.12, “Showing packages that have changed access paths” on page 266 for a query that assists in determining changes at a higher level than at the statement level. The query in Figure 11-2 shows relevant data from the DSN_STATEMNT_TABLE.

```
SELECT
  COLLID
, PROGNAME
, QUERYNO
, STMT_TYPE
, COST_CATEGORY
, PROCMS
, PROCSU
, STMT_ENCODE
, TOTAL_COST
, EXPLAIN_TIME
FROM PAOLOR3.DSN_STATEMNT_TABLE
WHERE PROGNAME = 'PGM23'
```

Figure 11-2 Sample query for the DSN_STATEMNT _TABLE

An index that PGM23 depended on was dropped between the first and second bind. The two rows shown in Figure 11-3 demonstrate that the cost of the statement has changed. By examining the cost figure, you can get an idea whether access paths have changed due to a REBIND. If the cost has changed, there is a good probability that the access path has changed. This method is not 100% accurate, nor does it tell you if the package runs better or worse than before the change. However, the TOTAL_COST does tell you whether DB2 estimates if the query will run better. In this case, the cost changed from 31 to 32 after dropping the index. The REBIND now uses the remaining index. DB2 indicates it would prefer the index that was dropped.

```
 PACK   QUERY STMT COST PROC PROC  STMT   TOTAL EXPLAIN
COLLID NAME   NO   TYPE CATE  MS   SU  ENCODE  COST   TIME
------ ----- ----- ---- ---- ---- ---- ------ ------ --------------------------
TEMP   PGM23    47 SELE A       1    5 U          31 2008-11-19-19.23.00.248835
TEMP   PGM23    47 SELE A       1    6 U          32 2008-11-19-19.27.16.672335
```

Figure 11-3 DSN_STATEMNT _TABLE after index drop and rebind

Tip: Investigate the access paths of all packages that have a changed cost estimate. Even when the change is small, or even if DB2 estimates the cost is lower, the performance of your package might change.

11.3.3 Performance measurements

Accounting information is available for packages through accounting traces 7, 8 and 10. Performance analysis tools can be used to analyze this data. The intent of this chapter is not to analyze a performance problem, but to show where you can find the data. In this chapter, we use the reports from IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS (OMEGAMON PE).

OMEGAMON PE can assist you in monitoring your packages, both as an online monitor as well as a batch reporting tool. Figure 11-4 shows the initial screen with the OMEGAMON PE product. This is how you would start your session to use OMEGAMON PE as an online monitor. In order to get data at the package level, you must be running accounting traces 7 and 8. Optionally, you can use accounting trace 10 to report on locks at the package level, although there is additional overhead in running this trace.

11.3.4 Online monitoring

In this section, we show how to display package level information from the online monitor portion of OMEGAMON PE. We begin with the initial OMEGAMON PE entry screen as shown in Figure 11-4.

```
 IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS
Command ===> ____________________________________________________________

Select one of the following.

3_ 1. Create and execute reporting commands
   2. View online DB2 activity - Classic Interface
   3. View online DB2 activity - PE ISPF OLM
   4. Maintain parameter data sets
   5. Customize report and trace layouts
   6. Exception profiling
```

Figure 11-4 Initial entry screen for OMEGAMON PE

As you navigate into the product, you tell OMEGAMON PE what you would like to see. We start with the thread activity screen as shown in Figure 11-5.

```
Select one of the following.

1_  1.  Display Thread Activity
    2.  Display Statistics
    3.  Display System Parameters
    4.  Options
    5.  Control Exception Processing
    6a. Collect Report Data - General
    6b. Collect Report Data - For Buffer Pool Analysis
    7.  Create and execute reporting commands
    8.  Maintain parameter data sets
    9.  Explain
```

Figure 11-5 Request the thread activity panel

OMEGAMON PE displays the thread detail screen as shown in Figure 11-6. You can look at various aspects of how your package is executing. On this screen, the interesting information is the class 1,2,3,7 and 8 elapsed and CPU times, the number of DML statements executed, the number of GETPAGEs, COMMITs, ROLLBACKs, buffer updates, and prefetch requests, as well as other information. The next item we want to investigate is the package detail, so we put an **S** next to that option and press the <ENTER> key.

```
08/11/20 13:35                  Thread Detail                  DB9A DB9A V9
Command ===> ____________________________________________________________________

For details, place any character next to heading, then press Enter.
                                                                    More:     +
_ Thread Identification
  Primauth . . . . . : PAOLOR3          Correlation Name  . . : PAOLOR3Q
  Planname . . . . . : PLAN1            Connection type . . . : TSO
  Connection ID  . . . : BATCH          Type  . . . . . : ALLIED
  Requesting Location  : DB9A           Status  . . . . . : I/O
_ Current Package      : PGM09
_ Times                                            Elapsed           CPU
  Class 1  . . . . . . . . . . . . . . . . . :     16.256364       1.757535
  Class 2  . . . . . . . . . . . . . . . . . :     16.090141       1.598439
  Class 3  . . . . . . . . . . . . . . . . . :     14.388724            N/A
  Class 7  . . . . . . . . . . . . . . . . . :     16.090136       1.598434
  Class 8  . . . . . . . . . . . . . . . . . :     14.388724            N/A
_ Locking Activity
  Timeouts . . . . . . . . . . . . . . . . . :              0
  Deadlocks  . . . . . . . . . . . . . . . . :              0
  Suspensions  . . . . . . . . . . . . . . . :              2
  Lock escalations . . . . . . . . . . . . . :              0
  Maximum page locks held  . . . . . . . . . :              0
_ Locked Resources
_ RID List Processing
  Unsuccessful - any reason  . . . . . . . . :              0
_ SQL Activity, Commits and Rollbacks
  DML . . :     114640 Commit  . . . . . . . :              0
  DCL . . :          0 Rollback  . . . . . . :              0
  DDL . . :          0 Changes/Commit  . . . :            0.0
_ Buffer Manager Activity
  Getpage requests . . . . . . . . . . . . . :         111593
  Buffer updates . . . . . . . . . . . . . . :           1029
  Prefetch requests  . . . . . . . . . . . . :           6604
  Synchronous I/O  . . . . . . . . . . . . . :            656
_ SQL Statement and Program  . . . . . . . . : PGM09
  Distributed Data
  Requester elapsed time . . . . . . . . . . :            N/P
_ IFI (Class 5) and Data Capture
_ Query Parallelism Data
  Data Sharing Locking Activity
  Suspensions  . . . . . . . . . . . . . . . :            N/P
  Group Buffer Pools Activity
```

Figure 11-6 OMEGAMON PE thread detail panel

Figure 11-7 shows the details of class 7 and class 8 times, which are the accounting times for the package, broken up by the various components. In our example here, most of the time is spent in TCB suspension, other read I/O.

```
 Current Package
Command ===> ____________________________________________________________

                                                              More:     +
Location  . . . . . . . . . . . . . . : DB9A
Collection ID . . . . . . . . . . . . : TEMP
Program name  . . . . . . . . . . . . : PGM09
Consistency token . . . . . . . . . . : X'186A66F41BBF16B7'

DB2 entry/exit events . . . . . . . . :        229279
SQL statements issued . . . . . . . . :        114639

                                             Total       Current
                                         Execution     Execution
Elapsed time (Class 7)  . . . . . . . :     16.090136      0.015898
CPU time  . . . . . . . . . . . . . . :      1.598434           N/A
  TCB . . . . . . . . . . . . . . . . :      1.598434           N/A
  Parallel tasks  . . . . . . . . . . :      0.000000           N/A
Waiting time  . . . . . . . . . . . . :     14.491701           N/A
Suspension time (Class 8) . . . . . . :     14.388724           N/A
  TCB . . . . . . . . . . . . . . . . :     14.388724           N/A
  Parallel tasks  . . . . . . . . . . :      0.000000           N/A
  TCB . . . . . . . . . . . . . . . . :     14.388724           N/A
  Parallel tasks  . . . . . . . . . . :      0.000000           N/A
Not accounted . . . . . . . . . . . . :      0.102978           N/A
                                        Elapsed time        Events
Suspensions (Class 8) . . . . . . . . :     14.388724          7234
  Locks and latches . . . . . . . . . :      0.001133           155
  Synchronous I/O . . . . . . . . . . :      0.409253           656
  Other read I/O  . . . . . . . . . . :     13.978337          6423
  Other write I/O . . . . . . . . . . :      0.000000             0
  Services task switch  . . . . . . . :      0.000000             0
  Archive log (quiesce) . . . . . . . :      0.000000             0
  Archive log read  . . . . . . . . . :      0.000000             0
  Drain lock  . . . . . . . . . . . . :      0.000000             0
  Claim release . . . . . . . . . . . :      0.000000             0
  Page latch  . . . . . . . . . . . . :      0.000000             0
  Stored procedures . . . . . . . . . :           N/A           N/A
  Notify messages . . . . . . . . . . :      0.000000             0
  Global contention . . . . . . . . . :      0.000000             0
```

Figure 11-7 Package details, class 7 and class 8 times

Figure 11-8 shows the thread times. On this screen, you can see the class 1 and class 2 times, along with wait times and lock suspensions. This screen confirms that most of the time in the package is spent waiting for other read I/O.

```
Thread Times
Command ===> ____________________________________________________________

                                                              More:     +
                              Class 1        Class 2
                              In Appl         In DB2    Outside DB2
Elapsed time . . . . . :  2:15.319790    2:14.121138       1.198652
CPU time . . . . . . . :    14.262284      12.995761       1.266522
  TCB  . . . . . . . . :    14.262284      12.995761       1.266522
  TCB - Stored Proc  . :     0.000000       0.000000
  Parallel tasks . . . :     0.000000       0.000000
Waiting time . . . . . :          N/A    2:01.125377
Suspension time  . . . :          N/A    2:00.172534
  TCB  . . . . . . . . :          N/A    2:00.172534
  Parallel tasks . . . :          N/A       0.000000
Not accounted  . . . . :          N/A       0.952843
                                              Time           Event
Suspensions (Class 3) . . . . . :  2:00.172534           59077
  Locks and latches . . . . . :     0.009836             695
  Synchronous I/O . . . . . :     3.295791            5075
  Other read I/O  . . . . . :  1:56.866906           53307
  Other write I/O . . . . . :     0.000000               0
```

Figure 11-8 OMEGAMON PE thread times

Figure 11-9 shows the SQL activity that has been executed by your package. In this case, we have issued 794,825 DML statements.

```
SQL Activity
Command ===> ____________________________________

                                        More:     +
Incremental bind  . . . . . :          0
Reoptimization  . . . . . :          0
Prepare statement match . . . . . :          0
Prepare statement no match  . . . . . :          0
Implicit prepare  . . . . . :          0
Prepare from cache  . . . . . :          0
Cache limit exceeded  . . . . . :          0
Prepare statement purged  . . . . . :          0
Commit  . . . . . :          0
Rollback  . . . . . :          0
Changes/Commit  . . . . . :        0.0

Total DML . . . . . :     794825
  Select  . . . . . :          0
  Insert  . . . . . :          0
  Update  . . . . . :          0
  Delete  . . . . . :          0
  Prepare . . . . . :          0
```

Figure 11-9 OMEGAMON PE SQL activity panel

Figure 11-10 shows the beginning of the SQL statement as well as various package attributes including COLLID, version (not used here) and CONTOKEN.

```
SQL Statement and Program
Command ===> ____________________________________________________________

                                                                More:     +
_ SQL Statement : DECLARE CSR1 CURSOR FOR SELECT SUBSTR ( DBNAME , 1

Location . . . . . . . . . . . . . : DB9A
Collection ID  . . . . . . . . . . : TEMP
Program name . . . . . . . . . . . : PGM09
Nested activity name . . . . . . . : N/P
Program type . . . . . . . . . . . : Package
Consistency token  . . . . . . . . : X'1869A73515808867'
Version  . . . . . . . . . . . . . : N/P
Statement type . . . . . . . . . . : OPEN
Statement number . . . . . . . . . : 73
Current SQL ID . . . . . . . . . . : PAOLOR3
Bind type  . . . . . . . . . . . . : Static
Cached dynamic SQL identifier  . . : 0
Thread status  . . . . . . . . . . : Performing I/O
Database name  . . . . . . . . . . : N/P
Page set name  . . . . . . . . . . : N/P
Page number  . . . . . . . . . . . : N/P
Program type . . . . . . . . . . . : Package
Consistency token  . . . . . . . . : X'1869A73515808867'
Version  . . . . . . . . . . . . . : N/P
Statement type . . . . . . . . . . : OPEN
Statement number . . . . . . . . . : 73
Current SQL ID . . . . . . . . . . : PAOLOR3
Bind type  . . . . . . . . . . . . : Static
Cached dynamic SQL identifier  . . : 0
Thread status  . . . . . . . . . . : Performing I/O
Database name  . . . . . . . . . . : N/P
Page set name  . . . . . . . . . . : N/P
Page number  . . . . . . . . . . . : N/P
Elapsed time . . . . . . . . . . . : N/P
CPU time . . . . . . . . . . . . . : N/P
_ Used Buffer Pools
  Total getpages . . . . . . . . . : N/P
  Total synch read I/O . . . . . . : N/P
```

Figure 11-10 SQL statement and detail panel

Figure 11-11 shows the entire SQL statement that you have drilled down to see.

```
SQL Statement                Row 1 to 3 of 3
Command ===> ____________________________________________________________


Program name . . . . . . . . . . . : PGM09
SQL Statement
DECLARE CSR1 CURSOR FOR SELECT SUBSTR ( DBNAME , 1 , 8 ) ,
SUBSTR ( TSNAME , 1 , 8 ) , SUBSTR ( NAME , 1 , 44 ) FROM SYSIBM
. SYSTABLES ORDER BY 1 , 2 , 3
*************************** Bottom of data ****************************
```

Figure 11-11 SQL statement detail

11.3.5 Optimization Service Center

The Optimization Service Center (OSC) is a full replacement of Visual Explain with additional features. Optimization Service Center can be used to show the access paths that a package is using with a graphical interface. It has a Statistics Advisor to assist you with tuning queries and workloads. It can assist you with annotating queries to assist in performance tuning. It can also assist you with explaining queries that are in your dynamic statement cache.

Optimization Service Center is available as a free download from the Web, or it can be ordered as a part of the DB2 Accessories Suite for z/OS, a no-extra-charge feature made available first with DB2 9 for z/OS. Support for the product is only available if you order it through the DB2 Accessories Suite. Figure 11-12 shows the differences between Visual Explain, Optimization Service Center, and Optimization Expert (OE). For more details on Optimization Service Center, refer to *IBM DB2 9 for z/OS: New Tools for Query Optimization*, SG24-7421.

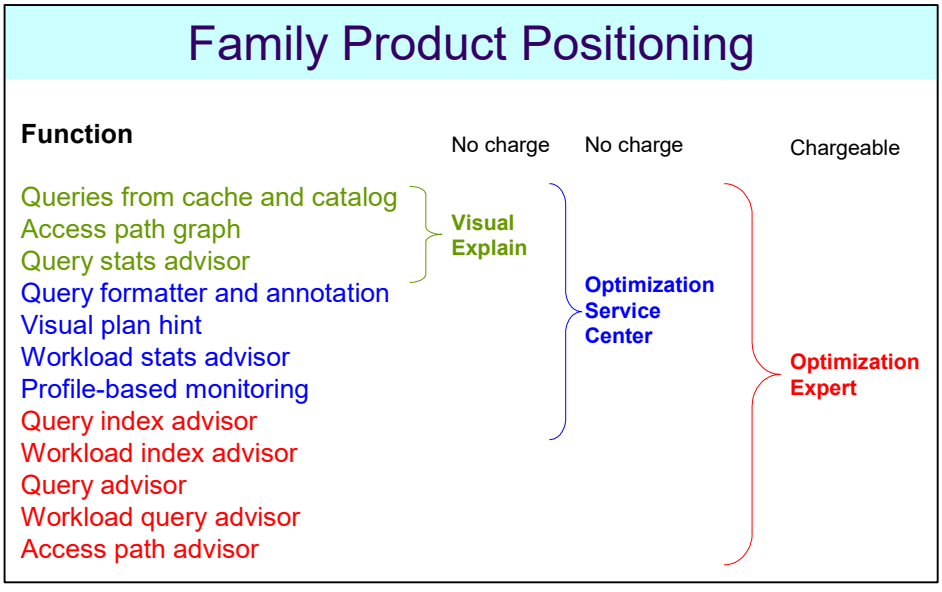

Figure 11-12 Comparison of Visual Explain, Optimization Service Center, and OE

11.4 Investigating a query from a package

Here we extract an SQL statement from a package bound on the DB2 subsystem. We create a new project called “Static01” and click **Identify Target Query**.

To investigate a query from a package:

1. Use the pull-down menu to select “Catalog plan or package” as the query source (see Figure 11-13). Use this option to extract the SQL statement from a DB2 package or plan.

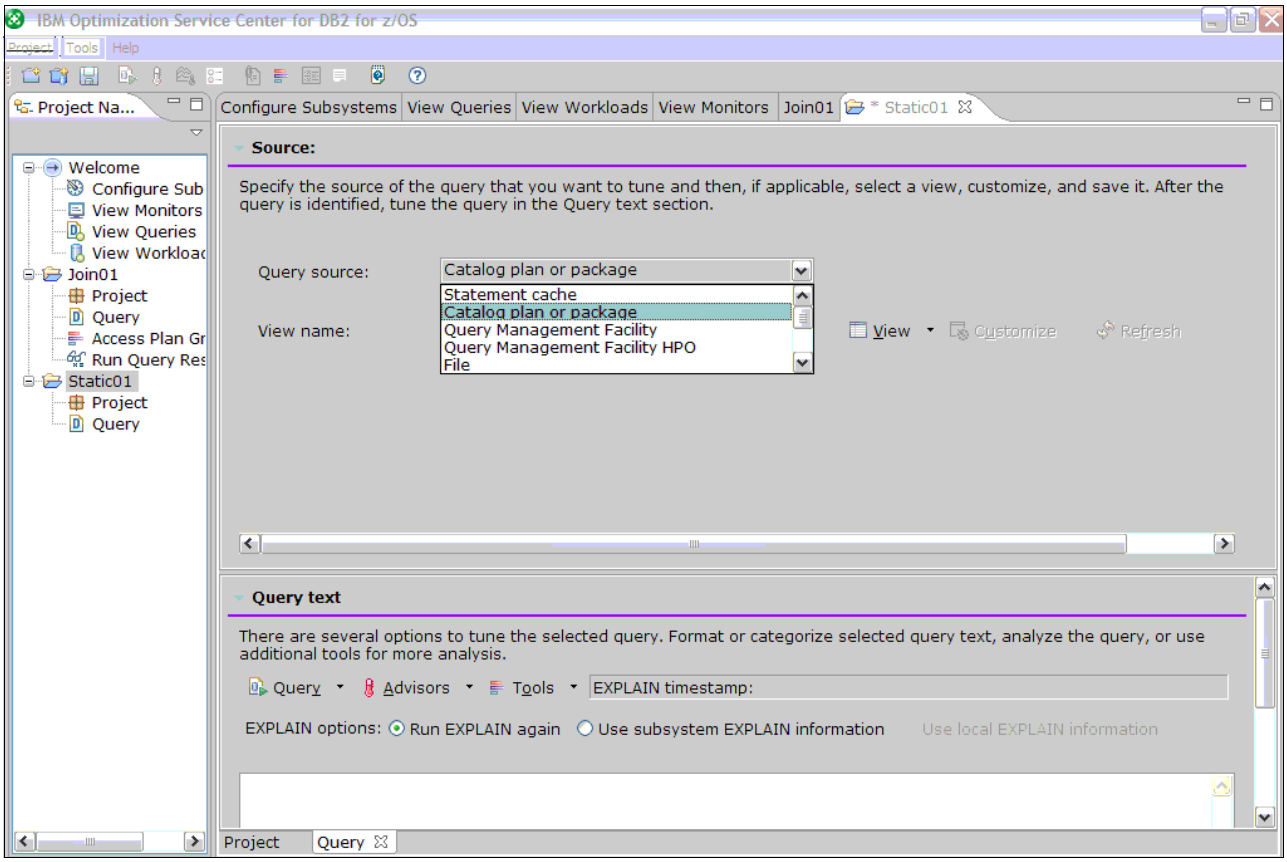

Figure 11-13 Choosing the Catalog as the query source

2. Now we define an Optimization Service Center view to restrict the number of SQL statements extracted from the DB2 Catalog. If you use the default view, the Optimization Service Center produces an empty list. To create a new view, you can either use the pull-down list next to the description View name and select **<New view...>,** as shown in Figure 11-14, or click the **View** icon to the right of the view name and select **New** from the drop-down menu.

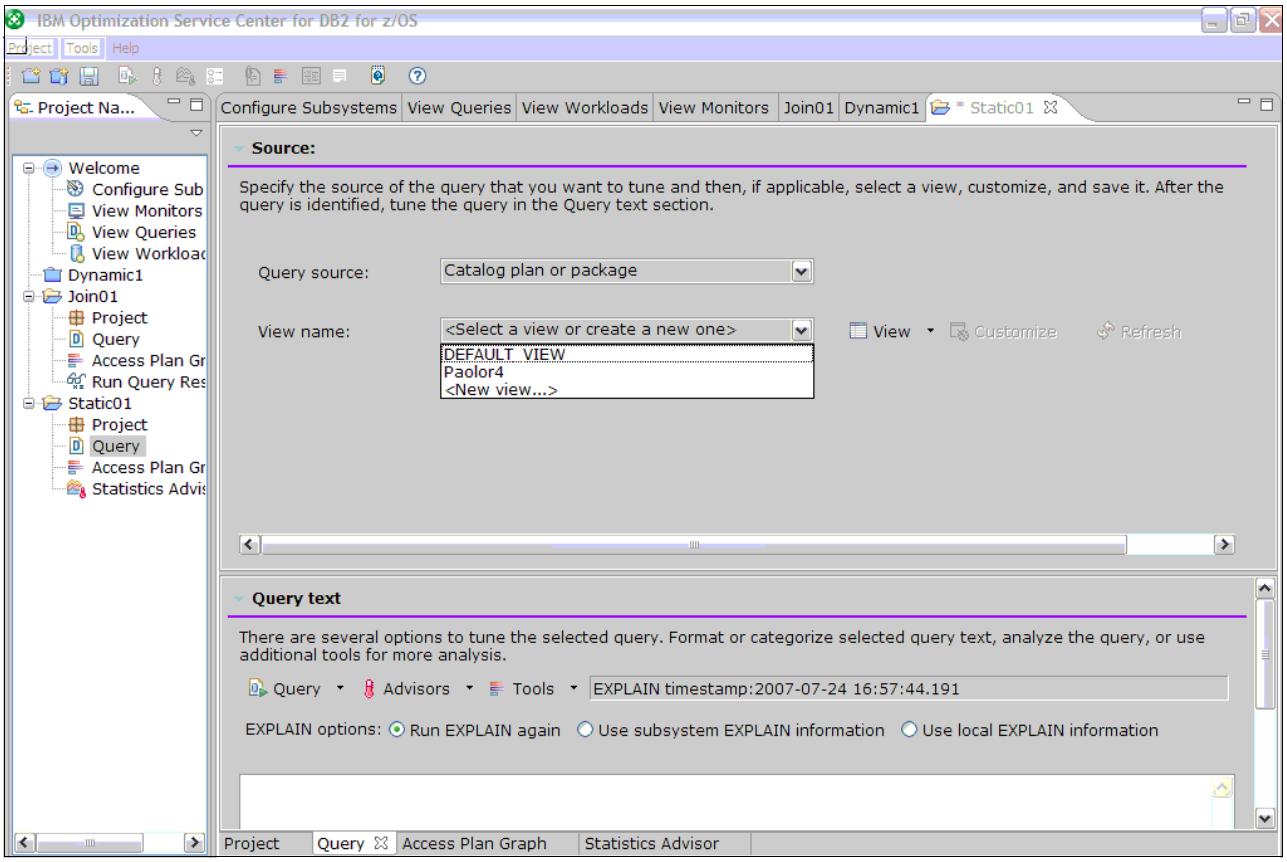

Figure 11-14 Selecting a view

3. The Create View panel opens as shown in Figure 11-15. In the figure, you typed in a name for the View name (PAOLOR4) and on the Packages tab, you double-clicked the Value column for OWNER and typed PAOLOR4. This extracts SQL statements from all packages with Owner authorization ID PAOLOR4. You can specify other criteria, as shown in the figure, to restrict the packages that are processed.

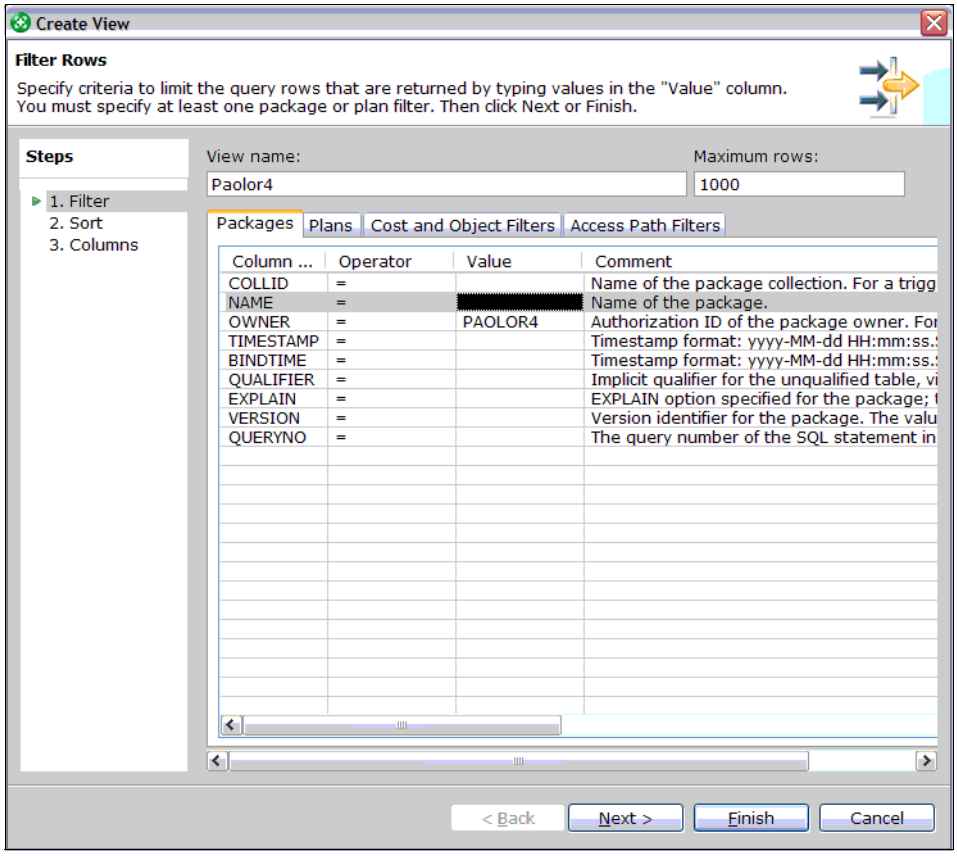

Figure 11-15 Create View panel

4. You can click the **Plans** tab to specify restrictive criteria for plans. Leave the “Plan options” blank if you are selecting the Package filter criteria. Make sure that all options are deselected on the Access Path Filter tab initially (see Figure 11-16), as selecting options here might cause SQL statements to be discarded.

> **Note:** If you specify selection options for both packages and plans, for example, specifying owner for both, you might get the SQL statements from your packages listed twice. Optimization Service Center finds them once based on the package filter specifications. It then finds the plans and locates all the packages accessible through those plans. If it finds the same packages, they get included twice.

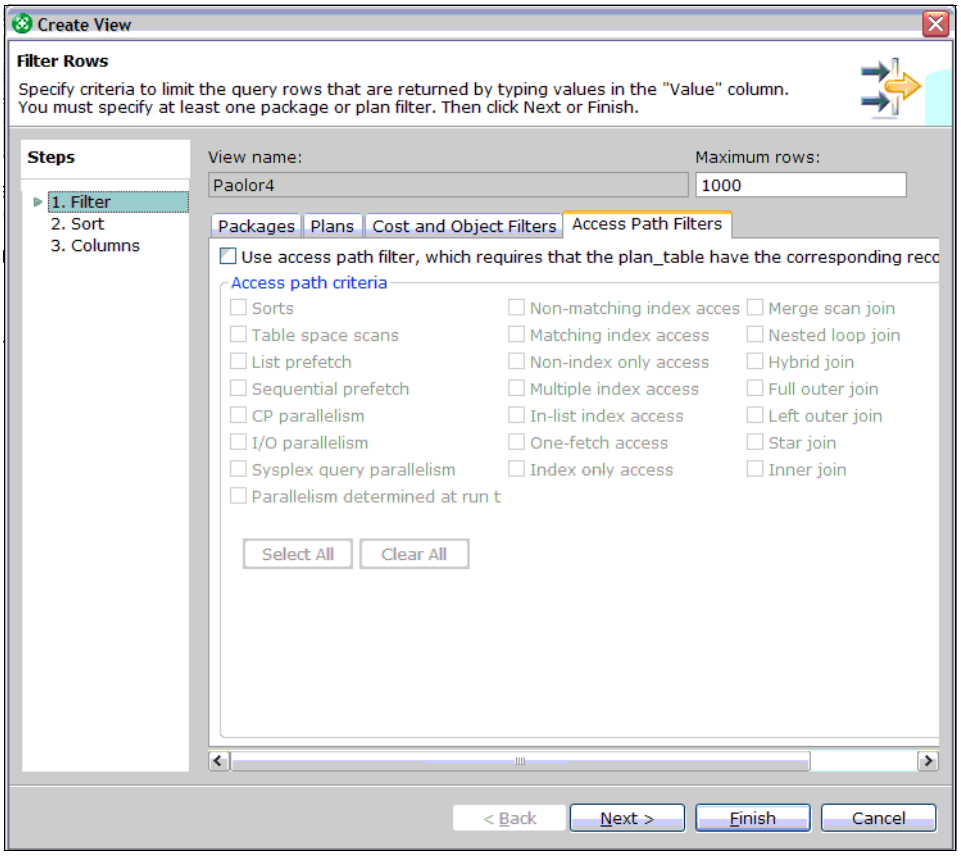

Figure 11-16 Access Path Filters tab

5. Specify the filter criteria for the view and click **Next** to specify the ordering of the SQL statements. In Figure 11-17, you selected the column STMTNO and clicked the right single arrow button to move it to the right-hand pane.

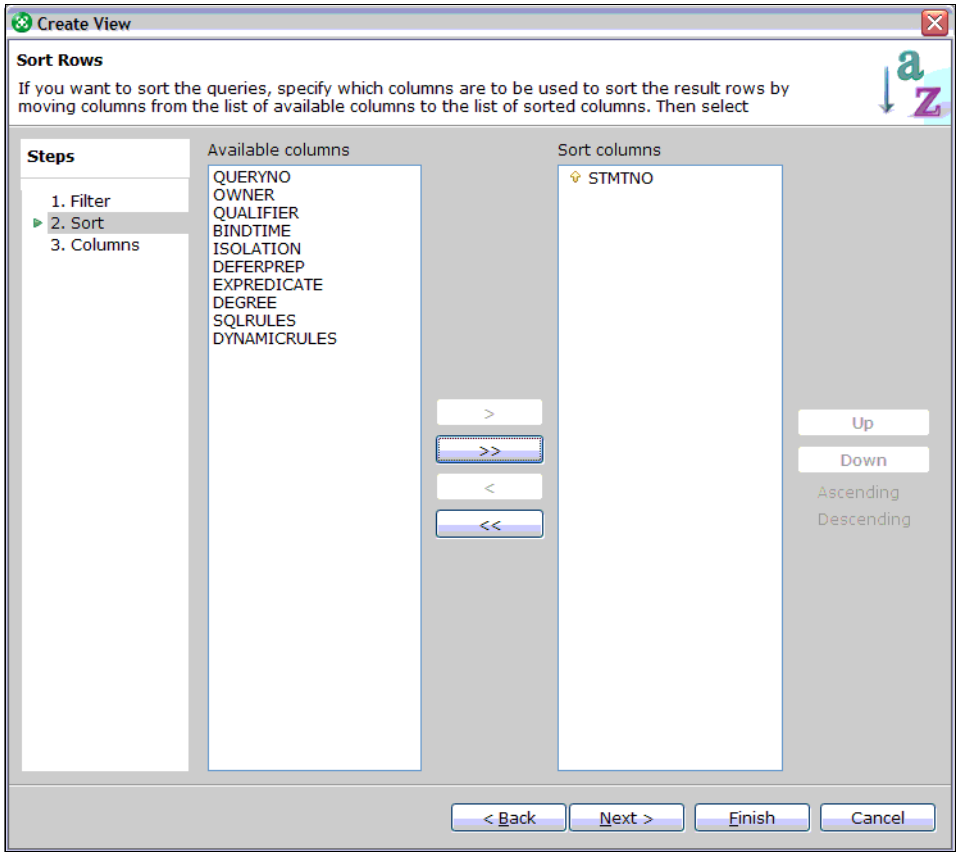

Figure 11-17 Choose sort sequence

6. The SQL statements returned from the DB2 Catalog are sorted by statement number. Click **Next** to select the data to be returned for each SQL statement as shown in Figure 11-18. Take the defaults, so click **Finish** to complete the view definition. As soon as you click Finish, Optimization Service Center starts to access the DB2 Catalog. You see a progress indicator during the access.

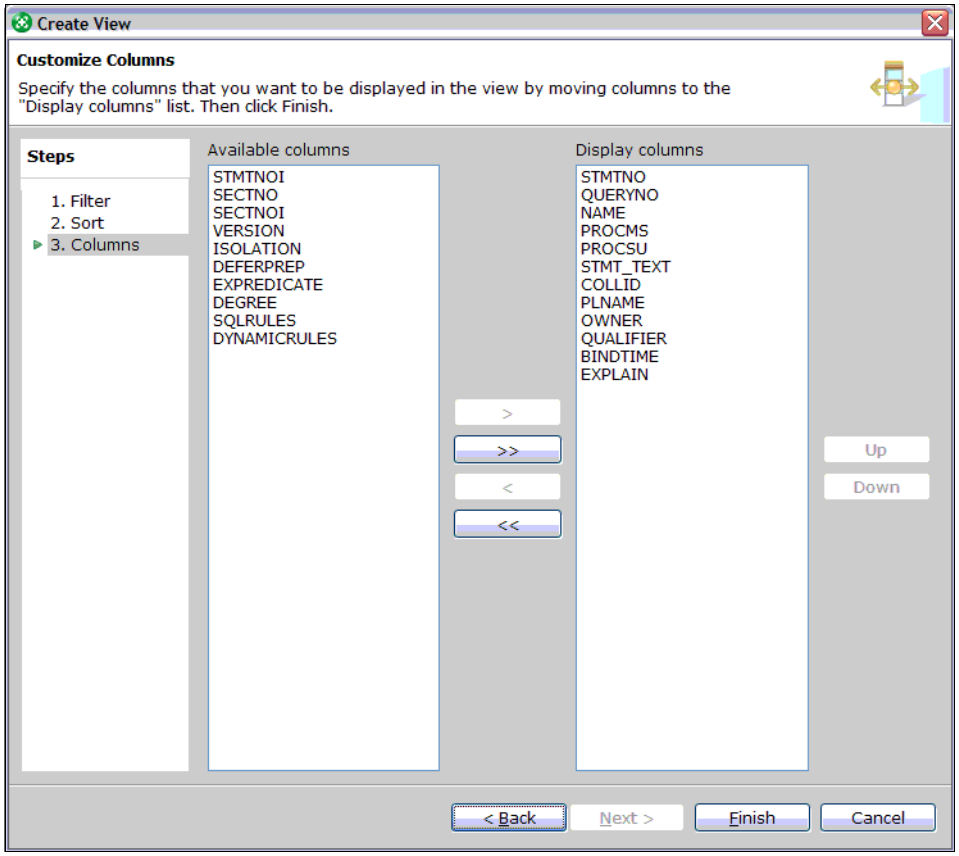

Figure 11-18 Choose data to display

7. When Optimization Service Center has retrieved the statements from the packages with owner PAOLOR4 (in our example), it returns to the Source panel with the SQL statements listed (see Figure 11-19).

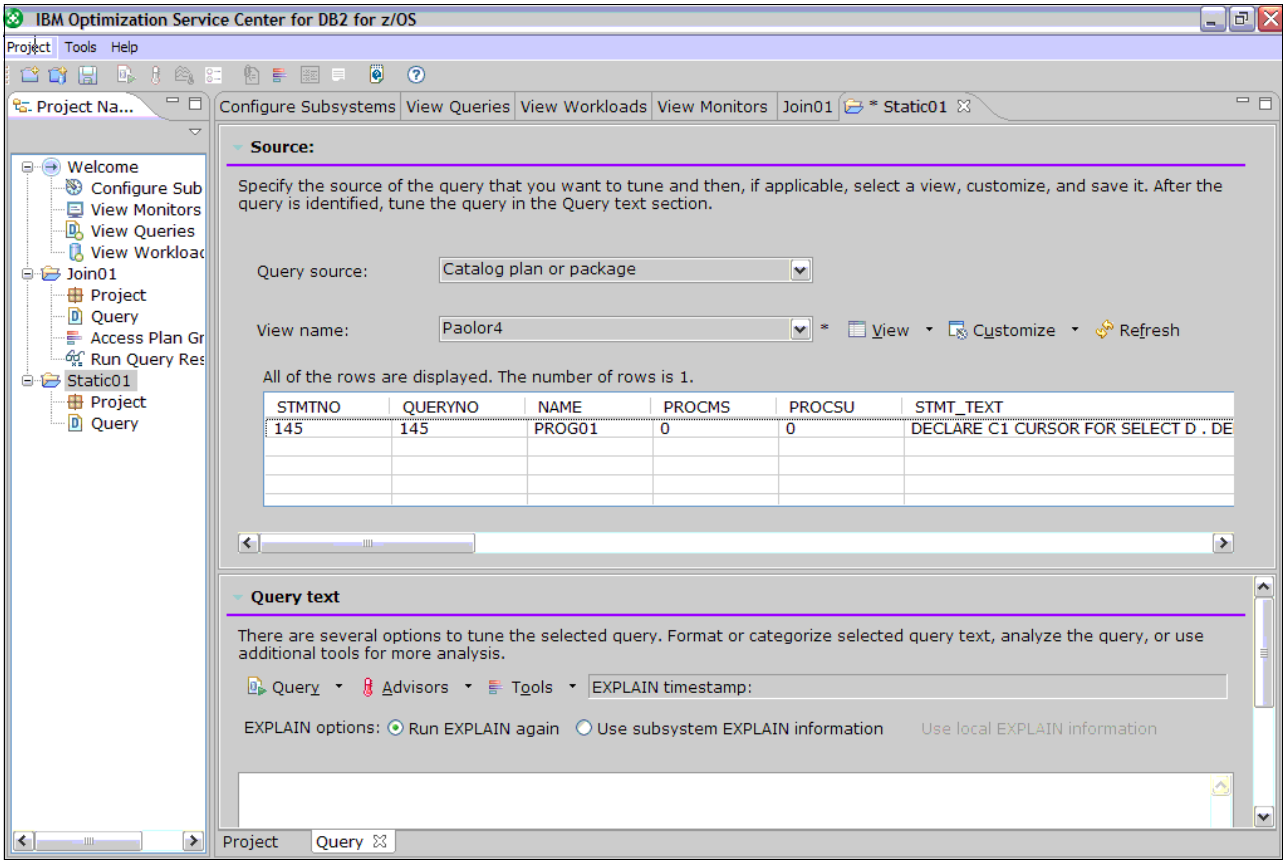

Figure 11-19 The SQL statements retrieved from the DB2 Catalog

8. In our example, there is only one SQL statement from a single package. You can drag the scroll bar below the list of statements to the right to see more information about the statements. If you click one statement, the text of the statement appears in the Query Text panel, as shown in Figure 11-20.

   You can see the Access Plan Graph by clicking **Tools** on the Query text panel and selecting **Access Plan Graph** from the drop-down menu.

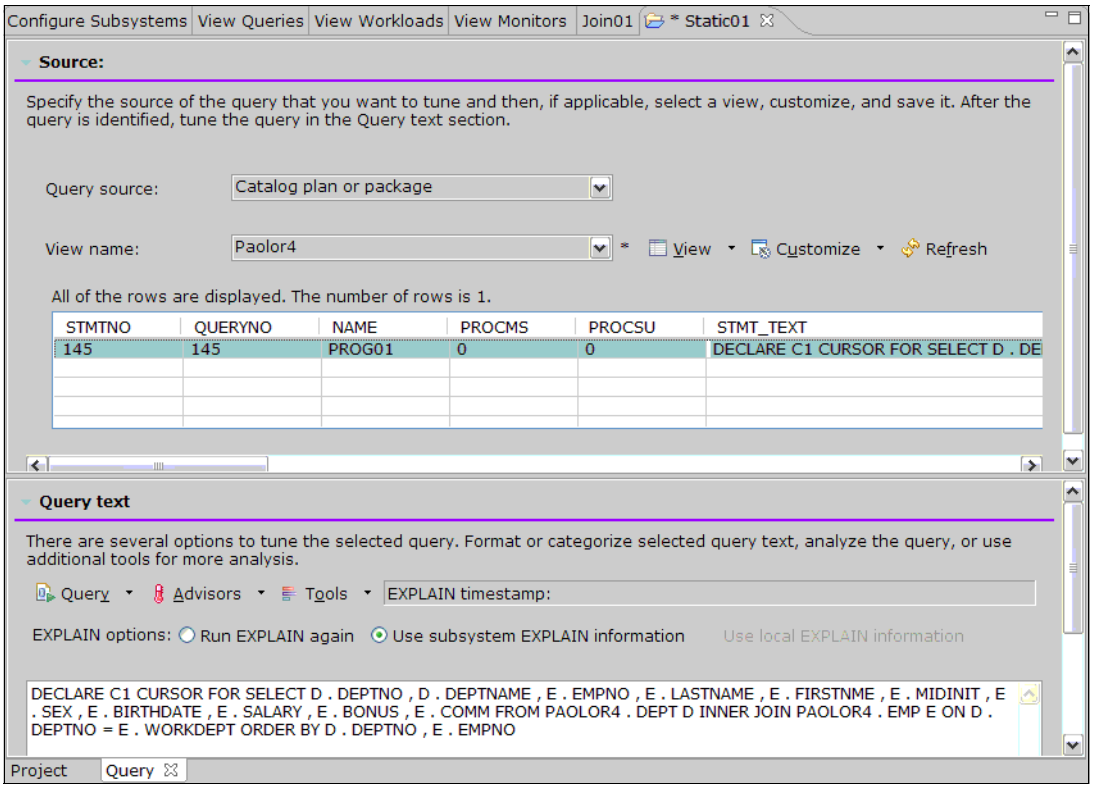

Figure 11-20 Seeing the statement text

9. This produces the access path graph as shown in Figure 11-21.

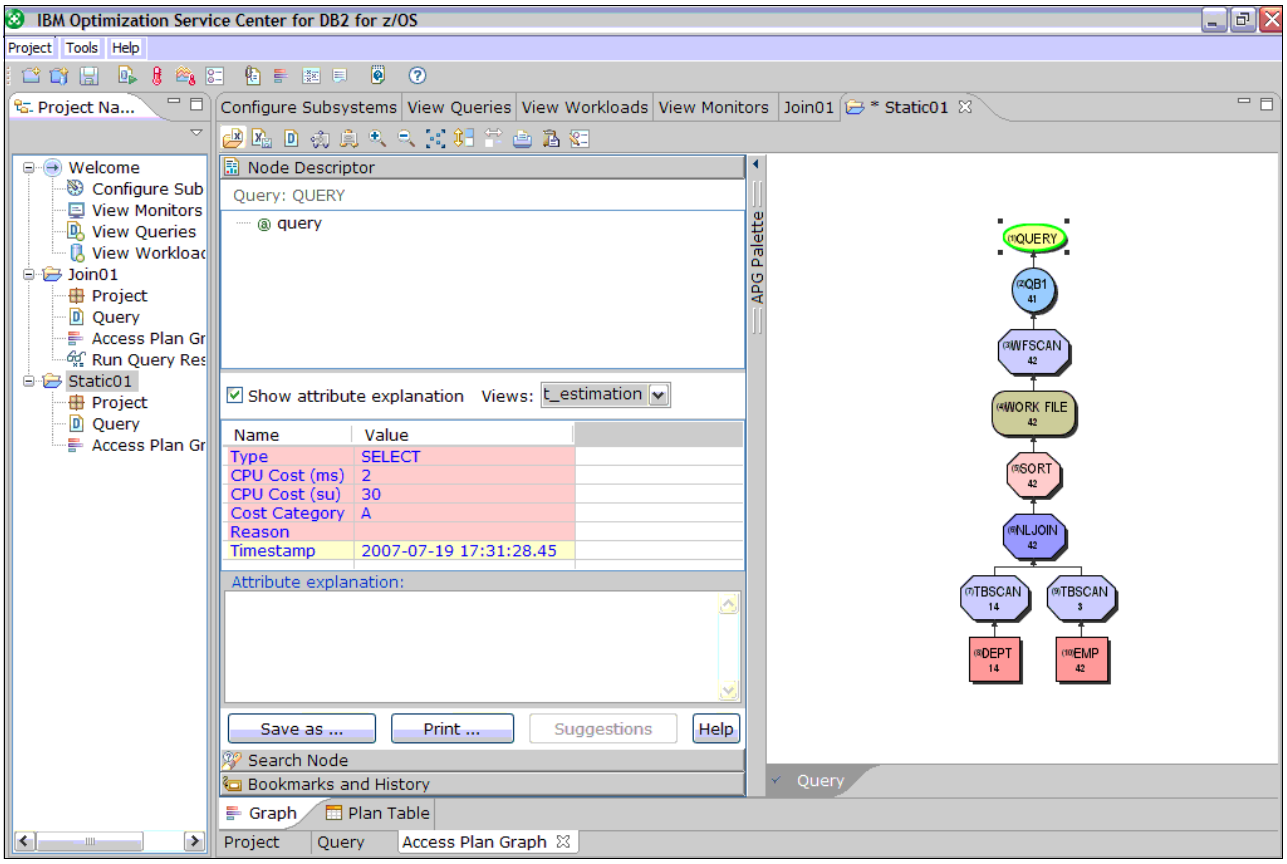

Figure 11-21 The Access Plan Graph

11.5 Considerations for packages with a large number of sections

A general consideration to keep in mind is that if you have packages with many sections and only a very small number of SQL statements are actually executed, the cost of allocation and deallocation is large relative to the cost of the SQL execution. In other words, a large common module that handles all SQL for an application is probably not a good idea, smaller, frequently used packages should perform better.

Two very specific infrequent situations have become critical in some environments:

- When a small number of short-running SQL statements out of many SQL statements in a large package are executed
- When a set of short-running SQL statements out of many different packages are executed.

The package CPU cost might be individually very small, but if the package is executed a very large number of times, then the total CPU time for the package can become expensive.

A detailed discussion of these issues can be found in the IBM Redpaper publication, *DB2 for z/OS: Considerations on Small and Large Packages,* REDP-4424.

11.5.1 CPU overhead for allocating and deallocating packages

The general view of this CPU increase is that there are four important performance criteria:

- You have very large packages (over 200 sections, for example).
- You only execute only a small number of SQL statements out of the hundreds in the package.
- The SQL statements that execute are very short in duration, as would be found in a transactional system.
- The package is executed a very large number of times,

If all four of these criteria are met, then the overhead of allocating and deallocating these packages in SPT01 can become a critical performance bottleneck. To determine if you have packages with more than 200 sections, use the query in Example 11-3.

Example 11-3 How to determine if you might be affected by package allocation

```
SELECT SUBSTR(NAME,1,8) AS NAME,
       MAX(SECTNOI)     AS MAX_SECTNO
FROM SYSIBM.SYSPACKSTMT
GROUP BY NAME
HAVING MAX(SECTNOI) > 200
ORDER BY MAX_SECTNO DESC;
```

11.5.2 Performance measurements

Figure 11-22 shows a relative comparison with various number of sections from 2 to 2002 per packages.

The performance difference after migration from DB2 V7 to V8 has shown an increase in CPU associated with the major changes in functionalities

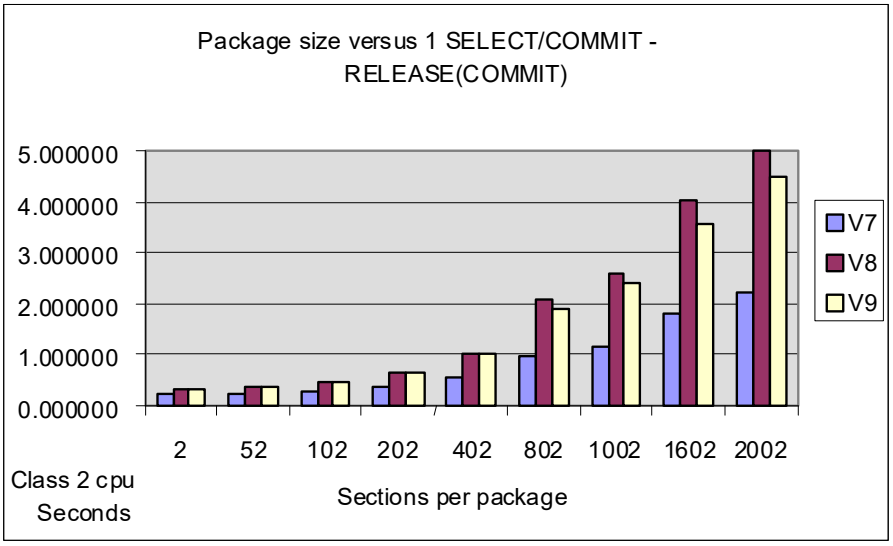

Figure 11-22 Influence of package size by DB2 version with RELEASE(COMMIT)

With DB2 V7, we can observe that a 202 section package takes about twice as long as a 2 section package, and a 2002 section package about 10 times longer.

With DB2 V8, the dependency on the number of sections is more critical: The largest of 2002 sections takes about 20 times longer in class 2 CPU over the same identical process in a 2 section package. The primary reasons that V8 takes a longer CPU time than V7 is that V8 is using 64-bit instructions to 32-bit instructions in V7, as well as additional functions in V8.

With DB2 9, this large package size performance issue with RELEASE(COMMIT) still exists, though it is slightly smaller.

Precise package cost is not directly shown by the instrumentation, but using a performance trace class (2, 3) you can identify SQL time and approximate the package cost as the difference between package class 7 CPU time and the sum of the class 7 CPU for the SQL statements in the package.

A good indicator, but not showing the total of the package cost, is the COMMIT CPU duration (IFCID 88-89). If the CPU cost of the SQL is relatively high, then the package cost might be relatively insignificant. This issue is most pronounced only when there are very short-running (even non I/O) SQL. You can also use an accounting report to note SQL/COMMIT activity.

11.5.3 Reducing CPU when there are many sections and minimal SQL usage

When a small number of short-running SQL statements, out of many SQL statements in a package, are executed, or when a set of short-running SQL statements is executed out of many different packages, one of the following tuning actions can be considered to reduce the increase in transaction CPU time:

- Redesign the application:

  You can reduce the number of SQL statements in a package to contain only those typically executed in a given package execution. Also try to keep those SQL statements typically executed in a transaction in one package.

- RELEASE(DEALLOCATE):

  Use the BIND option RELEASE(DEALLOCATE), instead of RELEASE(COMMIT), assuming virtual storage usage is not constrained. Additional significant improvement is available in DB2 9 for z/OS, such that the performance of small and large sized packages using RELEASE(DEALLOCATE) is much closer.

- CURSOR WITH HOLD:

  If any cursor is held open across the commit, the statement section is not released with RELEASE(COMMIT). If the cursor is closed, the statement section is released with RELEASE(COMMIT).

- KEEPDYNAMIC(YES):

  If you are using dynamic SQL, use KEEPDYNAMIC(YES) and explicit PREPARE (MAXKEEPD=0) with RELEASE(COMMIT). If you have done an explicit PREPARE, DB2 can not free the prepared section at COMMIT. This eliminates the overhead of building and destroying the PT in SPT01.

  For details about the use of dynamic statement caching see *DB2 for z/OS and OS/390: Squeezing the Most Out of Dynamic SQL*, SG24-6418.

11.6 Tracing packages

In this section, we discuss traces related to packages in DB2 for z/OS.

We cover the following considerations:

- Accounting class 7 and 8
- Accounting class 10
- Using tailored performance trace
- Auditing trace

11.6.1 Accounting class 7 and 8

DB2 for z/OS includes an instrumentation facility interface (IFI) that collects performance data at both system and application levels. In order to collect this data, you have to turn on specific traces to collect data for certain trace classes. To turn on traces, issue a -START TRACE command or configure DSNZPARM to have them start automatically with DB2. To monitor packages you generally need to start an accounting trace and collect data for classes 1, 2, 3, 7, and 8. Both elapsed time and CPU time are collected for each class. You can also activate class 10 when you need to collect package details.

See Table 11-3 for a description of the accounting classes.

Table 11-3 Description of accounting classes

Accounting class	Activated IFCIDs	Description of data collected
1	3,106,200,239	CPU time and elapsed time in application, at plan level
2	232	CPU time and elapsed time in DB2, at plan level
3	6-9,32,33,44,45,117,118,127, 128,170,171,174,175,213-216,226,227,242,243,321,322,329	Elapsed time due to suspensions, at plan level
7	232,240	CPU time and elapsed time in DB2, at package level
8	6-9,32,33,44,45,117,118,127, 128,170,171,174,175,213-216,226,227,241-243,321,322	Elapsed time due to suspensions, at package level
10	239	Detailed information about locks, buffers and SQL statistics at the package level. See also informational APAR II14421

Collecting accounting data

To collect accounting data, you need to start the accounting trace, if it has not been started with DB2 start-up. Example 11-4 shows a sample -START TRACE command that you can use to start an accounting trace.

Example 11-4 START TRACE command to monitor packages

```
-START TRACE(ACCTG) CLASS(1,2,3,7,8) DEST(SMF)
```

For more details on starting traces and the appropriate trace classes to choose, see *IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS Version 4.1.0, Reporting User's Guide*, SC18-9983-01. Another source of information on this tool is *A Deep Blue View of DB2 Performance: IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS*, SG24-7224.

Reporting accounting data

You can report accounting data from SMF data set using OMEGAMON PE. Example 11-5 shows a simple usage of OMEGAMON PE for Accounting Trace. For details, see *IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS Version 4.1.0: Report Command Reference*, SC18-9985.

Example 11-5 Reporting accounting trace

```
//DB2PE   EXEC PGM=DB2PE
//INPUTDD  DD  DSN=SMFDATA,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//JOBSUMDD DD  SYSOUT=*
//ACTTRC   DD  SYSOUT=*
//SYSIN    DD  *
ACCOUNTING TRACE  DDNAME(ACTTRC)
  LAYOUT(LONG)
  INCLUDE(CONNECT(SERVER))
EXEC
```

Package-level accounting data is reported if accounting class 7 and 8 are collected. Example 11-6 shows package level accounting trace in LAYOUT LONG.

Example 11-6 Package level accounting data

```
TESTSQCZ            VALUE                TESTSQCZ                   TIMES    TESTSQCZ                    TIME  EVENTS    TIME/EVENT
 ------------------ ------------------    ------------------  ------------    ------------------  ------------  ------  ------------
 TYPE               PACKAGE               ELAPSED TIME - CL7      0.002036    LOCK/LATCH             0.000000       0           N/C
 LOCATION           DB9A                  CP CPU TIME             0.000512    SYNCHRONOUS I/O        0.001232       1      0.001232
 COLLECTION ID      PAOLOR1                 AGENT                 0.000512    OTHER READ I/O         0.000000       0           N/C
 PROGRAM NAME       TESTSQCZ                PAR.TASKS             0.000000    OTHER WRITE I/O        0.000000       0           N/C
 CONSISTENCY TOKEN  64416B7951664B59      IIP CPU TIME            0.000250    SERV.TASK SWITCH       0.000000       0           N/C
 ACTIVITY TYPE      NONNESTED             SUSPENSION-CL8          0.001232    ARCH.LOG(QUIESCE)      0.000000       0           N/C
 ACTIVITY NAME      'BLANK'                 AGENT                 0.001232    ARCHIVE LOG READ       0.000000       0           N/C
 SCHEMA NAME        'BLANK'                 PAR.TASKS             0.000000    DRAIN LOCK             0.000000       0           N/C
 NBR OF ALLOCATIONS                1    NOT ACCOUNTED           0.000043    CLAIM RELEASE          0.000000       0           N/C
 SQL STMT - AVERAGE             2.00                                        PAGE LATCH             0.000000       0           N/C
 SQL STMT - TOTAL                  2    CP CPU SU                     15    NOTIFY MESSAGES        0.000000       0           N/C
 SUCC AUTH CHECK    YES                    AGENT                       15    GLOBAL CONTENTION      0.000000       0           N/C
                                           PAR.TASKS                    0    TCP/IP LOB             0.000000       0           N/C
                                         IIP CPU SU                     7    TOTAL CL8 SUSPENS.     0.001232       1      0.001232
                                         DB2 ENTRY/EXIT                 4

 TESTSQCZ                  ELAPSED TIME    EVENTS    TESTSQCZ                  ELAPSED TIME    EVENTS
 ------------------------- ------------  --------    ------------------------- ------------  --------
 GLOBAL CONTENTION L-LOCKS      0.000000         0    GLOBAL CONTENTION P-LOCKS      0.000000         0
  PARENT (DB,TS,TAB,PART)       0.000000         0     PAGESET/PARTITION             0.000000         0
  CHILD  (PAGE,ROW)             0.000000         0     PAGE                          0.000000         0
  OTHER                         0.000000         0     OTHER                         0.000000         0
```

You can also use Accounting Report, which is a summary of Accounting Trace. For details, see *IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS Version 4.1.0: Report Reference*, SC18-9984.

Using performance databases

You can use the FILE subcommand to format DB2 Accounting records and write them to sequential data sets suitable for use by the DB2 LOAD utility. You can place Accounting data into the OMEGAMON XE for DB2 PE performance database. You can use the performance database to produce tailored reports using a reporting facility such as Query Management Facility (QMF).

The following record format types are available:

- General Accounting data, for example, SQL counters, times, and locking activity
- Buffer pool data
- DDF data
- Package data
- Group buffer pool data

The parallel records are contained in the originating record. The number of records in the output are as follows:

- One record for General Accounting data
- Separate records for each package executed
- Separate records for each buffer pool used
- Separate records for each group buffer pool used
- Separate records for each remote location participating in the distributed activity

DDL statements, LOAD controls, and descriptions of the Accounting File data sets and the fields contained can be found in the RKO2SAMP library. See Table 11-4.

Table 11-4 Members in the RKO2SAMP library

PDS member name	Description
DGOACFGE	DDL statement for accounting table 'GENERAL', named DB2PMFACCT_GENERAL as default
DGOACFPK	DDL statement for accounting table 'PACKAGE', named DB2PMFACCT_PROGRAM as default
DGOACFBU	DDL statement for accounting table 'BUFFER POOL', named DB2PMFACCT_BUFFER as default.
DGOACFGP	DDL statement for accounting table 'GBP', named DB2PMFACCT_GBUFFER as default
DGOACFDF	DDL statement for accounting table 'DDF', named DB2PMFACCT_DDF as default
DGOALFGE	LOAD control for accounting table 'GENERAL'
DGOALFPK	LOAD control for accounting table 'PACKAGE'
DGOALFBU	LOAD control for accounting table 'BUFFER POOL'
DGOALFGP	LOAD control for accounting table 'GBP'
DGOALFDF	LOAD control for accounting table 'DDF'
DGOABFGE	Description for accounting FILE 'GENERAL' data
DGOABFPK	Description for accounting FILE 'PACKAGE' data
DGOABFBU	Description for accounting FILE 'BUFFER POOL' data
DGOABFGP	Description for accounting FILE 'GBP' data
DGOABFDF	Description for accounting FILE 'DDF' data

Note: Be aware that sample LOAD controls contains the REPLACE clause. If you create several tables in one segmented table space, the data already loaded is deleted. To avoid this, use RESUME YES instead of REPLACE or create one table space for each table.

If the sample LOAD control statement contains the WHEN clause, the records not matching the condition are discarded to SYSDISC.

Analyzing package-level accounting

In most cases, when an application connects to DB2, issues several DML statements, COMMITs or ROLLBACKs, and terminates, probably only a few packages are used. This can be measured effectively enough by plan-level accounting (class 1, 2, and 3).

In some cases, however, plan-level accounting cannot measure application performance effectively. For example, let us consider that an IMS region whose plan is associated with a considerable number of packages and updates multiple tables runs for a week, and several copies of such a region run concurrently to increase total throughput.

In this case, we must measure application performance by using package-level accounting data instead of plan-level accounting. Package-level accounting provides:

- Statements issued
- Accumulated CPU time
- Accumulated suspension time.

From these metrics, we can figure out the package which issues statements the most frequently, rough estimate for CPU time per statement and suspension time per statement for each package.

These statistics can allow you to identify which package is the bottleneck and to concentrate your business resources on analyzing it.

11.6.2 Accounting class 10

You might sometimes need more details for package activity. Accounting class 10 provides SQL activity, locking activity, and buffer pool activity for a package at the cost of extra overhead. Accounting data from class 10 can be reported in an accounting trace report and also written to sequential data sets for performance databases.

Reporting accounting trace

You can see the additional sections in your accounting traces if accounting class 10 is activated and IFCID 239 records are collected.

Example 11-7 shows package SQL activity.

Example 11-7 Package SQL activity

```
TESTSQCZ                  TOTAL
------------------     --------
SELECT                         0
INSERT                         1
UPDATE                         0
DELETE                         0

DESCRIBE                       0
PREPARE                        0
```

```
OPEN                          0
FETCH                         0
CLOSE                         0

LOCK TABLE                    0
CALL                          0
```

Example 11-8 shows package locking activity.

Example 11-8 Package locking activity

```
TESTSQCZ                   TOTAL
--------------------  --------
TIMEOUTS                      0
DEADLOCKS                     0
ESCAL.(SHARED)                0
ESCAL.(EXCLUS)                0
MAX PG/ROW LOCKS HELD         1
LOCK REQUEST                  3
UNLOCK REQUEST                1
QUERY REQUEST                 0
CHANGE REQUEST                0
OTHER REQUEST                 0
TOTAL SUSPENSIONS             0
 LOCK SUSPENS                 0
 IRLM LATCH SUSPENS           0
 OTHER SUSPENS                0
```

Example 11-9 shows package buffer pool activity.

Example 11-9 Package buffer pool activity

```
TESTSQCZ                   TOTAL
------------------    --------
BPOOL HIT RATIO (%)         100
GETPAGES                      7
BUFFER UPDATES                3
SYNCHRONOUS WRITE             0
SYNCHRONOUS READ              0
SEQ. PREFETCH REQS            0
LIST PREFETCH REQS            0
DYN. PREFETCH REQS            0
PAGES READ ASYNCHR.           0
```

11.6.3 Using tailored performance trace

Monitoring all packages using class 10 all the time is not practical. You can do so for some packages for some time interval. Example 11-10 shows how to start a performance trace filtered by PKGCOL, PKGPROG, CORRID, PLAN, and so on. For the other criteria that you can use, see *DB2 Version 9.1 for z/OS Command Reference: Chapter 81. -START TRACE (DB2)*, SC18-9844.

Example 11-10 Start tailored performance trace

```
START TRACE(PERFM) CLASS(30) IFCID(239) PKGCOL/PKGPROG/CORRID/PLAN
```

Tailored performance trace can be useful for many other purposes.

For example, when you need to perform ad hoc analysis about dynamic SQL statements, especially from data warehouse applications, a trace as shown in Example 11-11 can be useful. From DB2 9 for z/OS, a newly introduced object ROLE can be also used to filter a performance trace.

Example 11-11 Trace the access path, statement text, and sort activity of SQL issued by data warehousing users

```
-- Filter by PLANNAME(DISTSERV) and AUTHID of data warehouse applications.
-- 'DISTSERV' is a default PLANNAME for the distributed applicatoins.
START TRACE(PERFM) CLASS(30) IFCID(22 63 96) PLAN(DISTSERV) AUTHID(xxx) DEST(SMF)

-- From DB2 9 for z/OS
-- If you associate the trusted context from datawarehouse applications
-- with the role(DWUSER), You can also filter by ROLE(DWUSER).
START TRACE(PERFM) CLASS(30) IFCID(22 63 96) ROLE(DWUSER) DEST(SMF)
```

IFCIDs used in the traces are defined as follows:

- IFCID 022 - Minibind

  The minibind record shows information about mini plans, which are generated by the optimizer at bind and SQL prepare time. One mini plan is generated for each table and for each subselect block in the query. This means that if your query uses subqueries, more than one mini plan record is written.

- IFCID 063 - SQL Statement

  The SQL statement being processed. SQL text longer than 5000 characters is truncated. Host variables are represented as :H.

- IFCDI 317 - SQL Statement

  The SQL statement being processed. This IFCID contains the entire SQL statement text.

- IFCID 096 - Sort End

  Sort end record shows information about sort activities and includes the number of records sorted (RECNO), the sort data are size in bytes (AREA), the sort key size in bytes (KEYSZ), the number of sort column (SORT COLUMNS), the number of sort keys (SORT KEYS).

For details about dynamic SQL analysis, see Chapter 12 Monitoring, tracing, and explaining dynamic SQL of *Squeezing the Most Out of Dynamic SQL with DB2 for z/OS and OS/390,* SG24-6418.

11.6.4 Auditing trace

DB2 for z/OS IFI supports not only accounting data of applications but also audit data for applications.

See Table 11-5 for a description of the audit classes. For details about audit class, see *Chapter 9. Auditing access to DB2* on *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840.

Table 11-5 Description of audit classes

Audit class	Activated IFCIDs	Events that are traced
1	140	Access attempts denied due to inadequate authorization
2	141	Explicit GRANT and REVOKE
3	142	CREATE, ALTER, and DROP operations against audited tables
4	143	First change of audited object
5	144	First read of audited object
6	145	Bind time information about SQL statements that involve audited objects
7	55,83,87,169,319	Assignment or change of authorization ID
8	23,24,25,219,220	Utilities
9	146	Various types of records that are written to IFCID 146 by the IFI WRITE function
10	269	CREATE and ALTER TRUSTED CONTEXT statements, establish trusted connection information and switch user information

Collecting audit data

To collect audit data, you need to start audit trace. From DB2 9 for z/OS, a newly introduced object ROLE can be used to filter audit trace. Example 11-12 shows how to start an audit trace with including or excluding specific roles.

Example 11-12 Start trace with including or excluding roles

```
-- Start audit trace with including role
START TRACE(AUDIT) CLASS(...) ROLE(...)

-- Start audit trace with excluding role
START TRACE(AUDIT) CLASS(...) XROLE(...)
```

You also need to enable tables to be auditable. To turn on audit for the existing table, you can use ALTER TABLE statement as shown in Example 11-13.

Example 11-13 Alter table for auditing

```
ALTER TABLE <table_to_be_audited> AUDIT ALL/CHANGES
```

Reporting audit data

You can report audit data by using OMEGAMON PE. Example 11-14 shows sample usage of OMEGAMON PE for audit trace.

Example 11-14 Using OMEGAMON PE for audit trace

```
//DB2PE   EXEC PGM=DB2PE
//INPUTDD  DD  DSN=SMFDATA,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//JOBSUMDD DD  SYSOUT=*
```

```
//AUDTRC   DD  SYSOUT=*
//SYSIN    DD  *
AUDIT TRACE  DDNAME(AUDTRC)
  INCLUDE(CONNECT(SERVER))
EXEC
```

We provide sample audit trace along with the steps of the test JDBC application. The test JDBC application does the following actions:

1. Connects to database DB9A using AUTHID PAOLOR1 from 9.189.188.76. This connection is classified into TRUSTED CONTEXT SG247688 associated with the ROLE ITSOBANK.
2. Issues a DELETE statement. This is the first write to the audited table COMPANY_A.
3. Issues an INSERT statement.

Example 11-15 shows the audit trace from the test JDBC application.

Example 11-15 Audit trace sample

```
OPRIMAUTH CORRNAME CONNTYPE
 ORIGAUTH CORRNMBR INSTANCE
 PLANNAME CONNECT                TIMESTAMP   TYPE                                  DETAIL
 -------- -------- ------------ ----------- -------- ------------------------------------------------------------------------------------
 PAOLOR1  db2jcc_a DRDA         23:38:21.06 AUTHCHG  TYPE:               ESTABLISH TRUSTED CONTEXT        STATUS:            SUCCESS
 PAOLOR1  ppli     C350F798E892                      OBJECT OWNER:       AUTHID                           SQLCODE:                 0
 DISTSERV SERVER                                     SECURITY LABEL:
 REQLOC  :::9.189.188.76                             CONTEXT NAME:       SG247688
 ENDUSER :PAOLOR1                                    CONTEXT ROLE:       ITSOBANK
 WSNAME  :AB4906560                                  USER ROLE:
 TRANSACT:db2jcc_application                         PREV. SYSAUTHID:    PAOLOR1
                                                     REUSE AUTHID:
                                                     SERVAUTH NAME:
                                                     JOB NAME:
                                                     ENCRYPTION:         NONE
                                                     TCP/IP USED:        9.189.188.76
PAOLOR1  db2jcc_a DRDA         23:38:21.43 BIND     PACKAGE: DB9A.NULLID.SYSLH200.X'5359534C564C3031'
 PAOLOR1  ppli     C350F798E892                      TYPE: DELETE                    STMT#N/P     ISOLATION(CS)  KEEP UPD LOCKS:  NO
 DISTSERV SERVER                                     TEXT: DELETE FROM COMPANY_A
 REQLOC  :::9.189.188.76                             DATABASE: DSN00293          TABLE OBID:      3
 ENDUSER :PAOLOR1
 WSNAME  :AB4906560
 TRANSACT:db2jcc_application

 PAOLOR1  db2jcc_a DRDA         23:38:21.44 DML      TYPE    : 1ST WRITE
 PAOLOR1  ppli     C350F798E892                      DATABASE: DSN00293              TABLE OBID:      3
 DISTSERV SERVER                                     PAGESET : COMPANYR              LOG RBA   : X'000000000000'
 REQLOC  :::9.189.188.76
 ENDUSER :PAOLOR1
 WSNAME  :AB4906560
 TRANSACT:db2jcc_application

 PAOLOR1  db2jcc_a DRDA         23:38:21.78 BIND     PACKAGE: DB9A.NULLID.SYSLH200.X'5359534C564C3031'
 PAOLOR1  ppli     C350F798E892                      TYPE: INSERT                    STMT#N/P     ISOLATION(CS)  KEEP UPD LOCKS:  NO
 DISTSERV SERVER                                     TEXT: INSERT INTO COMPANY_A VALUES(5791, 'O''Brien', 38, 'Sales',
                                                           NULL, 18006.00)
 REQLOC  :::9.189.188.76                             DATABASE: DSN00293          TABLE OBID:      3
 ENDUSER :PAOLOR1
 WSNAME  :AB4906560
 TRANSACT:db2jcc_application
```

Part 4

Appendixes

In this part of the book, we put reference material not indispensable for the flow in the book, but good for reference.

We provide the following appendixes:

A

Useful queries

When working with packages, you often need to run queries against the DB2 catalog to manage your environment and troubleshoot any issues. In this appendix, we provide a set of queries that we think might be useful to you. They are meant to be a starting point and in most cases, need to be modified to suit the standards and policies in your environment.

This appendix provides the following queries:

- Identifying invalid or inoperative packages
- Showing the package list for a plan
- Showing all plans that contain a package
- Showing all packages that access a remote object
- Showing all authorizations for a set of collections
- Showing all packages that access all tables (CRUD matrix)
- Showing all trigger packages
- Showing packages dependent on an index
- Showing packages dependent on a table
- Displaying SYSPACKSTMT in EBCDIC
- Showing packages that have changed access paths
- Finding typical “bad” access paths
- Showing packages that use OPTHINTs
- Finding packages bound with EXPLAIN(NO)
- Finding obsolete PLAN_TABLE entries
- Cleaning up extended EXPLAIN tables
- Finding plans containing DBRMs
- Finding multiple DBRM members with differences
- Displaying CURRENTDATA usage for packages
- Showing packages dependent on an index

Note that in the queries that follow, to simplify formatting, we make the following assumptions:

- The queries are meant to be a starting point. They have specific references to our setup (for example, collection IDs, program names, and so on). You need to modify them to add your specific search criteria.
- We recommend an index on your PLAN_TABLE consisting of at least COLLID, PROGNAME, VERSION,QUERYNO and BIND_TIME
- Some of these queries might be resource intensive, based on the size of your tables and selection criteria.
- If you use the package stability feature, you need to account for multiple copies of the same version of the package, which are denoted by the column DTYPE is SYSPACKDEP. For details, see 10.13, “Package stability” on page 216.
- We assume that:
  - Collection ID is 18 bytes
  - Program name is 8 bytes
  - Location is 8 bytes
  - Owner is 8 bytes
  - Creator is 8 bytes
  - Version is 26 bytes
  - PDS name is 44 bytes
  - Grantor is 8 bytes
  - Grantee is 8 bytes
  - Schema is 8 bytes
  - Trigger name is 18 bytes
  - Index name is 18 bytes

A.1 Identifying invalid or inoperative packages

To list any invalid or inoperative packages, you can use the query in Example A-1.

See 8.5.2, “Invalid and inoperative packages” on page 169 and 7.6, “Inoperative packages” on page 157 for details.

Example: A-1 Invalid or inoperative packages

```
SELECT
 SUBSTR(COLLID,1,18)       AS COLLECTION
,SUBSTR(NAME,1,8)          AS PACKAGE
,HEX (CONTOKEN)            AS CONTOKEN
,SUBSTR(OWNER,1,8)         AS OWNER
,SUBSTR(CREATOR,1,8)       AS CREATOR
,VALID                     AS VALID
,OPERATIVE                 AS OPERATIVE
,EXPLAIN                   AS EXPLAIN
,SUBSTR(VERSION,1,26)      AS VERSION
,SUBSTR(PDSNAME,1,44)      AS PDSNAME
FROM SYSIBM.SYSPACKAGE
WHERE
(VALID = 'N' OR OPERATIVE = 'N')
ORDER BY COLLECTION , PACKAGE , VERSION
WITH UR;
```

Note that we list VALID = ‘N’ packages (instead of VALID <> ‘N’) because other values such as A and H are considered valid.

A.2 Showing the package list for a plan

To list all packages in a plan, you can use the query in Example A-2.

Example: A-2 All packages in plan

```
SELECT
 SUBSTR(PACKLIST.COLLID,1,18)  AS COLLECTION
,SUBSTR(PACKLIST.NAME,1,8)     AS PACKAGE
,SUBSTR(PACKLIST.PLANNAME,1,8) AS PLANNAME
,PACKLIST.SEQNO                AS SEQNO
,SUBSTR(PACKLIST.LOCATION,1,8) AS LOCATION
,SUBSTR(PLAN.CREATOR,1,8)      AS CREATOR
FROM SYSIBM.SYSPACKLIST PACKLIST
,    SYSIBM.SYSPLAN     PLAN
  WHERE PACKLIST.PLANNAME = PLAN.NAME
  AND PLAN.NAME = 'FINPBTCH'
ORDER BY COLLECTION,PACKAGE,PLANNAME,SEQNO
WITH UR;
```

A.3 Showing all plans that contain a package

To list all plans which contain a package, you can use the query in Example A-3.

Example: A-3 All plans that contain a package

```
SELECT
 SUBSTR(SYSPACK.COLLID,1,18)  AS COLLECTION
,SUBSTR(SYSPACK.NAME,1,8)     AS PACKAGE
,SUBSTR(SYSPACK.PLANNAME,1,8) AS PLANNAME
,SYSPACK.SEQNO                AS SEQNO
,SUBSTR(SYSPACK.LOCATION,1,8) AS LOCATION
,SUBSTR(SYSPLAN.CREATOR,1,8)  AS CREATOR
FROM SYSIBM.SYSPACKLIST SYSPACK
,    SYSIBM.SYSPLAN     SYSPLAN
WHERE SYSPACK.PLANNAME = SYSPLAN.NAME
  AND SYSPACK.LOCATION = ' '
  AND SYSPACK.COLLID   = 'FIN_TEST_BATCH'
  AND SYSPACK.NAME     = 'PGM03'
ORDER BY COLLECTION,PACKAGE,PLANNAME,SEQNO
WITH UR;
```

A.4 Showing all packages that access a remote object

To list all packages which access a remote object, you can use the query in Example A-4.

Example: A-4 Packages that access a remote object

```
SELECT
 SUBSTR(DEP.DCOLLID,1,18)   AS COLLECTION
,SUBSTR(DEP.DNAME,1,8)      AS PACKAGE
,HEX(DEP.DCONTOKEN)         AS CONTOKEN
,SUBSTR(DEP.BQUALIFIER,1,8) AS ALIAS_QUALIFIER
,SUBSTR(DEP.BNAME,1,18)     AS ALIAS_TABLENAME
,SUBSTR(TAB.LOCATION,1,8)   AS TABLE_LOCATION
,SUBSTR(TAB.TBCREATOR,1,8)  AS TABLE_CREATOR
,SUBSTR(TAB.TBNAME,1,18)    AS TABLE_NAME
FROM SYSIBM.SYSPACKDEP DEP
,    SYSIBM.SYSTABLES  TAB
WHERE DEP.BQUALIFIER = TAB.CREATOR
  AND DEP.BNAME      = TAB.NAME
  AND DEP.BTYPE      = 'A'
  AND TAB.TYPE       = 'A'
  AND TAB.LOCATION  <> ' '
ORDER BY COLLECTION,PACKAGE,CONTOKEN,ALIAS_QUALIFIER,ALIAS_TABLENAME
WITH UR;
```

The foregoing query specifically looks for a non-blank location that denotes an alias pointing to a remote object.

A.5 Showing all authorizations for a set of collections

To list all authorizations to manage a package, you can use the query in Example A-5.

Example: A-5 All authorizations for a package

```
SELECT
 SUBSTR(COLLID,1,8)    AS COLLECTION
,SUBSTR(NAME,1,8)      AS PACKAGE
,SUBSTR(GRANTOR,1,8)   AS GRANTOR
,SUBSTR(GRANTEE,1,8)   AS GRANTEE
,BINDAUTH              AS BINDAUTH
,COPYAUTH              AS COPYAUTH
,EXECUTEAUTH           AS EXECUTEAUTH
,TIMESTAMP             AS TIMESTAMP
FROM SYSIBM.SYSPACKAUTH
WHERE GRANTEETYPE = ' '
  AND LOCATION    = ' '
  AND COLLID = ‘FIN_TEST_BATCH'
  AND NAME = ‘PGM03’
ORDER BY COLLECTION,PACKAGE,GRANTOR,GRANTEE,TIMESTAMP
WITH UR;
```

A.6 Showing all packages that access all tables (CRUD matrix)

To list all packages that access a table and their intent (Create-Read-Update-Delete), you can use the query in Example A-6.

Example: A-6 Access intent (CRUD matrix) for all packages accessing a table

```
SELECT DISTINCT
 SUBSTR(TBAUTH.TCREATOR,1,8) AS TABLE_CREATOR
,SUBSTR(TBAUTH.TTNAME,1,18)  AS TABLE_NAME
,SUBSTR(TBAUTH.GRANTEE,1,8)  AS PACKAGE
,HEX(CONTOKEN)               AS CONTOKEN
,CASE WHEN TBAUTH.INSERTAUTH = 'Y' THEN 'C' ELSE ' ' END AS CREATE_AUTH
,CASE WHEN TBAUTH.SELECTAUTH = 'Y' THEN 'R' ELSE ' ' END AS READ_AUTH
,CASE WHEN TBAUTH.UPDATEAUTH = 'Y' THEN 'U' ELSE ' ' END AS UPDATE_AUTH
,CASE WHEN TBAUTH.DELETEAUTH = 'Y' THEN 'D' ELSE ' ' END AS DELETE_AUTH
FROM SYSIBM.SYSTABAUTH TBAUTH
,    SYSIBM.SYSTABLES  TB
WHERE TBAUTH.TCREATOR  = TB.CREATOR
AND TBAUTH.TTNAME      = TB.NAME
AND TBAUTH.LOCATION    = ' '
AND TBAUTH.GRANTEETYPE = 'P'
AND TB.TYPE         IN ('T','V')
AND TB.NAME            = 'EMP'
AND TB.CREATOR         = 'DSN8910'
ORDER BY TABLE_CREATOR,TABLE_NAME,PACKAGE,CONTOKEN
WITH UR;
```

A.7 Showing all trigger packages

To list all trigger packages, you can use the query in Example A-7.

Example: A-7 All trigger packages

```
SELECT
 SUBSTR(SCHEMA,1,18) AS SCHEMA,
 SUBSTR(NAME,1,18)   AS TRIGGER_PACKAGE
FROM SYSIBM.SYSTRIGGERS
ORDER BY SCHEMA,TRIGGER_PACKAGE
WITH UR;
```

A.8 Showing packages dependent on an index

Before planning to drop an index, you need to determine all dependent packages using that index. Example A-8 shows such packages. An alternative would be to find the invalid packages after dropping and recreating the index.

Note: You need to be aware of dynamic SQL using an index before dropping it.

Example: A-8 Packages dependent on index

```
SELECT
 SUBSTR(BQUALIFIER,1,8) AS INDEX_CREATOR
,SUBSTR(BNAME,1,18)     AS INDEX_NAME
,SUBSTR(DCOLLID,1,18)   AS COLLECTION
,SUBSTR(DNAME,1,8)      AS PACKAGE
,HEX(DCONTOKEN)         AS CONTOKEN
FROM SYSIBM.SYSPACKDEP
WHERE BQUALIFIER = 'PAOLOR3'
  AND BNAME = 'INDEX1'
  AND BTYPE = 'I'
ORDER BY COLLECTION,PACKAGE,CONTOKEN
WITH UR;
```

A.9 Showing packages dependent on a table

A query to get packages dependent on a table is listed in Example A-9

Example: A-9 Packages dependent on table

```
SELECT
SUBSTR(BQUALIFIER,1,8) AS TABLE_CREATOR
,SUBSTR(BNAME,1,20) AS TABLE_NAME
,SUBSTR(DCOLLID,1,18) AS COLLECTION
,SUBSTR(DNAME,1,8) AS PACKAGE
,HEX(DCONTOKEN) AS CONTOKEN
FROM SYSIBM.SYSPACKDEP
```

```
WHERE  BQUALIFIER = 'PAOLOR3'
 AND BNAME = 'EMP'
 AND BTYPE = 'T'
ORDER BY COLLECTION,PACKAGE,CONTOKEN
WITH UR;
```

A.10 Displaying CONTOKEN

The purpose of this query is to show the CONTOKEN and generate the LOADLIB CONTOKEN which can be used to confirm that the right version is being used. For our version of COBOL, the LOADLIB CONTOKEN is obtained by switching the first bytes with the second 4 bytes of the CONTOKEN in the package. This is shown in Example A-10.

Disclaimer: Be aware that what happens in the load library and the CONTOKEN is fully dependent upon the source language generated, the compiler language generation from that source, and the linkage editor. There is no committed external description and it is almost sure that it might change in the future.

Note: You need to be aware of dynamic SQL using an index before dropping it.

Example: A-10 CONTOKEN for a package

```
SELECT
 SUBSTR(NAME,1,8)            AS PACKAGE
,SUBSTR(COLLID,1,18)         AS COLLECTION
,HEX(CONTOKEN)               AS DBRM_TOKEN
,SUBSTR(HEX(CONTOKEN),9,8)
   CONCAT
 SUBSTR(HEX(CONTOKEN),1,8)   AS LOADLIB_TOKEN
,SUBSTR(VERSION,1,26)        AS VERSION
,SUBSTR(PDSNAME,1,44)        AS PDSNAME
FROM SYSIBM.SYSPACKAGE
WHERE NAME LIKE 'PGM%'
ORDER BY NAME, COLLECTION, VERSION
WITH UR;
```

A.11 Displaying SYSPACKSTMT in EBCDIC

To display the contents of SYSPACKSTMT IN EBCDIC, you can use the query in Example A-11. We use the CAST function to convert the text in Unicode from CCSID 1208 for display purposes. This query is quite simple and produces a somewhat incomplete but fully identifiable output. Sometimes the actual statement part of the SYSPACKSTMT STMT text string is made up of the host variable information is and would need more logic to be handled properly.

Note: IBM Bind Manager and various vendor products provide the ability to parse the text and present it in a meaningful way. See Appendix B.3.1, “IBM DB2 Bind Manager for z/OS” on page 282.

Example: A-11 Statement text from SYSPACKSTMT

```
SELECT
 SUBSTR(NAME,1,8) AS PACKAGE
,SECTNO           AS SECTNO
,STMTNO           AS STMTNO
,SEQNO            AS SEQNO
,SUBSTR(CAST(CAST(STMT AS VARCHAR(3500)CCSID 1208)
          AS VARCHAR(3500) CCSID EBCDIC),9,3492) AS SQL_STATEMENT
 FROM SYSIBM.SYSPACKSTMT
 WHERE
    NOT (STMTNO=0 AND SEQNO=0 AND SECTNO=0)
 AND NAME = 'PGM01'
ORDER BY NAME,SECTNO,STMTNO,SEQNO
WITH UR;
```

A.12 Showing packages that have changed access paths

After a BIND REPLACE or REBIND operation, it is possible for the access path for some of the queries to change. To detect any queries whose access paths have changed, you can use the query in Example A-12.

Ideally, you need to compare the actual access paths of the packages. In practice, this task is not easy because the QUERYNO can change from the previous copy of the package, as code changes (including non-SQL statements). A comparison that looks for a 100% match is likely to produce a large volume of data to review. making the usefulness of such a query questionable. For this reason, we present an alternative method.

We compare the total cost of each copy of the package from the DSN_STATEMNT_TABLE. If the costs are different, they are reported. Other options you can consider would be to report on the number of rows per package in the PLAN_TABLE for each copy.

Keep in mind that it is not possible to make a blanket statement as to whether or not these have improved or regressed, because only a close examination for the SQL along with the expected volumes reveals the impact. You can add your own heuristic criteria to reduce the output generated by this query. In rare cases, an actual access path change could be missed because it results in the same total cost.

Some vendor tools, such as IBM Path Checker go deeper than the simplistic analysis shown here and provide advice on which access path differences are “significant” and worth worrying about. See B.3.2, “IBM DB2 Path Checker for z/OS” on page 282 for details.

Example: A-12 Packages that have changed access paths

```
WITH MAXTM AS
 (SELECT COLLID
        ,PROGNAME
        ,MAX(EXPLAIN_TIME) AS EXPLAIN_TIME
  FROM PAOLOR1.DSN_STATEMNT_TABLE
  GROUP BY COLLID,PROGNAME),
PRVTM AS
 (SELECT COLLID
        ,PROGNAME
        ,MAX(EXPLAIN_TIME) AS EXPLAIN_TIME
```

```
  FROM PAOLOR1.DSN_STATEMNT_TABLE TEMPMAX
  WHERE  EXPLAIN_TIME <
         (SELECT EXPLAIN_TIME FROM MAXTM
          WHERE TEMPMAX.COLLID   = MAXTM.COLLID
            AND TEMPMAX.PROGNAME = MAXTM.PROGNAME
            AND TEMPMAX.EXPLAIN_TIME < MAXTM.EXPLAIN_TIME)
  GROUP BY COLLID,PROGNAME),
CUR AS
 (SELECT
    CUR.COLLID
   ,CUR.PROGNAME
   ,CUR.EXPLAIN_TIME
   ,SUM(CUR.TOTAL_COST) AS CUR_TOTAL_COST
  FROM MAXTM
      ,PAOLOR1.DSN_STATEMNT_TABLE CUR
  WHERE MAXTM.COLLID       = CUR.COLLID
    AND MAXTM.PROGNAME     = CUR.PROGNAME
    AND MAXTM.EXPLAIN_TIME = CUR.EXPLAIN_TIME
  GROUP BY CUR.COLLID,CUR.PROGNAME,CUR.EXPLAIN_TIME),
PRV AS
 (SELECT
    PRV.COLLID
   ,PRV.PROGNAME
   ,PRV.EXPLAIN_TIME
   ,SUM(PRV.TOTAL_COST) AS PRV_TOTAL_COST
  FROM PRVTM
      ,PAOLOR1.DSN_STATEMNT_TABLE PRV
  WHERE PRVTM.COLLID = PRV.COLLID
    AND PRVTM.PROGNAME = PRV.PROGNAME
    AND PRVTM.EXPLAIN_TIME = PRV.EXPLAIN_TIME
  GROUP BY PRV.COLLID,PRV.PROGNAME,PRV.EXPLAIN_TIME)

SELECT
 SUBSTR(CUR.COLLID,1,8)         AS COLLECTION
,SUBSTR(CUR.PROGNAME,1,8)       AS PACKAGE
,CAST(PRV_TOTAL_COST AS DEC(9)) AS PRV_TOTAL_COST
,CAST(CUR_TOTAL_COST AS DEC(9)) AS CUR_TOTAL_COST
, CAST(100.00 * ((CUR_TOTAL_COST - PRV_TOTAL_COST) / PRV_TOTAL_COST)
      AS DEC(6))                AS PCT_INCREASE
FROM CUR , PRV
WHERE CUR.COLLID = PRV.COLLID
  AND CUR.PROGNAME = PRV.PROGNAME
  AND CUR.CUR_TOTAL_COST <> PRV.PRV_TOTAL_COST
ORDER BY PCT_INCREASE DESC
```

A.13 Finding typical “bad” access paths

Various definitions of a “bad” access path are possible based on your environment and on your service level agreements. In our example, we arbitrarily define the following exceptions as worth reporting.

- A table space scan of a table with more than 100,000 rows
- A non-matching index scan against an index with more than 10,000 leaf pages

- A LIST PREFETCH access
- A multiple-index access
- A hybrid join

To display the packages that have such typical “bad” access path, you can use the query in Example A-13.

Example: A-13 Packages with “bad” access paths

```
SELECT
 SUBSTR(PLNTBL.COLLID,1,18)           AS COLLECTION
,SUBSTR(PLNTBL.PROGNAME,1,8)          AS PACKAGE
,SUBSTR(PLNTBL.VERSION,1,26)          AS VERSION
,CAST(SYSTAB.CARDF AS INTEGER)        AS NUMBER_OF_ROWS
,CAST(COALESCE(IX.NLEAF,0)AS INTEGER) AS LEAF_PAGES
,PLNTBL.QUERYNO                       AS QUERYNO
,PLNTBL.QBLOCKNO                      AS QBLOCKNO
,PLNTBL.PLANNO                        AS PLANNO
,PLNTBL.METHOD                        AS METHOD
,SUBSTR(PLNTBL.CREATOR,1,8)           AS TABLE_CREATOR
,SUBSTR(PLNTBL.TNAME,1,18)            AS TABLE_NAME
,PLNTBL.TABNO                         AS TABNO
,PLNTBL.ACCESSTYPE                    AS ACCESSTYPE
,PLNTBL.MATCHCOLS                     AS MATCHCOLS
,SUBSTR(PLNTBL.ACCESSCREATOR,1,8)     AS INDEX_CREATOR
,SUBSTR(PLNTBL.ACCESSNAME,1,18)       AS INDEX_NAME
,PLNTBL.PREFETCH                      AS PREFETCH


FROM PLAN_TABLE       PLNTBL INNER JOIN
     SYSIBM.SYSTABLES SYSTAB
ON    PLNTBL.CREATOR = SYSTAB.CREATOR
  AND PLNTBL.TNAME   = SYSTAB.NAME

LEFT OUTER JOIN (SELECT C.CREATOR,C.NAME,C.NLEAF
FROM  SYSIBM.SYSINDEXES C
WHERE C.NLEAF > 10000)  AS IX
ON   PLNTBL.ACCESSCREATOR = IX.CREATOR
 AND PLNTBL.ACCESSNAME    = IX.NAME
WHERE
    (
       (PLNTBL.ACCESSTYPE = 'R' AND SYSTAB.CARDF > 100000)
   OR  (PLNTBL.ACCESSTYPE = 'I'
    AND PLNTBL.MATCHCOLS  = 0
    AND IX.NLEAF           > 10000)
   OR  (PLNTBL.PREFETCH   = 'L')
   OR  (PLNTBL.ACCESSTYPE = 'M')
   OR  (PLNTBL.METHOD     = 4)
    )
WITH UR;
```

A.14 Showing packages that use OPTHINTs

To display the packages with an OPTHINT, you can use the query in Example A-14. We show those packages where the hint is proposed (might or might not be applied) and also those where the hint is actually applied. If a hint was proposed on two statements but applied to only one, we do not cover this scenario (we need to deal with it at the statement level from the PLAN_TABLE).

Example: A-14 Packages using OPTHINTs

```
SELECT 'HINT PROPOSED'      AS HINT_TYPE
,SUBSTR(COLLID,1,18)        AS COLLECTION
,SUBSTR(PROGNAME,1,8)       AS PACKAGE
,SUBSTR(VERSION,1,26)       AS VERSION
,SUBSTR(OPTHINT,1,20)       AS OPTHINT
,SUBSTR(HINT_USED,1,20)     AS HINT_USED
FROM PLAN_TABLE
  WHERE OPTHINT <> ' '
UNION ALL
SELECT DISTINCT 'HINT USED' AS HINT_TYPE
,SUBSTR(COLLID,1,18)        AS COLLECTION
,SUBSTR(PROGNAME,1,8)       AS PACKAGE
,SUBSTR(VERSION,1,26)       AS VERSION
,SUBSTR(OPTHINT,1,20)       AS OPTHINT
,SUBSTR(HINT_USED,1,20)     AS HINT_USED
FROM PLAN_TABLE
WHERE HINT_USED <> ''

ORDER BY COLLECTION,PACKAGE,VERSION
WITH UR;
```

A.15 Finding packages bound with EXPLAIN(NO)

To display the packages which have been bound with EXPLAIN(NO), you can use the query in Example A-15.

Example: A-15 Packages without an EXPLAIN

```
SELECT
 SUBSTR(COLLID,1,18)  AS COLLECTION
,SUBSTR(NAME,1,8)     AS PACKAGE
,SUBSTR(VERSION,1,26) AS VERSION
,EXPLAIN              AS EXPLAIN
FROM SYSIBM.SYSPACKAGE
WHERE EXPLAIN = 'N'
ORDER BY COLLECTION, PACKAGE, VERSION
WITH UR;
```

A.16 Finding obsolete PLAN_TABLE entries

To identify the obsolete PLAN_TABLE entries, you can use the query in Example A-16. You can modify it adding a DELETE in front of it or a CREATE TABLE to insert the results for later examination.

In this query, we list the package version which might be purged. The criteria we use to decide what needs to be kept are:

- All entries for packages bound in the last 90 days must be kept.
- All entries for queries which were hinted must be kept.
- All entries for the last 5 versions must be kept.

Example: A-16 SQL to find obsolete PLAN_TABLE entries

```
SELECT DISTINCT
 SUBSTR(PKGS.COLLID,1,18)   AS COLLECTION
,SUBSTR(PKGS.PROGNAME,1,8)  AS PACKAGE
,PKGS.BIND_TIME             AS BIND_TIME
,PKGS.OPTHINT               AS OPTHINT
FROM PAOLOR1.PLAN_TABLE PKGS
-- WHERE 90 REPRESENTS THE MINIMUM NUMBER OF DAYS TO KEEP
WHERE DATE(PKGS.BIND_TIME) < DATE(CURRENT TIMESTAMP - 90 DAYS)
  AND PKGS.OPTHINT = ''
-- WHERE 5 REPRESENTS THE MINIMUM NUMBER OF VERSIONS TO KEEP
  AND 5 <= (
           SELECT COUNT(DISTINCT CNT.BIND_TIME)
           FROM PAOLOR1.PLAN_TABLE CNT
           WHERE CNT.COLLID    = PKGS.COLLID
             AND CNT.PROGNAME  = PKGS.PROGNAME
             AND CNT.BIND_TIME > PKGS.BIND_TIME
           )
WITH UR;
```

Note that we make no attempt to preserve ALL rows for a package with OPTHINTs, only those queries which had an OPTHINT.

A few other things should be considered:

- If you are using Package Stability (see 10.13, “Package stability” on page 216 for details), you should keep all EXPLAIN information for up to three copies of the package: original, previous, and current. Unfortunately, while DB2 records the dependencies in SYSPACKDEP, there is no externalized means of detecting which copies are part of package stability whose EXPLAIN information should be retained. The easiest means of retaining this information is to exclude such packages from the foregoing list. You need to modify the foregoing query to implement this.
- It might also be advisable to keep the access path for the last version bound under the prior version of DB2. For example if you are running DB2 9, you should keep the last access path in DB2 V8. SYSPACKAGE contains a column IBMREQUIRED that indicates the version of DB2 in which the package was bound. For example, a value of L indicates a V8 package and a value of M indicates it was bound in DB2 9. You can modify the query as follows:
  - Include an INNER JOIN to SYSPACKAGE.
  - Exclude rows for the package with the highest BIND_TIME for the packages with IBMREQUIRED = L (to keep last V8 rows).

The PLANMGMT(EXTENDED) option saves the copy of the package before the version upgrade as the original copy and you can switch between the current and previous copy without affecting the original. See 10.13, “Package stability” on page 216 for details.

We have not shown this in the foregoing query because we think it is of limited use to the general audience and might impact performance of the query.

- If you are using the extended EXPLAIN tables (see C.2.2, “Extended EXPLAIN tables” on page 306 for details), you should coordinate the purge of the base PLAN_TABLE with these additional tables. These tables cannot be tied via a DB2 Referential Integrity (with the DELETE CASCADE option) because BIND_TIME is not unique on the PLAN_TABLE and a Primary Key is therefore not possible. We suggest that you create a trigger on the PLAN_TABLE to accomplish this. This is discussed further in 8.4, “Package and PLAN_TABLE maintenance” on page 166 and an example of the trigger is provided in A.17, “Cleaning up extended EXPLAIN tables” on page 271.

A.17 Cleaning up extended EXPLAIN tables

As discussed in A.16, “Finding obsolete PLAN_TABLE entries”, one of the ways to clean up the extended EXPLAIN tables is to create a trigger that automatically deletes the extended EXPLAIN tables. This is shown in Example A-17. Purging the data in these tables on a periodic basis (for example, older than 3 months) with statements with no-EXPLAIN data (if the statements get rebound on a similar schedule) might also be an acceptable alternative to implement.

Important: Some of these tables are linked via COLLID, PROGNAME and EXPLAIN_TIME while others are linked only by QUERYNO and EXPLAIN_TIME. The implication of this is that you might (in extremely rare cases) delete rows in the extended EXPLAIN tables that are not related to this package.

Example: A-17 Cleaning up extended EXPLAIN tables

```
-- ------------------------------------------------------------
--   We USE # AS THE SQL TERMINATOR IN SPUFI IN THIS EXAMPLE
-- ------------------------------------------------------------
CREATE TRIGGER PURGE_EXTENDED
  AFTER DELETE ON PLAN_TABLE
  REFERENCING  OLD_TABLE AS OT
  FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
------------------------------------------------------------
--   DSN_FUNCTION_TABLE
------------------------------------------------------------
DELETE FROM DSN_FUNCTION_TABLE
   WHERE  (COLLID,PROGNAME,EXPLAIN_TIME) IN
    (SELECT COLLID,PROGNAME,BIND_TIME FROM OT) ;
------------------------------------------------------------
-- DSN_STATEMNT_TABLE
------------------------------------------------------------
DELETE FROM DSN_STATEMNT_TABLE
   WHERE  (COLLID,PROGNAME,EXPLAIN_TIME) IN
    (SELECT COLLID,PROGNAME,BIND_TIME FROM OT) ;
------------------------------------------------------------
-- DSN_DETCOST_TABLE
```

```
-------------------------------------------------------------
DELETE FROM DSN_DETCOST_TABLE
   WHERE  (PROGNAME,QUERYNO,EXPLAIN_TIME) IN
    (SELECT PROGNAME,QUERYNO,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_FILTER_TABLE
-------------------------------------------------------------
DELETE FROM DSN_FILTER_TABLE
   WHERE  (COLLID,PROGNAME,EXPLAIN_TIME) IN
    (SELECT COLLID,PROGNAME,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_PGRANGE_TABLE
-------------------------------------------------------------
DELETE FROM DSN_PGRANGE_TABLE
   WHERE  (QUERYNO,EXPLAIN_TIME) IN
    (SELECT QUERYNO,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_PGROUP_TABLE
-------------------------------------------------------------
DELETE FROM DSN_PGROUP_TABLE
   WHERE  (COLLID,PROGNAME,EXPLAIN_TIME) IN
    (SELECT COLLID,PROGNAME,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_PREDICAT_TABLE
-------------------------------------------------------------
DELETE FROM DSN_PREDICAT_TABLE
   WHERE  (PROGNAME,QUERYNO,EXPLAIN_TIME) IN
    (SELECT PROGNAME,QUERYNO,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_PTASK_TABLE
-------------------------------------------------------------
DELETE FROM DSN_PTASK_TABLE
   WHERE  (PROGNAME,QUERYNO,EXPLAIN_TIME) IN
    (SELECT PROGNAME,QUERYNO,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_PREDICAT_TABLE
-------------------------------------------------------------
DELETE FROM DSN_PREDICAT_TABLE
   WHERE  (PROGNAME,QUERYNO,EXPLAIN_TIME) IN
    (SELECT PROGNAME,QUERYNO,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_SORT_TABLE
-------------------------------------------------------------
DELETE FROM DSN_SORT_TABLE
   WHERE  (PROGNAME,QUERYNO,EXPLAIN_TIME) IN
    (SELECT PROGNAME,QUERYNO,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_SORTKEY_TABLE
-------------------------------------------------------------
DELETE FROM DSN_SORTKEY_TABLE
   WHERE  (COLLID,PROGNAME,EXPLAIN_TIME) IN
    (SELECT COLLID,PROGNAME,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_STRUCT_TABLE
-------------------------------------------------------------
```

```
DELETE FROM DSN_STRUCT_TABLE
   WHERE  (PROGNAME,QUERYNO,EXPLAIN_TIME) IN
    (SELECT PROGNAME,QUERYNO,BIND_TIME FROM OT) ;
-------------------------------------------------------------
-- DSN_VIEWREF_TABLE
-------------------------------------------------------------
DELETE FROM DSN_VIEWREF_TABLE
   WHERE  (COLLID,PROGNAME,EXPLAIN_TIME) IN
    (SELECT COLLID,PROGNAME,BIND_TIME FROM OT) ;

END#
```

A.18 Finding packages with too many versions

As discussed in 8.2, “Package proliferation” on page 162, it is important to periodically clean up older versions of packages. We suggest that you keep all versions which are recent. We also suggest you keep a minimum number of versions. In this example we use three months for the date and three copies for the number of versions to keep.

To display the packages which have too many versions, you can use the query in Example A-18.

Example: A-18 Packages with “too many” versions

```
SELECT
 SUBSTR(SYSPACK.COLLID,1,18)  AS COLLECTION
,SUBSTR(SYSPACK.NAME,1,8)     AS PACKAGE
,SUBSTR(SYSPACK.VERSION,1,26) AS VERSION
,BINDTIME                     AS BINDTIME
FROM SYSIBM.SYSPACKAGE SYSPACK
WHERE EXISTS
    (SELECT 1
       FROM SYSIBM.SYSPACKAGE SYSPACK1
       WHERE SYSPACK.LOCATION   = SYSPACK1.LOCATION
         AND SYSPACK.COLLID     = SYSPACK1.COLLID
         AND SYSPACK.NAME       = SYSPACK1.NAME
         AND SYSPACK1.BINDTIME < CURRENT TIMESTAMP - 3 MONTHS
      )
  AND EXISTS
    (SELECT SYSPACK2.COLLID, SYSPACK2.NAME, COUNT(*)
     FROM SYSIBM.SYSPACKAGE SYSPACK2
     WHERE SYSPACK.LOCATION = SYSPACK2.LOCATION
       AND SYSPACK.COLLID   = SYSPACK2.COLLID
       AND SYSPACK.NAME     = SYSPACK2.NAME
     GROUP BY SYSPACK2.COLLID, SYSPACK2.NAME
     HAVING COUNT(*) > 3
    )
ORDER BY COLLECTION, PACKAGE, BINDTIME
WITH UR;
```

A.19 Finding plans containing DBRMs

To display such packages, you can use the query in Example A-19. As discussed in 4.3, “Modifying existing plans to use packages” on page 52, you should plan to convert such plans as soon as possible.

Example: A-19 Plans containing DBRMs

```
SELECT PLNAME AS PLANNAME,
       NAME   AS DBRMNAME
FROM   SYSIBM.SYSDBRM
ORDER BY PLANNAME,DBRMNAME
WITH UR;
```

A.20 Finding multiple DBRM members with differences

This topic relates to DBRM-to-package conversion. When converting, if multiple “versions” (that is, CONTOKEN or other precompile parameters) of the DBRM exist, unpredictable results are possible as discussed in Chapter 8, “Administration of packages” on page 161.

To display such DBRMs, you can use the query in Example A-20.

Example: A-20 Multiple DBRMs with differences

```
SELECT DBRM1.NAME,DBRM1.PLNAME
FROM SYSIBM.SYSDBRM DBRM1
,    SYSIBM.SYSDBRM DBRM2
WHERE (DBRM1.NAME           = DBRM2.NAME
  AND  DBRM1.PLNAME        <> DBRM2.PLNAME)

  AND (DBRM1.TIMESTAMP     <> DBRM2.TIMESTAMP
   OR  DBRM1.PDSNAME       <> DBRM2.PDSNAME
   OR  DBRM1.PLCREATOR     <> DBRM2.PLCREATOR
   OR  DBRM1.PRECOMPTIME   <> DBRM2.PRECOMPTIME
   OR  DBRM1.PRECOMPDATE   <> DBRM2.PRECOMPDATE
   OR  DBRM1.QUOTE         <> DBRM2.QUOTE
   OR  DBRM1.COMMA         <> DBRM2.COMMA
   OR  DBRM1.HOSTLANG      <> DBRM2.HOSTLANG
   OR  DBRM1.IBMREQD       <> DBRM2.IBMREQD
   OR  DBRM1.CHARSET       <> DBRM2.CHARSET
   OR  DBRM1.MIXED         <> DBRM2.MIXED
   OR  DBRM1.DEC31         <> DBRM2.DEC31
   OR  DBRM1.VERSION       <> DBRM2.VERSION
   OR  DBRM1.PRECOMPTS     <> DBRM2.PRECOMPTS
   OR  DBRM1.PLCREATORTYPE <> DBRM2.PLCREATORTYPE
   OR  DBRM1.RELCREATED    <> DBRM2.RELCREATED)
ORDER BY DBRM1.NAME,DBRM1.PLNAME
WITH UR;
```

A.21 Displaying CURRENTDATA usage for packages

The value of the parameter CURRENTDATA used during a package bind is located in a column called DEFERPREP in the SYSIBM.SYSPACKAGE table. The column name can be confusing because it does NOT relate to whether or not the package creation is deferred.

To display the counts for each type of package, you can use the query in Example A-21.

Example: A-21 CURRENTDATA usage for packages

```
SELECT
CASE(DEFERPREP)
WHEN 'A' THEN 'A-CURRENTDATA-YES-ALL-CURSORS'
WHEN 'B' THEN 'B-CURRENTDATA-NO -AMBIGUOUS-CURSOR'
WHEN 'C' THEN 'C-CURRENTDATA-YES-AMBIGUOUS-CURSOR'
WHEN ' ' THEN 'D-PRE-CURRENTDATA'
END AS USAGE
,ISOLATION
,COUNT(*) AS NUMBER_OF_PACKAGES
FROM SYSIBM.SYSPACKAGE
WHERE DEFERPREP IN ('A','B','C',' ')
GROUP BY DEFERPREP,ISOLATION
WITH UR;
```

For local packages, CURRENTDATA only applies if the package is bound with CS. For remote packages, CURRENTDATA applies with RR, RS and CS.

The query is listing what is in the catalog. See the discussion in the *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841 for the interpretation of these values.

A.22 Selecting only the most useful EXPLAIN information

The PLAN_TABLE contains information about the access path of a query. There are several columns in the PLAN_TABLE, of which a few are almost always required. Example A-22 shows one way of obtaining the most useful information from the latest bind of the package.

Multiple index access is not shown in this example. If used, you need to add MXOPSEQ to your order by clause.

Example: A-22 Most useful EXPLAIN information

```
SELECT
 SUBSTR(PLNTBL.COLLID,1,8)           AS COLLECTION
,SUBSTR(PLNTBL.PROGNAME,1,8)         AS PACKAGE
,SUBSTR(DIGITS(PLNTBL.QUERYNO),5,5)  AS QNUMB
,SUBSTR(DIGITS(PLNTBL.QBLOCKNO),3,3) AS QB
,SUBSTR(DIGITS(PLNTBL.PLANNO),3,3)   AS PLN
,SUBSTR(DIGITS(PLNTBL.METHOD),3,3)   AS MTH
,SUBSTR(PLNTBL.TNAME,1,18)           AS TABLE_NAME
,SUBSTR(DIGITS(PLNTBL.TABNO),3,3)    AS TNO
,PLNTBL.ACCESSTYPE                   AS ACTYP
,SUBSTR(PLNTBL.ACCESSNAME,1,8)       AS INDEX_NAME
,PLNTBL.MATCHCOLS                    AS MCOL
,PLNTBL.INDEXONLY                    AS IO
```

```
,PLNTBL.PREFETCH                     AS PF
,(SORTN_UNIQ    CONCAT
  SORTN_JOIN    CONCAT
  SORTN_ORDERBY CONCAT
  SORTN_GROUPBY)                       AS UJOGN
,(SORTC_UNIQ    CONCAT
  SORTC_JOIN    CONCAT
  SORTC_ORDERBY CONCAT
  SORTC_GROUPBY)                       AS UJOGC

FROM PLAN_TABLE PLNTBL
WHERE PLNTBL.COLLID = 'TEMP'
  AND PLNTBL.PROGNAME = 'PGM10'
AND PLNTBL.BIND_TIME IN
  (SELECT MAX(MAX_BND_TM.BIND_TIME) FROM PLAN_TABLE MAX_BND_TM
   WHERE PLNTBL.PROGNAME = MAX_BND_TM.PROGNAME
     AND PLNTBL.COLLID   = MAX_BND_TM.COLLID)
ORDER BY  PLNTBL.QUERYNO
         ,PLNTBL.QBLOCKNO
         ,PLNTBL.PLANNO
WITH UR;
```

B

DSNZPARMs, trigger packages, and tools

In this appendix we discuss the DSNZPARMs relevant to packages, trigger packages, and briefly cover some of the tools available to develop, manage and monitor packages.

This appendix covers the following topics:

- DSNZPARMs
- Trigger packages
- Tools dealing with packages

B.1 DSNZPARMs

In this section, we discuss the DSNZPARMs relevant to packages. For details, refer to *DB2 Version 9.1 for z/OS Installation Guide,* GC18-9846.

ABIND - AUTO BIND

ABIND specifies whether to allow auto rebind of packages or plans at runtime. Automatic rebinds can improve usability and availability but they can also impact performance when heavily accessing DB2 directory and catalog.

The valid values are YES, NO, COEXIST:

- YES - DB2 automatically rebinds a package or plan one time at execution time on these conditions:
  - If it is marked as non VALID, the SYSPLAN or SYSPACKAGE column VALID contains “N”.
  - Was last bound in DB2 9 but is now running on a previous version of DB2.

    This autobind is performed only for the first attempt to run the plan or package on V8
  - Was last autobound on DB2 V8 but is now running again on V9

    This V8-to-V9 autobind is performed on DB2 9 because that is where the user is attempting to run the plan or package. This autobind is performed only for the first attempt to run the plan or package on V9 after an autobind on V8.

  Note: You can tell if a package was last bound on DB2 9 by checking SYSPACKAGE.RELBOUND

- NO - Automatic rebind operations are not allowed under any circumstances.
  - You must explicitly rebind any plans or packages that are invalid, inoperative or are on a version prior to the latest bind, in case of fallback of a release migration. Also if the package was last bound with an AUTO BIND.

  If you attempt to run any plan or package on DB2 9 one of the situations described before, you receive SQLCODE -908 SQLSTATE 23510.
- COEXIST - This option allows automatic rebind operations to be performed in a DB2 data sharing environment only when:
  - SYSPLAN or SYSPACKAGE is marked invalid
  - The plan or package was last bound on DB2 9 and is now running on DB2 V8. In this case, DB2 performs an automatic rebind on V8 for the plan or package before running it there. This automatic rebind is performed only for the first attempt to run the plan or package on V8.

Important: If a package is bound on a DB2 9 system and the system falls back to version 8, if an auto bind is performed while in Version 8, after the migration back to DB2 9, the package runs as a Version 8 package. No DB2 9 features is used in this package, such as optimization improvements, improved access paths or new index enhancements. In order to gain this functionality, you must do a explicit BIND or REBIND.

Tip: To reduce the rate of automatic rebinds in a data sharing environment, consider the use of COEXIST for AUTO BIND

ABEXP - EXPLAIN PROCESSING

ABEXP defines whether to use EXPLAIN processing when an auto bind of a plan or package occurs:

- If the plan or package is bound with EXPLAIN(YES) and a plan table exists for the OWNER of the plan/package, then EXPLAIN processing occurs.
- If the plan/package is bound with EXPLAIN(YES) and the PLAN_TABLE for the OWNER does not exist, then auto bind continues but EXPLAIN processing does not occur.
- If the plan/package is bound with EXPLAIN(NO), no EXPLAIN processing takes place even if ABEXP is specified as YES.
- If ABEXP is defined as NO, no EXPLAIN processing occurs, even for packages bound with EXPLAIN(YES).

APPENSCH - APPLICATION ENCODING

Specifies the system default application encoding scheme (such as ASCII, EBCDIC, UNICODE, or CCSID (1 to 65533))

AUTHCACH - PLAN AUTH CACHE

AUTHCACH defines the authorization caching size. It is the default for a plan for authorization checking if no CACHESIZE is specified on the BIND PLAN. The default is 3072 and can be specified from 0 to 4096 in multiples of 256. You can use authorization cache to reduce the amount of time DB2 spends checking authorizations for frequently executed plans.

BINDNV - BIND NEW PACKAGE

BINDNV sets the authority for VERSIONs.

If you specify a value of BIND, a user with BIND authority on a package can create a new VERSION. If you specify BINDADD, only users with BINDADD authority can add new VERSIONs of a package.

CACHEDYN - CACHE DYNAMIC SQL

CACHEDYN determines whether or not to cache prepared, dynamic SQL statements. This has impact on the size of your EDM POOL. See *DB2 Version 9.1 for z/OS Installation Guide*, GC18-9846, Calculating EDM pool space for the prepared statement cache.

CACHEPAC - PACKAGE AUTH CACHE

CACHEPAC sets how much storage to allow for ALL package authorization caching in your DB2 subsystem. The default is 100 KB and can be specified from 0 to 5 MB. You can use package authorization cache to reduce the amount of time DB2 spends checking authorizations for frequently executed packages.

CACHERAC - ROUTINE AUTH CACHE

CACHERAC defines how much storage to allow for ALL stored procedures and user defined function caching in your DB2 subsystem. The default is 100K and can be specified from 0 to 5M. You can use routine authorization cache to reduce the amount of time DB2 spends checking authorizations for frequently executed stored procedures and user defined functions.

CDSSRDEF - CURRENT DEGREE

CDSSRDEF defines what default degree of parallelism is used on plans and packages that do not specify CURRENT DEGREE during the BIND. As a system parameter, this should be set to 1 and use the CURRENT DEGREE at the package level to selectively implement parallelism.

CMTSTAT - DDF THREADS

Specifies whether to make a thread active or inactive after it successfully commits or rolls back and holds no cursors.

DESCSTAT - DESCRIBE FOR STATIC

Specifies whether DB2 is to build a DESCRIBE SQLDA when binding static SQL statements.

DYNRULS - USE FOR DYNAMICRULES

Specifies whether DB2 should use the application programming defaults that are specified on this panel or use the values of the DB2 precompiler options for dynamic SQL statements that are bound by using DYNAMICRULES bind, define, or invoke behavior.

MAINTTYPE - CURRENT MAINT TYPES

CURRENT MAINT TYPES defines how to populate MQTs.

If you use CURRENT REFRESH AGE of ANY, which enables the possibility of using query rewrite for MQTs.

- Specifying SYSTEM allows DB2 to use system maintained MQTs for query rewrite.
- Specifying USER allows DB2 to use user maintained MQTs for query rewrite.
- Specifying ALL allows DB2 to use both system and user maintained MQTs for query rewrite.
- Specifying NONE does not allow DB2 to use any MQTs for query rewrite.

MAXKEEPD - MAX KEPT DYN STMTS

MAXKEEPD limits the number of statements that are kept in the dynamic statement cache past a commit point if KEEPDYNAMIC(YES) is specified. DB2 can mimic the KEEPDYNAMIC(YES) parameter by performing implicit prepares to rebuild the executable version of the dynamic SQL for statements moved out of the statement cache.

MAX_CONCURRENT_ PKG_OPS

MAX_CONCURRENT_PKG_OPS sets the number of *automatic* bind requests that can be processed concurrently. The default value is 10. If the value of the AUTO BIND field of panel DSNTIPO is set to NO, the value of MAX_CONCURRENT_PKG_OPS has no effect.

OPTHINTS - OPTIMIZATION HINTS

Optimization hints are used to guide the optimizer in choosing specific access paths. The default for this option is NO. If you accept this default, you are not able to use Optimization hints.

Tip: Set this value to YES to give you the option of using optimization hints.

PARAMDEG - MAX DEGREE

PARAMDEG sets the maximum number of parallel tasks that your DB2 subsystem can use. The default is 0, which means that DB2 chooses a maximum degree based upon your environment.

PLANMGMT - PACKAGE STABILITY

PLANMGMT defines whether to use PLANMGMT(NONE) - default-, which keeps only the current package in SPT01. PLANMGMT(BASIC) keeps two copies in SPT01. PLANMGMT(EXTENDED) keeps up to three copies of each package in SPT01 and SYSPACKDEP. See 10.13, “Package stability” on page 216 for more details on this function.

REFSHAGE - CURRENT REFRESH AGE

CURRENT REFRESH AGE enables query rewrite for materialized query tables (MQTs).

If you accept the default of 0 for this DSNZPARM, you disable query rewrite for materialized query tables (MQT). Use ANY if you are using MQTs. This parameter is updateable dynamically.

SMFACCT - SMF ACCOUNTING

Specifies whether DB2 is to send accounting data to SMF automatically when DB2 is started. This field also specifies what classes are sent.

SMFSTAT - SMF STATISTICS

Specifies whether DB2 is to send statistical data to SMF automatically when DB2 is started. This field also specifies what classes are sent.

STORMXAB - MAX ABEND COUNT

STORMXAB -defines how many times can a stored procedure abend before requests to the stored procedure are rejected. A value of 0 means that after the first abend from a stored procedure, all subsequent calls are rejected until the procedure or UDF is started again.

B.2 Trigger packages

Trigger packages in your system can be identified by running the SQL shown in Example A-7 on page 264.

You can use the REBIND TRIGGER PACKAGE command to change a limited subset of the default bind options that DB2 used when creating the package. You might also rebind a trigger package to re-optimize its SQL statements after you create a new index or use the RUNSTATS utility. Additionally, you can rebind a trigger package if it has been marked invalid because an index, or another object it was dependent on, was dropped.

To obtain the access path information for a trigger, you must REBIND TRIGGER PACKAGE with the EXPLAIN option turned on. This is because the trigger package is created as a result of the CREATE TRIGGER statement and cannot be explicitly bound.

See Chapter 5., “Application development framework” on page 69 for further details.

B.3 Tools dealing with packages

We list here, with a brief description, IBM tools that can be useful when dealing with packages

B.3.1 IBM DB2 Bind Manager for z/OS

IBM DB2 Bind Manager for z/OS allows application programmers to safely bypass the DB2 bind process for code changes that do not alter existing SQL structures in an application.

A “catalog cleanup” feature lets you compare bound DB2 packages to their load libraries. Where there is no match, a Free Command removes obsolete packages and a backup of the DBRM is recreated from the system catalog. If you have missing DBRMs, you can recreate them from the system catalog. DB2 Bind Manager, used in conjunction with DB2 Path Checker, reduces the amount of catalog data to be migrated, and also helps identify potentially harmful access path changes before they occur.

Key features

Here we list some key features of this tool:

- Determines if a bind is required after an application has been precompiled and automatically resets the time stamp and bypasses the bind.
- Handles consistency checking between an existing DBRMLIB and a DB2 subsystem (through the DBRM Checker function).
- Matches time stamps in a DBRM to time stamps in a load module.
- Scans a load library for modules and CSECTS that contain static SQL.

B.3.2 IBM DB2 Path Checker for z/OS

IBM DB2 Path Checker for z/OS helps you increase the stability of your DB2 environments and avoid painful and costly disruptions. It can help you discover and correct unwanted and unexpected access path changes before you are notified about them. These changes can occur as a result of binding when installing a new release of DB2, applying service, or migrating an application from one system to another. DB2 Path Checker can efficiently scan hundreds or thousands of SQL statements and identify just the statement or statements that have experienced or will experience an access path change. It can seamlessly integrate into your change control procedures for automated protection.

Key features

Here we list some key features of this tool:

- Predict whether a bind of a DBRM results in a changed access path
- Run path checks on a batch of DBRMs in one pass
- Reduce costs by avoiding unnecessary binding steps between application programs and the database (when used in conjunction with DB2 Bind Manager for z/OS).
- Compare DBRMs across subsystems and load modules
- Access the compare function with ISPF support
- Skip multiple binds for the latest version and compare the access path to the prior version using COMPARE TO PREVIOUS VERSION. This is useful when a program has been rebound multiple times due to an access path issue.

- View a history table for a summary of changes detected in DB2 Path Checker processing. This table contains the same information as the SYSCHG ddname report.

B.3.3 IBM DB2 SQL Performance Analyzer

IBM DB2 SQL Performance Analyzer provides you with an extensive analysis of SQL queries without executing them. This analysis aids you in tuning your queries to achieve maximum performance.

DB2 SQL Performance Analyzer delivers complete integration with Easy Explain in both TSO and batch, allowing you to compare the costs of old and new plans and to re-explain existing plans stored in any plan table. Programming language data sets can be scanned directly to extract SQL for study.

Key features

Here we list some key features of this tool:

- Drill down to view all the SQL or all the DB2 tables within the given input source (such as plans, packages, and DBRMs). This enables you to explain a single SQL query or all of the SQL for the DB2 table that is selected.
- Use the DSN8EXP procedure to enable application programmers to see SQL performance against tables for which they do not have authority to run SQL.
- Offers “what-if” scenarios to help determine the performance that can be obtained using various database design alternatives and production volumes. “What-if” analysis capabilities include updating catalog statistics and creating new indexes.
- Provides user-modifiable thresholds that can set high water marks, trigger warnings and alerts, and control advisory messages.
- Migrates catalog statistics to test machines for in-depth analysis of production applications. Analyzes new access paths, determines if action is needed, and estimates costs of new paths in database resources consumed when used with DB2 Path Checker.

B.3.4 IBM Data Studio pureQuery Runtime

IBM’s pureQuery Runtime is a new, high-performance Java data access platform focused on simplifying the tasks of developing and managing data access. The benefits of using pureQuery extend throughout the development, deployment, management, and governance stages of the data life cycle. pureQuery provides access to data in databases and in-memory Java objects with its tools, APIs, and runtime environment.

We used pureQuery in Chapter 6, “Developing distributed applications” on page 101.

Key features

Here we list some key features of this tool:

- Use pureQuery APIs to build your applications with much less code in comparison with JDBC
- Build applications to query in-memory collections and databases using one single API
- Work with XML JPA format to keep all SQL in one location outside of your Java files
- Bind static SQL to improve the performance and management of applications

When used with IBM Data Studio Developer, you can:

- Bridge the gap between relational data and Java, working seamlessly within the Eclipse environment
- Develop or customize the SQL inside Java applications to your needs using the highly sophisticated SQL editor integration inside the Java editor
- Build sample Java applications to access IBM data servers without writing a single line of code
- Have flexibility in application development to build pureQuery applications starting from database objects, SQL, or Java beans with highly integrated, rich, and easy-to-use tools
- Create pureQuery applications to work with tables, views, aliases, or procedures
- Bind static SQL in pureQuery applications to data sources
- Auto-generate test applications and JUnits for generated code

B.3.5 IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS

IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS gives you a single, comprehensive tool to help assess the efficiency of—and optimize performance from—your DB2 on z/OS environment. The software combines the sophisticated reporting, monitoring and buffer pool analysis features of IBM Tivoli OMEGAMON XE for DB2 Performance Monitor on z/OS and IBM performanceDB2 Buffer Pool Analyzer products. It also adds expert database analysis functions to help you maximize performance and enhance productivity.

Robust graphical data views in Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS inform your IT staff about the state of your DB2 subsystem and its applications. The software collects trace data and provides performance counters and separate data views of relevant database statistics, such as:

- Buffer pool and environmental descriptor manager (EDM) pool usage.
- Structured query language (SQL) activities.
- Locking and log manager information.

Additionally, the product provides granular control over routine DB2 subsystem operations to help you maximize performance.

Examples include:

- Monitoring data-sharing groups for threads and statistical information.
- Drilling down from views of thread activities to the SQL and locked resource levels to understand the progress or status of the applications.
- Displaying locking conflicts — allowing you to discover more thread details or cancel a specific thread.
- Tracking near-term performance history to see problems that might otherwise go unnoticed and predict future problems.
- Conducting object analysis of database tables, table spaces and other elements to tune performance.

B.3.6 Optimization Service Center

IBM Optimization Service Center for DB2 for z/OS is a Windows workstation tool designed to ease the workload of DBAs by providing a rich set of autonomic tools that help optimize query performance and workloads. OSC is built on Eclipse, open-source platform for the construction of powerful software development tools and rich desktop applications. We show this tool at 10.2, “Using Optimization Service Center to view access paths” on page 201.

C

Catalog tables and EXPLAIN tables

In this appendix, we list the attributes of the catalog tables deemed important from the perspective of using packages. For each table, we provide the indexes available, which columns are typically used to access the table, and some of the important columns to look at.

We also include information about the base and extended EXPLAIN tables that capture the result of an EXPLAIN.

This appendix covers the following topics:

- Package related catalog tables
- EXPLAIN tables

For sample queries to access these tables, see Appendix A, “Useful queries” on page 259.

C.1 Package related catalog tables

We provide information about the following catalog tables:

- SYSDBRM
- SYSOBJROLEDEP
- SYSPACKAGE
- SYSPACKAUTH
- SYSPACKDEP
- SYSPACKLIST
- SYSPACKSTMT
- SYSPKSYSTEM
- SYSPLAN
- SYSPLANAUTH
- SYSPLANDEP
- SYSPLSYSTEM
- SYSRESAUTH
- SYSROUTINES
- SYSSTMT
- SYSTRIGGERS
- SYSUSERAUTH

Important: We cover only the important columns and common accesses. For complete information about all DB2 catalog and directory tables, see *DB2 Version 9.1 for z/OS SQL Reference* SC18-9854.

C.1.1 Relationship among package related tables

Before you look at the important columns in each of the tables, you need to understand how these tables are logically related to each other. Several of the tables do not have unique indexes on them, so the representation shown is strictly logical. Also keep in mind that the authorization table for packages (SYSPACKAUTH) does not contain the version or CONTOKEN of the package.

The tables can be roughly divided in to three classes:

- Those related to packages
- Those related to plans (SYSPACKLIST connects these two classes)
- Those related to global authorizations

Figure C-1 shows this relationship among the tables.

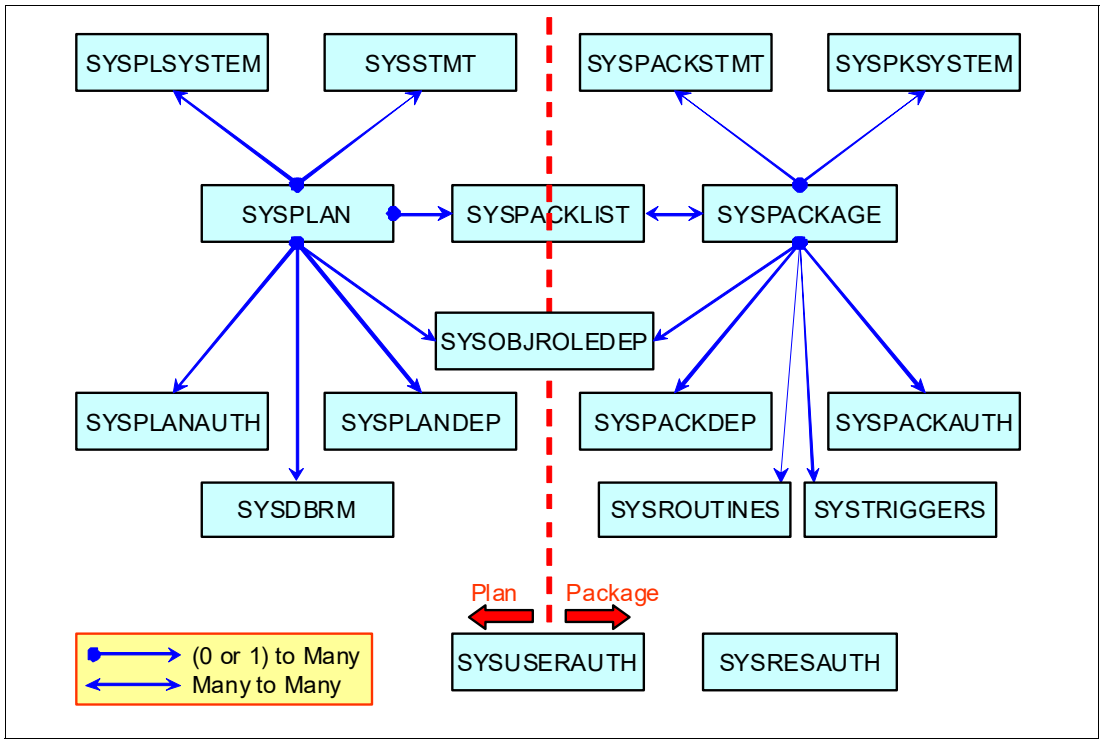

Figure C-1 Catalog table relationships

C.1.2 SYSDBRM

This table contains one row for each DBRM of each application plan. Plans containing only packages do not have an entry in this table, and after you migrate away from DBRM-based plans, this table becomes irrelevant.

Indexes on this table

There are no indexes on this table.

However, if you still have many DBRM-based plans, and you query the table, we suggest creating an index by PLNAME and NAME, as well as a second index by NAME and PLNAME.

Typical access to the table

This table is typically accessed by PLNAME to see all DBRMs in the plan or by NAME to see all plans that have the DBRM bound into them.

Important columns

Some of the important columns to look at are:

- NAME

  Name of the DBRM.

- TIMESTAMP

  Consistency token. You need to use the HEX function to display this properly as indicated in Example A-10 on page 265.

- PLNAME

  Name of the application plan of which the DBRM is a part.
- PLCREATOR

  authorization ID of the owner of the application plan.
- VERSION

  Version identifier of the DBRM.
- PLCREATORTYPE

  Indicates the type of creator (authorization ID or role) of the plan.

C.1.3 SYOBJROLEDEP

This table lists the dependent objects for a role. The objects of interest to us are those dependencies related to packages and plans.

Indexes on this table

The indexes on this table are:

- DSNRDX01 - DSCHEMA, DNAME, DTYPE, ROLENAME
- DSNRDX02 - ROLENAME

Typical access to the table

This table is typically accessed by DSCHEMA, DNAME, DTYPE, and ROLENAME to see what roles have BIND or EXECUTE privileges on a plan or a package. Alternatively, the table could be accessed by ROLENAME to see what privileges are held by a role.

Important columns

Some of the important columns to look at are:

- ROLENAME

  Name of the role on which there is a dependency.
- DSCHEMA

  Name of the schema of the dependent object. (COLLID for packages and OWNER for plans).
- DNAME

  Name of the dependent object (package or plan name).
- DTYPE

  The type of dependent object in DNAME. Of interest to us are: K = Package and P = Plan.

C.1.4 SYSPACKAGE

This table contains a row for every package.

Important: The LOCATION column always contains blanks.

Indexes on this table

The indexes on this table are:

- DSNKKX01 - LOCATION, COLLID, NAME, VERSION
- DSNKKX02 - LOCATION, COLLID, NAME, CONTOKEN

Because the LOCATION is always blank, it is a common mistake to leave out the LOCATION in your search criteria. In most cases, (assuming the overhead of the extra index is not significant), we recommend that an index by COLLID, NAME, VERSION be created, because this is the most common access used by application developers.

Typical access to the table

This table is typically accessed by COLLID and NAME (see the previously suggested index).

Important columns

Some of the important columns to look at are:

- LOCATION,COLLID,NAME,CONTOKEN

  These columns comprise the unique key for the package used at execution time.

- LOCATION,COLLID,NAME,VERSION

  These columns comprise the unique index used by the DB2 BIND process.

- OWNER

  Authorization ID of the package owner. For a trigger package, the value is the authorization ID of the owner of the trigger and for a stored procedure, it is the schema name.

- PCTIMESTAMP

  Date and time when the DBRM for this package was last pre-compiled.

- TIMESTAMP

  Timestamp indicating when the package was created.

- BINDTIME

  Timestamp indicating when the package was last bound.

- QUALIFIER

  Implicit qualifier for the unqualified table, view, index, and alias names in the static SQL statements of the package.

- AVGSIZE

  The average section size

- CACHESIZE

  Used with AVGSIZE for estimating memory needs.

- PKSIZE

  This is the size of the base section of the package. The base section of the package is resident in the EDMPOOL for the entire time the package is executing. Other sections can be moved in and out of the EDMPOOL as needed.

- VALID

  Whether the package is valid. If the package is not valid, the package can be rebound automatically depending on your system parameters.

- OPERATIVE

  Whether the package can be allocated. If the package is not operative, it must be bound or rebound explicitly.

Important: VALIDATE, DEFERPREP, and DEFERPREPARE options appear to be similar but have vastly different meanings.

- VALIDATE

  Whether validity checking must be performed at BIND time or can be deferred until execution time. See 2.1.11, “VALIDATE and SQLERROR in BIND PACKAGE” on page 24 for details on this option,

- DEFERPREP

  Indicates the CURRENTDATA option when the package was bound or rebound.

- DEFERPREPARE

  Whether PREPARE processing is deferred until OPEN is executed (as opposed to when the PREPARE is issued.

- ISOLATION

  The isolation level of the package, for example CS=Cursor Stability, RR = Repeatable read and so on.

- RELEASE

  The release option used when the package was last bound, for example C=Commit, D=DEALLOCATE and blank=‘it was not specified’ at the package level and inherits the RELEASE level from the PLAN.

- EXPLAIN

  Whether the PACKAGE was bound with EXPLAIN(YES) or EXPLAIN(NO).

Tip: All packages should be bound with EXPLAIN(YES) to obtain the access path information.

- DEGREE

  The DEGREE option used when the package was last bound (to request or disable parallelism). Note that requesting the package for parallelism does not mean that parallelism is always deployed.

- DYNAMICRULES

  The DYNAMICRULES option used when the package was last bound.

- REOPTVAR

  Whether the access path is determined again at execution time using input variable values.

- KEEPDYNAMIC

  Whether prepared dynamic statements are to be kept after each commit.

- OPTHINT

  Identifies rows in the *owner*.PLAN_TABLE to be used as input to DB2.

- OWNERTYPE

  If this is populated with an L, it means that a ROLE owns the package.

- SYSENTRIES

  Number of enabled or disabled entries for this package in SYSPKSYSTEM. A value of 0 if all types of connections are enabled.

C.1.5 SYSPACKAUTH

This table records the privileges that are held by users over packages. The granularity of package authority is at the package level. Granting or revoking for a package is for all versions.

Indexes on this table

The indexes on this table are:

- DSNKAX01 - GRANTOR, LOCATION, COLLID, NAME, GRANTORTYPE
- DSNKAX02 - GRANTEE, LOCATION, COLLID, NAME, BINDAUTH, COPYAUTH, EXECUTEAUTH, GRANTEETYPE
- DSNKAX03 - LOCATION, COLLID, NAME

Typical access to the table

This table is typically accessed by LOCATION, COLLID, and NAME to see all authorizations for this package. Alternatively, the table can be accessed by GRANTEE to see all authorizations for this authorization ID or role.

Important columns

Some of the important columns to look at are:

- GRANTOR

  Authorization ID of the user who granted the privilege.

- GRANTEE

  Authorization ID of the user who holds the privilege or the name of the plan that uses the privilege.

- COLLID

  Collection name for the package or packages on which the privilege was granted. Note that it possible to grant a privilege to the entire collection by using the COLLID.* option of the command.

- NAME

  Name of the package on which the privileges are held.

- GRANTEETYPE

  Type of grantee: (An authorization ID, role or an application plan).

- BINDAUTH

  Whether GRANTEE can use the BIND and REBIND subcommands on the package.

- COPYAUTH

  Whether GRANTEE can COPY the package.

- EXECUTEAUTH

  Whether GRANTEE can execute the package.

C.1.6 SYSPACKDEP

This table records the dependencies of packages on local tables, views, synonyms, table spaces, indexes, aliases, functions, and stored procedures.

Indexes on this table

The indexes on this table are:

- DSNKDX01 - DLOCATION, DCOLLID, DNAME, DCONTOKEN
- DSNKDX02 - BQUALIFIER, BNAME, BTYPE
- DSNKAD03 - BQUALIFIER, BNAME, BTYPE, DTYPE

Typical access to the table

This table is typically accessed by DCOLLID and DNAME to see the objects that this package is dependent on, or by BQUALIFIER and BNAME to see the packages that are dependent on this object.

Important columns

Some of the important columns to look at are:

- BNAME

  The name of an object that a package depends on.
- BQUALIFIER

  The value of the column depends on the type of object.
- BTYPE

  Type of object identified by BNAME and BQUALIFIER.
- DCOLLID

  Name of the package collection.
- DNAME

  Name of the package.
- DCONTOKEN

  Consistency token for the package.
- DOWNER

  Owner of the package.
- DTYPE

  Type of package (for example, N= native SQL procedure, T= trigger). If you are using package stability (see 10.13, “Package stability” on page 216), O = original copy, P = previous copy.

C.1.7 SYSPACKLIST

This table contains one or more rows for every local application plan bound with a package list. Each row represents a unique entry in the plan’s package list.

Indexes on this table

The indexes on this table are:

- DSNKLX01 - LOCATION, COLLID, NAME
- DSNKLX02 - PLANNAME, SEQNO, LOCATION, COLLID, NAME

We also recommend an index by COLLID and NAME because the LOCATION is blank for local packages, and a common query is “which plans contain this package.”

Typical access to the table

This table is typically accessed by PLANNAME to see all packages in the plan in the correct sequence (ordered by SEQNO). Alternatively, it could be used to see which plans contain a specific package identified by COLLID and NAME (see the previous suggested index).

Important columns

Some of the important columns to look at are:

- PLANNAME

  Name of the application plan.

- SEQNO

  Sequence number of the entry in the package list.

- LOCATION

  Location of the package. Blank if this is local. An asterisk (*) indicates location to be determined at runtime.

- COLLID

  Collection name for the package. An asterisk (*) indicates that the collection name is determined at runtime.

- NAME

  Name of the package. An asterisk (*) indicates an entire collection.

C.1.8 SYSPACKSTMT

This table contains one or more rows for each statement in a package.

Indexes on this table

The indexes on this table are:

- DSNKSX01 - LOCATION, COLLID, NAME, CONTOKEN, STMTNOI,SECTNOI,SEQNO

Typical access to the table

This table is typically accessed by COLLID and NAME to see the text for all SQL statements in that package. It is ordered by CONTOKEN, STMTNOI, SEQNO within this package. Because the text is stored as varying-length string, it is typically “sliced up” to make it readable.

Important columns

Some of the important columns to look at are:

- COLLID

  Name of the package collection.

- NAME

  Name of the package.

- CONTOKEN

  Name of the package collection.

- VERSION

  Version identifier for the package.

- SEQNO

  Sequence number of the row with respect to a statement in the package. The numbering starts with 0.

- STMTNO/STMTNOI

  The statement number of the statement in the source program. If the STMTNOI is populated (and not zero), you should use this value. If not, you should use STMTNO. Note that STMTNO is a SMALLINT (maximum value it can hold is 32,767) while STMTNOI is an INTEGER column an does not have this limitation.

  To summarize: use STMTNOI always. Also note that both can be zero and represent DB2 rows.

- STMT

  All or a portion of the text for the SQL statement that the row represents.

- ACCESSPATH

  For static statements, indicates if the access path for the statement is based on user-specified optimization hints. A value of 'H' indicates that optimization hints were used. See 10.9, "Optimization hints" on page 209 for details.

- QUERYNO

  The query number of the SQL statement in the source program.

C.1.9 SYSPKSYSTEM

This table contains zero or more rows for every package. Each row for a given package represents one or more connections to an environment in which the package could be executed. If the package is enabled for all systems, this table does not contain any rows. The column SYSENTRIES on SYSPACKAGE contains the number of rows in this table for a given package, which is always 0 (enable for all) or 8 (enabled for some and disabled for others).

The index on this table is:

- DSNKSX01 - LOCATION, COLLID, NAME, CONTOKEN,SYSTEM,ENABLE

Typical access to the table

This table is typically accessed by COLLID and NAME to see if the package has been explicitly enabled or disabled for any environments. It is also possible that the table is scanned by ENABLE or DISABLE column to see all exceptions if the practice is NOT to enable or disable, in general.

Important columns

Some of the important columns to look at are:

- COLLID

  Name of the package collection.

- NAME

  Name of the package.
- CONTOKEN

  Consistency token for the package.
- SYSTEM

  Environment (for example, BATCH, CICS, IMSBMP, and so on)
- ENABLE

  Indicates whether the connections represented by the row are enabled or disabled.
- CNAME

  Identifies the connection or connections to which the row applies (for example, requestor's location).

C.1.10 SYSPLAN

This table contains one row for each application plan. Several of these options apply to only to plans containing DBRMs. See Chapter 2, "BIND options" on page 17 for details.

The index on this table is:

- DSNPPH01 - NAME

Typical access to the table

This table is typically accessed by NAME to see the attributes of the plan.

Important columns

Some of the important columns to look at are:

- NAME

  Name of the application plan.
- CREATOR

  Authorization ID of the owner of the application plan.
- VALIDATE

  Whether validity checking can be deferred until runtime.
- ISOLATION

  Isolation level for the plan.
- VALID

  Whether the application plan is valid.
- OPERATIVE

  Whether the application plan can be allocated.
- PLSIZE

  Size of the base section of the plan, in bytes.
- ACQUIRE

  When resources are acquired.
- RELEASE

When resources are released.

- EXPLAIN

  Option specified for the plan; that is, whether information on the plan's statements was added to the owner's PLAN_TABLE table.

- BOUNDBY

  Primary authorization ID of the last binder of the plan.

- QUALIFIER

  Implicit qualifier for the unqualified table, view, index, and alias names in the static SQL statements of the plan.

- DEFERPREP

  Whether the package was last bound with the DEFER(PREPARE) option.

- DEGREE

  The DEGREE option used when the plan was last bound to request or disable parallelism.

- DYNAMICRULES

  The DYNAMICRULES option used when the plan was last bound.

- REOPTVAR

  Whether the access path is determined again at execution time using input variable values.

- KEEPDYNAMIC

  Whether prepared dynamic statements are to be kept after each commit point.

- OPTHINT

  Value of the OPTHINT bind option. Identifies rows in the owner.PLAN_TABLE to be used as input to DB2.

- SYSENTRIES

  Number of rows associated with the plan in SYSPLSYSTEM. The negative of that number if such rows exist but the plan was bound in a prior release after fall back. A negative value or zero means that all connections are enabled.

C.1.11 SYSPLANAUTH

This table records the privileges that are held by users over application plans.

Indexes on this table

The indexes on this table are:

- DSNAPH01 - GRANTEE, NAME, EXECUTEAUTH, GRANTEETYPE
- DSNAPX01 - GRANTOR, GRANTORTYPE

We suggest an additional index by NAME and GRANTEE.

Typical access to the table

This table is typically accessed by GRANTEE and NAME to see what authorization has been granted to this authorization ID or role for the specific plan. Alternatively, the table is accessed by NAME to see which authorization IDs or roles have authorizations granted for this plan.

Important columns

Some of the important columns to look at are:

- GRANTOR

  Authorization ID of the user who granted the privileges.

- GRANTEE

  Authorization ID of the user who holds the privileges.

- NAME

  Name of the application plan on which the privileges are held.

- BINDAUTH

  Whether the GRANTEE can use the BIND, REBIND, or FREE subcommands against the plan.

- EXECUTEAUTH

  Whether the GRANTEE can run application programs that use the application plan.

- GRANTEETYPE

  Indicates the type of grantee (authorization ID or role).

- GRANTORTYPE

  Indicates the type of grantor (authorization ID or role).

C.1.12 SYSPLANDEP

This table records the dependencies of plans on tables, views, aliases, synonyms, table spaces, indexes, functions, and stored procedures. It is populated only if you have any DBRM-based plans and it should become irrelevant as you migrate away from these.

The index on this table is:

- DSNGGX01 - BCREATOR, BNAME, BTYPE

Typical access to the table

This table is typically accessed by BCREATOR and BNAME to see all dependent plans for this object (a table or index most likely). Alternatively, it could be accessed by DNAME to show all objects on which this plan is dependent.

Important columns

Some of the important columns to look at are:

- BNAME

  The name of an object the plan depends on.

- BCREATOR

  If BNAME is a table space, its database. Otherwise, the schema.

- BTYPE

  Type of object identified by BNAME(e.g table, index, view, and so on).

- DNAME

  Name of the plan.

C.1.13 SYSPLSYSTEM

This table contains zero or more rows for every plan. Each row for a given plan represents one or more connections to an environment in which the plan could be used.

The index on this table is:

- DSNKPX01 - NAME, SYSTEM, ENABLE

Typical access to the table

This table is typically accessed by NAME to see if the plan has been explicitly enabled or disabled for any environments. It is also possible that the table is accessed by ENABLE column if enabling or disabling is not the norm.

Important columns

Some of the important columns to look at are:

- NAME

  Name of the plan.

- SYSTEM

  Environment (for example, BATCH, CICS, IMSBMP, and so on).

- ENABLE

  Indicates whether the connections represented by the row are enabled or disabled.

- CNAME

  Identifies the connection or connections to which the row applies. Interpretation depends on the environment specified by SYSTEM.

C.1.14 SYSRESAUTH

The SYSRESAUTH table records CREATE IN and PACKADM privileges for collections; USAGE privileges for distinct types; and USE privileges for buffer pools, storage groups, and table spaces.

Indexes on this table

The indexes on this table are:

- DSNAGH01 - GRANTEE, QUALIFIER, NAME, OBTYPE, GRANTEETYPE
- DSNAGX01 - GRANTOR, QUALIFIER, NAME, OBTYPE, GRANTORTYPE

Typical access to the table

This table is typically accessed by NAME (and OBTYPE = ‘C’) to see which authorization IDs or roles have been granted privileges to this collection.

Important columns

Some of the important columns to look at are:

- GRANTOR

  Authorization ID of the user who granted the privilege.

- GRANTEE

  Authorization ID of the user who holds the privilege.

- QUALIFIER

  Misc. values - the value is PACKADM if the privilege is for a collection (OBTYPE='C') and the authority held is PACKADM.

- NAME

  Name of the object e.g C=collection.

- OBTYPE

  Type of object (in particular, C=collection).

- GRANTEETYPE

  Indicates the type of grantee (authorization ID or role).

- GRANTORTYPE

  Indicates the type of grantor (authorization ID or role).

C.1.15 SYSROUTINES

This table contains a row for every routine (user-defined function, cast function created for a distinct type or stored procedure).

Indexes on this table

The indexes on this table are:

- DSNOFX01 - NAME, PARM_COUNT, ROUTINETYPE,PARM_SIGNATURE, SCHEMA, PARM1, PARM2, PARM3, PARM4, PARM5, PARM6, PARM7, PARM8, PARM9, PARM10, PARM11, PARM12, PARM13, PARM14, PARM15, PARM16, PARM17, PARM18, PARM19, PARM20, PARM21, PARM22, PARM23, PARM24, PARM25, PARM26, PARM27, PARM28, PARM29, PARM30, VERSION
- DSNOFX02 - SCHEMA, SPECIFICNAME, ROUTINETYPE, VERSION
- DSNOFX03 - NAME, SCHEMA, CAST_FUNCTION, PARM_COUNT,PARM_SIGNATURE, PARM1
- DSNOFX04 - ROUTINE_ID
- DSNOFX05 - SOURCESCHEMA, SOURCESPECIFIC, ROUTINETYPE
- DSNOFX06 - SCHEMA, NAME, ROUTINETYPE, PARM_COUNT
- DSNOFX07 - NAME, PARM_COUNT, ROUTINETYPE, SCHEMA, PARM_SIGNATURE, PARM1, PARM2, PARM3, PARM4, PARM5, PARM6, PARM7, PARM8, PARM9, PARM10, PARM11, PARM12, PARM13, PARM14, PARM15, PARM16, PARM17, PARM18, PARM19, PARM20, PARM21, PARM22, PARM23, PARM24, PARM25, PARM26, PARM27, PARM28, PARM29, PARM30, VERSION
- DSNOFX08 - JARSCHEMA, JAR_ID

Typical access to the table

Typical access to this table of interest to us is by SCHEMA, NAME, and CONTOKEN to locate the specific routine to execute.

Important columns

Some of the important columns to look at are:

- SCHEMA

  Schema of the routine.

- NAME

  The name of the routine.
- CONTOKEN

  These consistency token of the routine.
- COLLID

  The name of the package collection to be used when the routine is executed. A value of blank means the collection ID of the calling program is to be used.

C.1.16 SYSSTMT

This table contains one or more rows for each SQL statement of each DBRM and should become irrelevant after you migrate away from DBRM-based plans.

Indexes on this table

There are no indexes on this table.

Typical access to the table

This table is typically accessed by PLNAME to show the SQL text for all statements in it. Typically, it is ordered by STMTNO and SEQNO. Because the text is stored as varying-length string, it is typically needs to be formatted properly to make it readable.

Important columns

Some of the important columns to look at are:

- NAME

  Name of the DBRM.
- PLNAME

  Name of the application plan.
- STMTNO

  The statement number of the statement in the source program.
- SEQNO

  Sequence number of this row with respect to a statement of the DBRM. The numbering starts with zero.
- TEXT

  Text or portion of the text of the SQL statement.
- ACCESSPATH

  For static statements, indicates if the access path for the statement is based on user-specified optimization hints. A value of ‘H’ indicates that optimization hints were used.
- QUERYNO

  The query number of the SQL statement in the source program.

C.1.17 SYSTRIGGERS

This table contains a row for each trigger.

Indexes on this table

The indexes on this table are:

- DSNOTX01 - SCHEMA, NAME, SEQNO
- DSNOTX02 - TBOWNER, TBNAME
- DSNOTX03 - SCHEMA, TRIGNAME

Typical access to the table

This table is typically accessed by SCHEMA and NAME to view information about the trigger such as granularity, activation time, and so on.

Important columns

Some of the important columns to look at are:

- SCHEMA

  Schema of the trigger. This is also the COLLID of the package.

- NAME

  Name of the trigger. The trigger package has the same name.

C.1.18 SYSUSERAUTH

This table records the system privileges that are held by users.

Indexes on this table

The indexes on this table are:

- DSNAUH01 - GRANTEE, GRANTEDTS, GRANTEETYPE
- DSNAUX01 - GRANTOR, GRANTORTYPE

Typical access to the table

This table is typically accessed by GRANTEE to see if a privilege (such as BINDADD authority) is held by a user. Alternatively, it could be used to see all users who have a certain privilege.

Important columns

Some of the important columns to look at are:

- GRANTOR

  Authorization ID of the user who granted the privileges.

- GRANTEE

  Authorization ID of the user that holds the privilege.

- BINDADDAUTH

  Whether the GRANTEE can use the BIND subcommand with the ADD option. Appendix B.1, “DSNZPARMs” on page 278 explains some DSNZPARMs affect this behavior.

- BINDAGENTAUTH

  Whether the GRANTEE has BINDAGENT privilege to bind packages with the grantor as the owner of the package.

Note: While BINDAGENT appears to be a system-wide privilege, the user is limited to binding packages only to the specific grantor. For example, to be able to bind TEST and PROD packages, the user must possess 2 separate grants as BINDAGENT: one issued by TEST and one issued by PROD. This important fact is commonly misunderstood.

- GRANTEETYPE

  Indicates the type of grantee (authorization ID or role).
- GRANTORTYPE

  Indicates the type of grantor (authorization ID or role).
- SYSADMAUTH

  Indicates whether the grantee has SYSADM privilege.
- SYSCTRLAUTH

  Indicates whether the grantee has SYSCTRL privilege.

C.2 EXPLAIN tables

We provide information about the base PLAN_TABLE and extended EXPLAIN tables which you should use to analyze the access paths for queries in your packages. They are used by optimization tools such as IBM Optimization Service Center for DB2 for z/OS as well as by the EXPLAIN function.

If the PLAN_TABLE does not exist and you bind a package with EXPLAIN(YES), the bind is unsuccessful and returns SQLCODE -219. The extended EXPLAIN table are optional (and you can create only those you need).

- PLAN_TABLE
- DSN_FUNCTION_TABLE
- DSN_STATEMNT_TABLE
- DSN_DETCOST_TABLE
- DSN_FILTER_TABLE
- DSN_PGRANGE_TABLE
- DSN_PGROUP_TABLE
- DSN_PREDICAT_TABLE
- DSN_PTASK_TABLE
- DSN_SORT_TABLE
- DSN_SORTKEY_TABLE
- DSN_STRUCT_TABLE
- DSN_VIEWREF_TABLE

C.2.1 Base PLAN_TABLE

The base PLAN_TABLE is always populated when a package is bound with EXPLAIN(YES) option. The extended EXPLAIN tables are also populated if present and they are also used by optimization tools. Refer to *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851, for details on the current format of the PLAN_TABLE and the CREATE statement you need to issue.

Your existing instances of PLAN_TABLE, especially those created in earlier version of DB2 can use many other formats with fewer columns. However, you should use the 59-column format because it gives you the most information.

The various supported formats include:

- V2.1 - 25-column format
- V2.2 - 28-column format
- V2.3 - 30-column format
- V3 - 34-column format
- V4 - 43-column format
- V5 - 46-column format
- V6 - 49-column format
- V7 - 51-column format
- V8 - 58-column format
- V9 - 59-column format

The recommended and current format is the 59-column format. The format MUST conform to one of those listed previously.

Indexes on this table

The indexes on this table are:

- QUERYNO, APPLNAME, PROGNAME, VERSION, COLLID, OPTHINT when using a hint.
- COLLID, PROGNAME, QUERYNO, QBLOCKNO, PLANNO for access path analysis.

Important columns

Some of the important columns to look at are:

- COLLID

  The collection ID for the package.

- PROGNAME

  The name of the program or package containing the statement being explained.

- QUERYNO

  A number that identifies the statement that is being explained.

- QBLOCKNO

  A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.

- PLANNO

  The number of the step in which the query that is indicated in QBLOCKNO was processed. This column indicates the order in which the steps were executed.

- METHOD

  A number that indicates the join method that is used for the step.

- TNAME

  The name of a table, materialized query table, created or declared temporary table, materialized view, or materialized table expression.

- ACCESSTYPE

  The method of accessing the new table.

- ACCESSNAME

For ACCESSTYPE I, I1, N, MX, or DX, the name of the index; for ACCESSTYPE P, DSNPJW (mixopseqno) is the starting pair-wise join leg in MIXOPSEQNO; otherwise, blank.

- MATCHCOLS

  For ACCESSTYPE I, I1, N, MX, or DX, the number of index keys that are used in an index scan; otherwise, 0.

- INDEXONLY

  Indication of whether access to an index alone is enough to perform the step, or Indication of whether data too must be accessed.

- PREFETCH

  Indication of whether data pages are to be read in advance by prefetch.

- SORTN_UNIQ, SORTN_JOIN, SORTN_ORDERBY, SORTN_GROUPBY

  Indicators for new table sorted for uniqueness, join, order by or group by.

- SORTC_UNIQ, SORTC_JOIN, SORTC_ORDERBY, SORTC_GROUPBY

  Indicators for composite table sorted for uniqueness, join, order by or group by.

Tip: See A.22, “Selecting only the most useful EXPLAIN information” on page 275 for a query to obtain the information commonly needed for the latest bind for a package.

C.2.2 Extended EXPLAIN tables

See Refer to *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851, for the CREATE statement you need to issue for these tables.

DSN_FUNCTION_TABLE

You can use DSN_FUNCTION_TABLE to collect information about user-defined functions that are referenced in SQL statements.

DSN_STATEMNT_TABLE

You can use DSN_STATEMNT_TABLE to collect information about the estimated cost of a statement

DSN_DETCOST_TABLE

The detailed cost table, DSN_DETCOST_TABLE, contains information about detailed cost estimation of the mini-plans in a query.

DSN_FILTER_TABLE

The filter table, DSN_FILTER_TABLE, contains information about how predicates are used during query processing.

DSN_PGRANGE_TABLE

The page range table, DSN_PGRANGE_TABLE, contains information about qualified partitions for all page range scans in a query.

DSN_PGROUP_TABLE

The parallel group table, DSN_PGROUP_TABLE, contains information about the parallel groups in a query.

DSN_PREDICAT_TABLE

The predicate table, DSN_PREDICAT_TABLE, contains information about all of the predicates in a query.

DSN_PTASK_TABLE

The parallel tasks table, DSN_PTASK_TABLE, contains information about all of the parallel tasks in a query.

DSN_SORT_TABLE

The sort table, DSN_SORT_TABLE, contains information about the sort operations required by a query.

DSN_STRUCT_TABLE

The structure table, DSN_STRUCT_TABLE, contains information about all of the query blocks in a query.

DSN_SORTKEY_TABLE

The sort key table, DSN_SORTKEY_TABLE, contains information about sort keys for all of the sorts required by a query.

DSN_VIEWREF_TABLE

The view reference table, DSN_VIEWREF_TABLE, contains information about all of the views and materialized query tables that are used to process a query.

D

Sample test environment

In this appendix we detail the sample environment that is used throughout this project for all examples.

This appendix covers the following topics:

- Hardware and software
- Sample database environment

D.1 Hardware and software

We have used a sysplex complex in an LPAR with 2094 z9® with 2 shared CPs and 4 GB of real storage.

The OS level is z/OS V1R10.

The DB2 9 for z/OS has been updated at level D08100 at the start of our project. We have added more recent maintenance as listed in the book.

The subsystem ID is DB9A.

D.2 Sample database environment

Throughout our samples we made the following assumptions related to the application environment and the DB2 objects nomenclature:

- We use two applications called FIN (Financial Stuff) and PAY (Payroll Processing).
- Each application has three environments: DEVT, TEST, and PROD.
- Each development environment can distinguish between code intended for batch, CICS, routines (stored procedures, user defined functions, triggers, and so on).

We use the FIN application primarily, with PAY being an application that we can interface to. A realistic application configuration is more complex and flexible.

Table D-1 shows the DB2 objects naming standards intended to allow examples to consistently illustrate the practical usage of packages.

Table D-1 Our naming standards

Object Type	Naming standard	Notes
Databases	FINeeee	‘eeee’ represents Environment
Table spaces	TSnn	TS01, TS02, TS03, and so on
Table Creators	FINeeee	‘eeee’ represents Environment
Plans	FINeeeex	‘eeee’ represents Environment, ‘x’ represents Usage (for example, (C)ICS or (B)atch
Collections	FIN_eeee_xxxxx	‘eeee’ represents Environment, ‘xxxxx’ represent Usage (BATCH, CICS, ROUTINES, COMMON, and so on)
Programs / Packages	PGMnn	PGM01, PGM02, PGM03, and so on
RACF Group: Developers	FINeeeeD	Access like BIND and Select / Insert / Update / Delete is granted to the developers so they can bind and test.
RACF group: Production control	FINeeeeP	Access is given only to execute Plans.

Index

Numerics

A

B

C

D

E

F

G

S

T

U

V

W

X

Z

DB2 9 for z/OS: Packages Revisited

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

DB2 9 for z/OS: Packages Revisited

Learn all about packages, collections, plans

Manage local and remote packages

Understand and maintain access plan stability

DB2 packages were introduced with DB2 V2.3 in 1993. During the 15 years that have elapsed, a lot has changed. In particular, there is a more widespread use of distributed computing, Java language, new tools, and upgrades in the platform software and hardware. The best practices back then just might not be optimal today. In this IBM Redbooks publication, we take a fresh look at bringing packages into the 21st century.

We begin with an overview of packages and explain the advantages of using packages. Because database request module (DBRM) based plans have been deprecated in DB2 9, you need to convert to packages if you did not use packages already. We provide guidance on using a DB2 provided function to convert from DBRM-based plans to packages.

We re-examine the application development frameworks for packages: program preparation, package setup, and execution. For distributed applications, we include a discussion of a utility to identify and remove deprecated private protocol and converting to DRDA as well as an introduction to the new pureQuery function of Data Studio. We also discuss common problems and their resolutions.

We then explore administration and operational activities dealing with packages, including security, access path management (where we discuss the newly introduced package stability feature to allow for a seamless fallback), and management and performance aspects.

The appendixes include useful queries and mention tools for managing packages effectively.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7688-00 ISBN 0738432369