DB2 Packages: Implementation and Use

Document Number GG24-4001-00

October 1993

International Technical Support Center
San Jose

Take Note!

Before using this information and the product it supports, be sure to read the general information under “Special Notices” on page xiii.

First Edition (October 1993)

This edition applies to DB2 Version 3, Program Number 5685-DB2.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSC Technical Bulletin Evaluation Form for readers' feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Center
Dept. 471, Building 098
5600 Cottle Road
San Jose, California 95193-0001

Abstract

This document is a comprehensive guide to the new package function brought by DB2 V2.3. It considers both the distributed and nondistributed environments. It provides a detailed explanation of the BIND PACKAGE function, including the use of collections and versioning, recommendations on how to implement packages, and discussion of the related performance issues. The document also provides working examples on possible package management solutions.

This document is intended for customers and technical professionals who need to know how to take advantage of the new package function and to administer packages. A knowledge of DB2 V2.3, DB2 V3, and Distributed Relational Database Architecture (DRDA) connectivity is assumed.

DS (177 pages)

Contents

Figures

Tables

Special Notices

This publication is intended to help customers and technical professionals understand and evaluate the package function and package management made available with DB2 V2.3. The information in this publication is not intended as the specification of any programming interfaces that are provided by DB2 V2.3 or DB2 V3. See the PUBLICATIONS section of the IBM Programming Announcement for the DB2 product for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AD/Cycle
AS/400
CICS
DB2
DDCS/2
Distributed Relational Database Architecture
DRDA

IBM
IMS/ESA
OS/2
OS/400
QMF
SAA
SQL/DS
SQL/400
VM/ESA
VTAM

The following terms, which are denoted by a double asterisk (**) in this publication, are trademarks of other companies:

Candle is a trademark of the Candle Corporation.
!DB/EXPLAIN is a trademark of Candle Corporation.
PLATINUM Package/It and PLATINUM Plan Analyzer
are registered trademarks of PLATINUM technology, inc.

Preface

This document provides detailed explanations of the usage and management of DB2 packages and discusses performance issues.

The book is written for database administrators, application programmers, and users who are involved in package use and administration and would like to understand and evaluate DB2 packages.

The document is organized as follows:

- Chapter 1, “Package Overview”

  This chapter provides an overview of packages. It outlines the program preparation process before and after DB2 V2.3 and discusses the terminology and benefits associated with packages as well as the authorizations required for administering and using packages.

- Chapter 2, “Package Processing”

  Before using packages, it is important to understand how DB2 handles them at execution time. This chapter discusses the processing of packages in the EDM pool and the search algorithm that is used in locating them at execution time.

- Chapter 3, “Using Collections”

  DB2 collections provide a method for logically grouping packages. This chapter discusses the various ways of grouping packages into collections and provides some examples of application promotion and versioning scenarios using collections in an application development environment. The chapter also discusses how to use collections to support mirror tables and testing of a new package using a try-out collection.

- Chapter 4, “Using the VERSION Keyword”

  This chapter describes the concept of program versioning using the precompiler VERSION keyword, which is specific to the DB2 environment, and shows some examples of application promotion scenarios that implement this new function.

- Chapter 5, “Distributed DB2 Packages”

  This chapter discusses how a DB2 package is created and used in a distributed database environment where the application server is DB2 and the application requester is either a DB2, SQL/DS, OS/2, or OS/400 database manager. The chapter covers the remote BIND issued at the requester site, restrictions of the remote bind related to each type of application requester, authorizations for binding and executing remote packages, and DB2 package name resolution at execution time.

- Chapter 6, “Implementing Packages”

  This chapter discusses the factors that should be considered in planning for and migrating to the use of packages and includes

migration planning, modification of existing plans to use packages, getting started with packages, and package administration.

- Chapter 7, “Performance Implications”

  This chapter discusses the performance considerations regarding package list search and EDM pool storage use. The chapter also describes the monitoring of packages and their imbedded SQL statements.

Related Publications

The publications listed below are considered particularly suitable for a more detailed discussion of the topics covered in this document.

DB2 Documentation

DB2 General Information, GC26-4373

DB2 Administration Guide, SC26-4374

DB2 Licensed Program Specification, GC26-4375

DB2 SQL Reference, SC26-4380

DB2 Application Programming and SQL Guide, SC26-4377

DB2 Command and Utility Reference, SC26-4378

DB2 Messages and Codes, SC26-4379

DB2 Master Index, GC26-4772

DB2 Reference Summary, SX26-3771

DB2 Diagnosis Guide and Reference, LY27-9536, available to IBM licensed customers only.

SQL/DS Documentation

SQL/DS General Information, GH09-8074

SQL/DS Licensed Program Specifications, GH09-8076

SQL/DS Operation, SH09-8080

SQL/DS Database Administration, GH09-8083

SQL/DS System Administration, GH09-8084

SQL/DS Managing SQL/DS, SH09-8077

SQL/DS Installation, GH09-8078

SQL/DS Application Programming, SH09-8086

SQL/DS Database Services Utility, SH09-8088

SQL/DS SQL Reference, SH09-8087

SQL/DS Interactive SQL Guide and Reference, SH09-8085

SQL/DS Master Index, SH09-8089

SQL/DS Reference Summary, SX09-1173

SQL/DS Diagnosis Guide and Reference, LH09-8081

SQL/DS Messages and Codes, SH09-8079

AS/400 Documentation

OS/400 Distributed Relational Database Guide, SC41-0025

OS/400 Question-and-Answer Database Coordinator's Guide, SC41-8086

OS/400 Database Guide, SC41-9659

OS/400 Distributed Data Management Guide, SC41-9600

SAA Structured Query Language/400 Programmer's Guide, SC41-9609

SAA Structured Query Language/400 Reference, SC41-9608

OS/400 System Concepts, GC41-9802

AS/400 Basic Security Guide, SC41-0047

AS/400 Security Reference, SC41-8083

AS/400 Communications Management Guide, SC41-0024

AS/400 Work Management Guide, SC41-8078

AS/400 Local Area Network Guide, SC41-0004

AS/400 APPN Guide, SC41-8188

AS/400 APPC Programmer's Guide, SC41-8189

AS/400 Network Planning Guide, GC41-9861

AS/400 Communications Configuration Reference, SC41-0001

Extended Services for OS/2 Documentation

IBM Extended Services for OS/2, G326-0161

IBM Extended Services for OS/2 Messages and Error Recovery Guide, SO4G-1017

IBM Extended Services for OS/2 Communications Manager APPC Programming Reference, S04G-1025

IBM Extended Services for OS/2 Command Reference, S04G-1020

IBM Extended Services for OS/2 Communications Manager Configuration Guide, S04G-1002

IBM Extended Services for OS/2 Database Manager Programming Guide and Reference, S04G-1022

IBM Extended Services for OS/2 Guide to Database Manager, S04G-1013

IBM Extended Services for OS/2 Guide to Database Manager Client Application Enablers, S04G-1114

IBM Extended Services for OS/2 Guide to User Profile Management, S04G-1112

IBM Extended Services for OS/2 Hardware and Software Reference, S04G-1017

IBM Extended Services Structured Query Language (SQL) Reference, S04G-1012

DB2/2 Documentation

DATABASE 2 OS/2 Guide, S62G-3663

DATABASE 2 OS/2 Command Reference, S62G-3670

DATABASE 2 OS/2 Information and Planning Guide, S62G-3662

DATABASE 2 OS/2 Installation Guide, S62G-3664

DATABASE 2 OS/2 Master Index and Glossary, S62G-3669

DATABASE 2 OS/2 Messages and Problem Determination Guide, S62G-3668

DATABASE 2 OS/2 Programming Guide, S62G-3665

DATABASE 2 OS/2 Programming Reference, S62G-3666

DATABASE 2 OS/2 SQL Reference, S62G-3667

International Technical Support Center Publications

Introduction to Relational Data, GG24-3300

DB2 V2.3 Nondistributed Performance Topics, GG24-3823

DB2 Distributed Database Application Implementation and Installation Primer, GG24-3400

Distributed Relational Database - Application Scenarios, GG24-3513

Distributed Relational Database Planning and Design Guide for DB2 Users, GG24-3755

Distributed Relational Database Remote Unit of Work Implementation DB2-DB2 and DB2-SQL/DS Vol. I, GG24-3715

Distributed Relational Database Remote Unit of Work Implementation DB2-DB2 and DB2-SQL/DS Vol. II, GG24-3716

Distributed Relational Database: Using OS/2 DRDA Client Support with DB2, GG24-3771

OS/2 DDCS/2 and DB2 V2.3 Distributed Relational Database Performance: Early Experience, GG24-3926

Structured Query Language/400: A Guide for Implementing OS/400 Version 2 Release 2.0, GG24-3321-3.

Additional Publications

DB2 Package Bind, TR 03.453

Formal Register of Extensions and Differences in SQL, SC26-3316

Acknowledgments

This document is the result of a residency project conducted at the International Technical Support Center-San Jose in the fourth quarter of 1992.

The project was designed and managed by:

- Viviane Anavi-Chaput, DB2 Assignee, ITSC-San Jose.

We want to acknowledge the excellent work of:

- Terry Gallagher IBM Australia
- Anthony Kwang June Wen IBM Singapore
- Mika Suzuki IBM Japan

Special thanks are due to the following people for the invaluable advice and guidance they provided in the production of this document:

- Linnette Bakow IBM Development, Santa Teresa
- Corinne Baragoin IBM France
- Yann Palomba CNRO, France
- Curt Cotner IBM Development, Santa Teresa
- William Favero IBM Skills Dynamics, Dallas
- Sue Greenlee IBM Development, Santa Teresa
- Josette Huang IBM Development, Santa Teresa
- Ravi Kumar IBM Australian Programming Center
- Katy Lee IBM Development, Santa Teresa
- Stephen Lindsdell IBM Australia
- Edwin Ma IBM Development, Toronto
- Maureen McDevitt IBM Development, Santa Teresa
- Roger Miller IBM Development, Santa Teresa
- Barbara Olson IBM Development, Santa Teresa
- Christian Peterhans IBM ITSC, San Jose
- Akira Shibamiya IBM Development, Santa Teresa
- Peter Shum IBM Development, Toronto.
- Roy Smith IBM Development, Santa Teresa
- Bart Steegmans Belgische Boerenbond, Belgium
- Jacky Wielandts IBM Belgium
- Jerry Wong IBM Development, Santa Teresa

Thanks also to Maggie Cutler for her editorial assistance.

Viviane Anavi-Chaput, Project Leader

International Technical Support Center-San Jose
October 1993

Chapter 1. Package Overview

Packages, a new bind enhancement introduced in DB2* V2.3, provide the binding flexibility customers have been demanding. In a distributed environment with a database management system supporting the Distributed Relational Database Architecture* (DRDA*), such as SQL/DS* or OS/2* Database Manager (DBM), it is mandatory to use packages. In a nondistributed environment, using packages is not mandatory for applications. However, packages provide answers to many of the design, application development, and maintenance problems associated with earlier releases. Use of packages requires that users fully understand the options available with packages and assess the applicability of each option to their own environment. Use of packages implies careful choices and management.

This chapter provides an overview of packages. We outline the program preparation process before and after DB2 V2.3, discuss the package identifiers and the terminology associated with packages, review the benefits associated with packages, and summarize the authorizations required for administering and using packages.

1.1 Program Preparation Process

The implementation of packages has an impact on the program preparation process. In this section we discuss the changes that occurred in the DB2 program preparation process. We also position packages in the program preparation process.

1.1.1 Before DB2 V2.3

You should already be familiar with the program preparation process before DB2 V2.3 as depicted in Figure 1 on page 2.

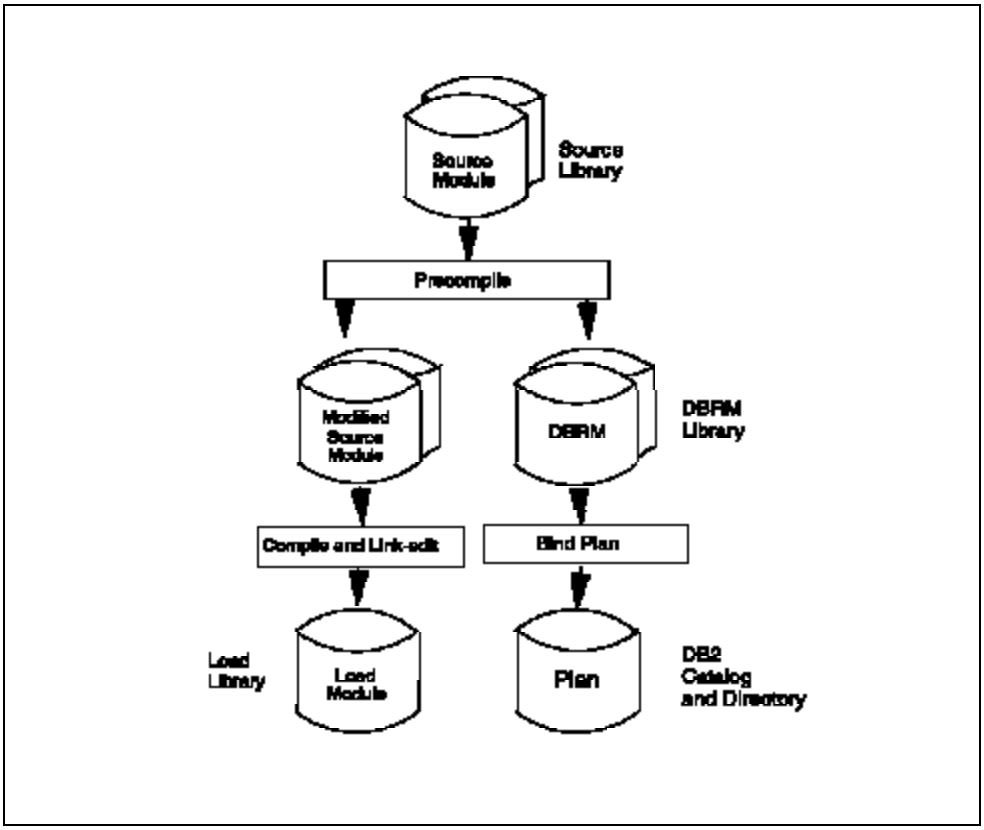

Figure 1. DB2 Program Preparation Process before DB2 V2.3

When the source code is ready, the first step is to precompile the program with the DB2 precompiler. The DB2 precompiler generates two different data sets: the modified source module and the Database Request Module (DBRM). 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 EXEC 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 to check at run time that the two data sets originally came from the same source program. This consistency token is passed to the load module from the modified source module in the compiling and link-editing process.

The BIND process transforms the DBRM into DB2 interpretable form. The output of BIND is called a plan. A consistency token is copied from the DBRM to the plan at this point. At execution time, this consistency token is checked against the consistency token of the load module.

In the bind plan process, multiple DBRMs can be included in one plan. Because one DBRM consists of one program, a plan can contain information on SQL statements for multiple programs. When a program is a common routine shared by multiple applications, one DBRM is bound into multiple plans.

One of the biggest problems with the many-to-many relationship between DBRMs and plans was that, whenever a program modification occurred, the whole plan containing the DBRM of the modified program had to be bound

again. Binding a plan with multiple DBRMs can be time-consuming and resource intensive because DB2 optimizes all the SQL statements included in all the DBRMs of a plan, even those in an unmodified DBRM. Figure 2 on page 3 illustrates this mechanism.

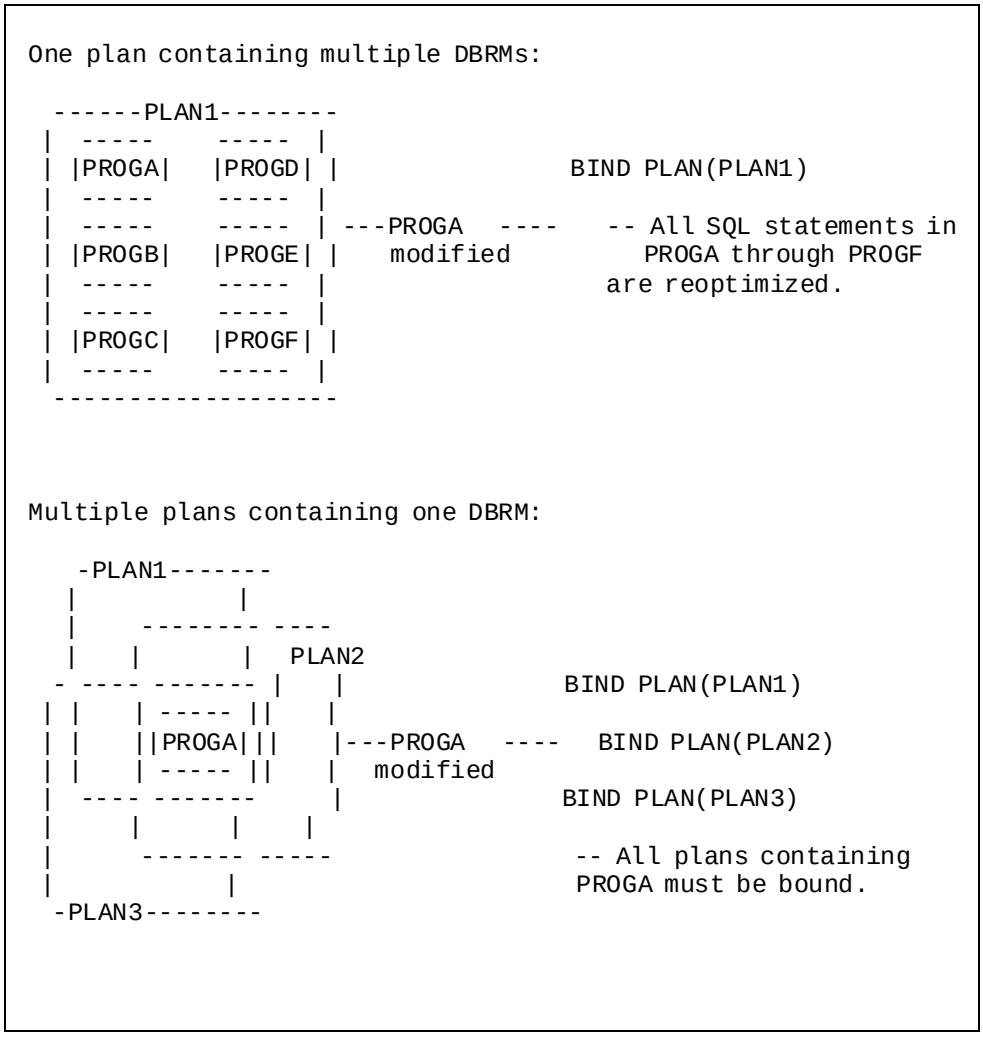

Figure 2. Many-to-Many Relationship between a Plan and the DBRM

1.1.2 After DB2 V2.3

DB2 V2.3 introduces binding a package into the program preparation process. Refer to Figure 3 on page 4 for an overall view of the process.

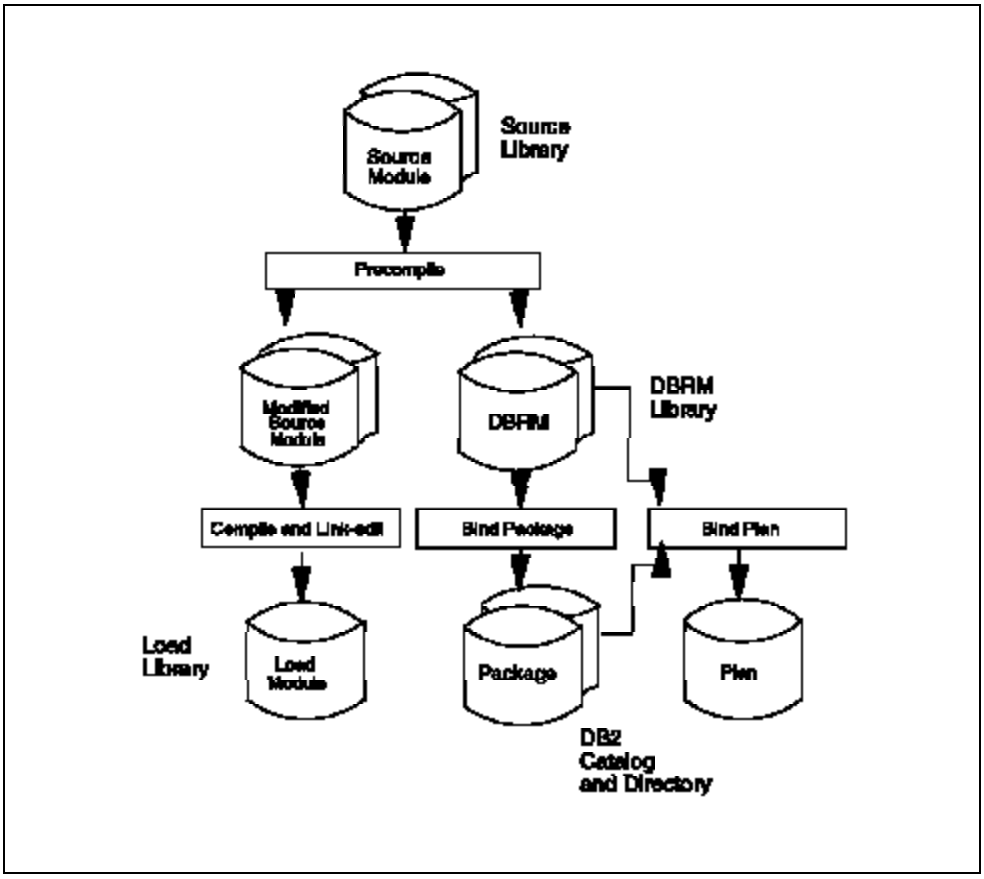

Figure 3. DB2 Program Preparation Process after DB2 V2.3

Instead of binding a DBRM into a plan, one can bind it into a unit called a package. The difference between binding a DBRM into a plan and into a package is that a package is always bound from a single DBRM, whereas a plan can be bound from multiple DBRMs. Packages and their information are stored in the DB2 directory and catalog just as plans are after they are bound.

Here is an example of binding a package:

```
BIND PACKAGE(COL1) OWNER(USER1) QUALIFIER(USER2)      -
    MEMBER(PARTINQ) LIBRARY(PD.CURR.DBRMLIB)         -
    SQLERROR(NOPACKAGE) VALIDATE(BIND)               -
    ISOLATION(CS) RELEASE(COMMIT) EXPLAIN(NO)        -
    ACTION(REPLACE) ENABLE(*)
```

The process of binding a package replaces a part of the bind plan process, which does not mean that binding the plan is no longer necessary. With the introduction of packages, the concept of a plan has been enhanced to a flexible and dynamic construct.

Plans can still have one or more DBRMs directly bound into them. However, with the implementation of packages, a plan can include, in addition to the DBRMs, a list of packages called PKLIST (package list). DB2 can access packages only through this package list. When binding a plan

that includes only a package list, DB2 does not go through an optimization process for the packages listed because optimization has already been done at bind package time. It is possible to have both DBRMs and packages included in one plan. In that case, optimization is done only for DBRMs directly bound into the plan at bind plan time. Figure 4 illustrates three different types of plans, one with DBRMs only, one with both DBRMs and packages, and one with packages only.

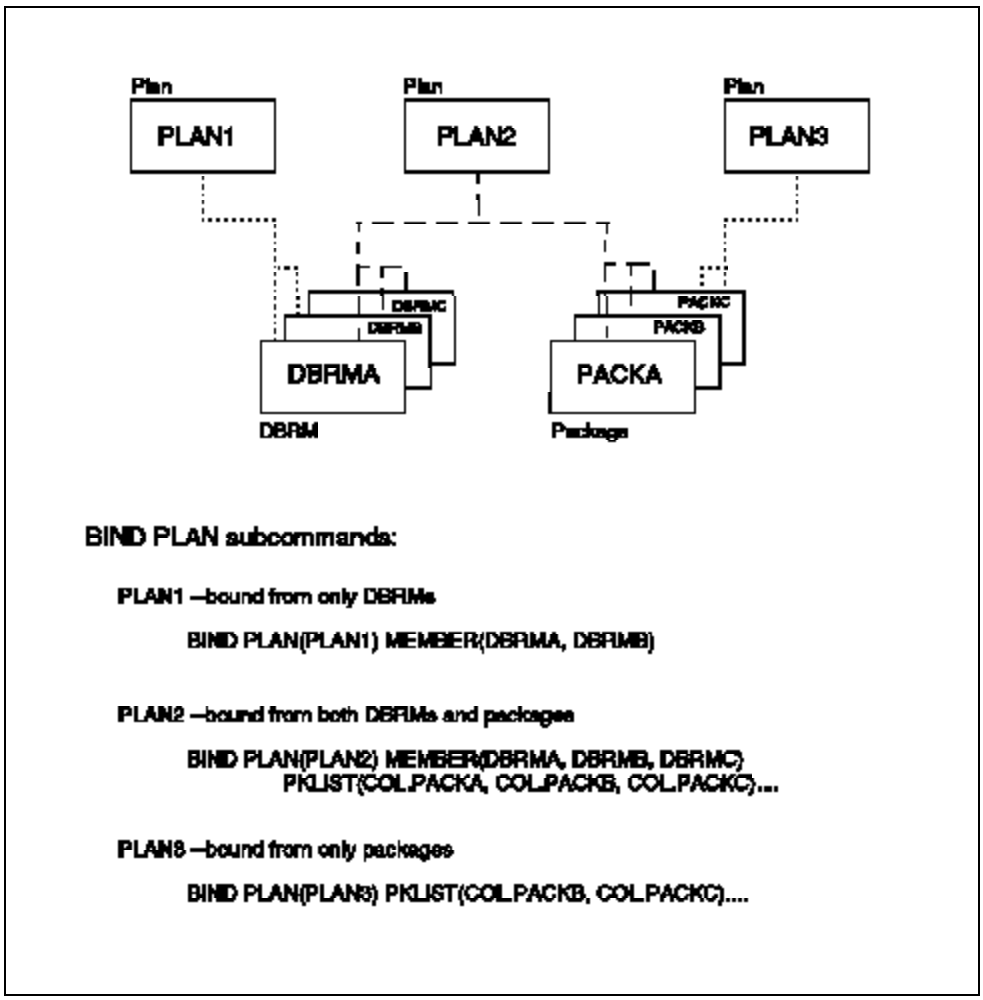

Figure 4. Plan, DBRM, and Package. One plan can include multiple DBRMs and multiple packages. One DBRM or one package can be included in multiple plans.

1.1.3 Changes in the Precompiler

The DB2 precompile option has been enhanced. The VERSION keyword is available to specify the program version being precompiled; it supports multiple versions of packages for one program. For a detailed description of the VERSION keyword, see Section 1.2.4, “Version_id” on page 7.

1.2 Package Name and Its Attributes

This section describes the components of a package name and its attributes.

A package has a three-part name:

location_name.collection_id.package_id.

A package has two attributes: version_id and consistency token.

The qualifier or attribute you specify for each package-related subcommand, such as BIND, REBIND, and FREE, differs for each subcommand. (Consistency token cannot be specified for any of these subcommands.) For example, when binding packages, you cannot specify the package_id and version_id you will be creating, as they will be inherited from the DBRM bound to the package or copied from another package. When freeing packages, however, you can specify the package_id and version_id:

```
BIND   PACKAGE(location-name.collection-id)  MEMBER(dbrmmember)
REBIND PACKAGE(location-name.collection-id.package-id.(version-id))
FREE   PACKAGE(location-name.collection-id.package-id.(version-id))
```

1.2.1 Location_name

The location_name is a location identifier that designates the database management system (DBMS) in which the package is stored. The default is the local DB2.

You can bind a package to a remote site by specifying the location_name as the location of the remote DBMS.

1.2.2 Collection_id

The collection_id is a qualifier that denotes a logical grouping of packages. When binding a package, you must always specify the collection_id (see Figure 5).

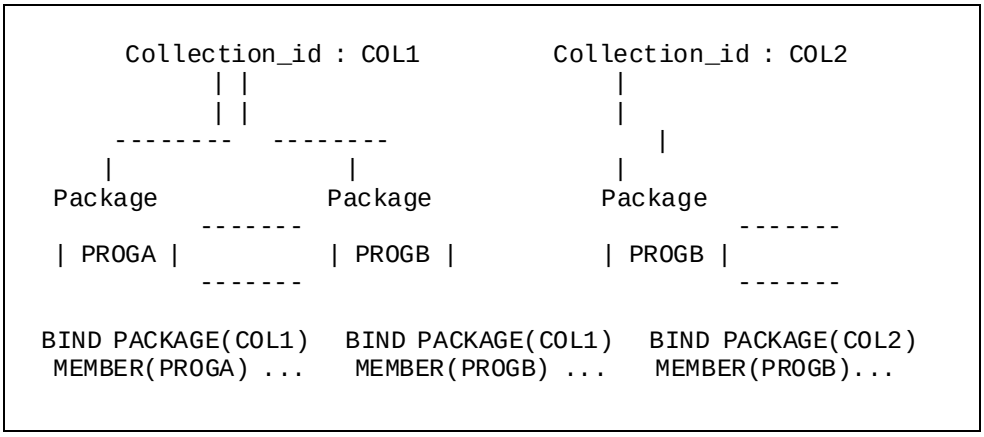

Figure 5. Collection_id

A collection is also a unit of authorization control because it is considered as one DB2 resource. Here are two examples of collection authorization control:

- Controlling which collection_id can be used when creating a new package
- Allowing the execution of all packages in a certain collection.

Additional considerations for using collections are discussed in Chapter 3, “Using Collections” on page 43.

1.2.3 Package_id

The package_id is a qualifier that is automatically generated from the name of the DBRM input at bind package time. Thus, you could say that the package_id can actually be specified as the DBRM name at precompile time. Because the DBRM name is usually the same as the program name, the package_id is equal to the program name in most cases. This makes packages easier to manage than plans. Before the advent of packages, the plan name was not always the same as the program name because, in many cases, plans included multiple programs.

1.2.4 Version_id

By using the package attribute, version_id, you can keep multiple versions of packages with the same package name, in the same collection. For example:

```
LOC1.COL1.PROGA.V1R1
LOC1.COL1.PROGA.V2R1
LOC1.COL1.PROGA.TEST
```

The version_id is set at precompile time by specifying the precompiler option VERSION. When VERSION is not specified, the version_id will be an empty string. It cannot be changed at bind time. In the precompilation process, the version_id is embedded in both the load module and the DBRM. When bound, a package inherits the version_id from the DBRM.

You can either specify the version_id based on your naming convention at precompile time or use VERSION(AUTO). Use of VERSION(AUTO) causes the consistency token to be used as the version_id, and thus users do not have to change the version_id for each precompilation. The version_id can be considered a synonym or external representation of the consistency token with VERSION(AUTO).

Additional considerations for using the version_id are discussed in Chapter 4, “Using the VERSION Keyword” on page 69.

1.2.5 Consistency Token

Consistency tokens are used by DB2 during package search. They ensure package uniqueness because DB2 maintains the integrity of the executing load module and package by their consistency tokens.

The consistency token is embedded in the modified source module and the DBRM at precompile time, and it is propagated to the load module and the package for integrity checking by DB2 at execution time.

The consistency token is a DB2 internal time stamp unless the LEVEL keyword is used at precompile time. If you use the LEVEL keyword, the consistency token becomes the value you specify in that keyword. Hence, using the LEVEL keyword enables you to control changes in the consistency token. (The words LEVEL and VERSION are sometimes confusing. LEVEL is used to control the consistency token, whereas VERSION is used to keep multiple instances of the same package name.)

The load module being executed has the version_id and consistency token embedded at precompile time. The package also has the version_id and consistency token embedded. When there are multiple versions of a package, it is the consistency token, not the version_id, that DB2 refers to when selecting which version to execute (see Figure 6).

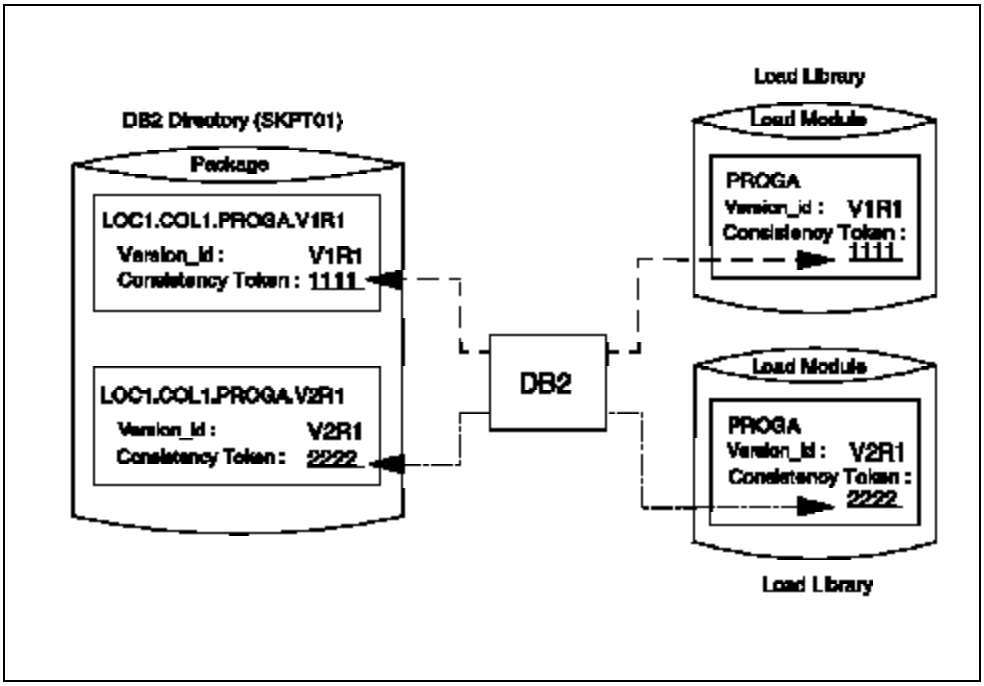

Figure 6. Version_id and Consistency Token

As described so far, packages must be uniquely identified in a given collection either by specifying the version_id or the consistency token as shown in the following example:

location_name.collection_id.package_id.*version_id*

location_name.collection_id.package_id.*consistency token*.

The SYSIBM.SYSPACKAGE has a unique index for each of these criteria (see Figure 7).

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.

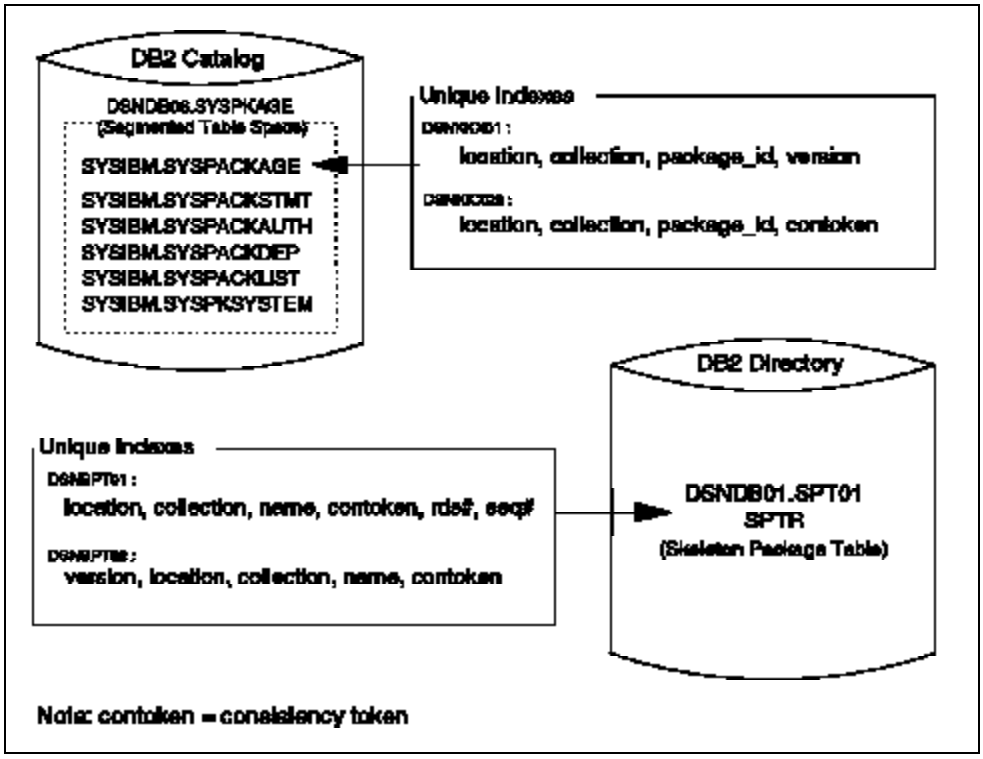

Figure 7. DB2 Catalog and Directory for Packages

1.3 Special Registers

Several new DB2 mechanisms need to be clarified before we go on to discuss bind keywords (see Section 1.4, “Bind Keywords” on page 11). Two special registers, CURRENT SERVER and CURRENT PACKAGESET, have a significant impact on package selection at program execution time.

1.3.1 CURRENT SERVER

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. SERVER has the same meaning as location_name.

The default value of the CURRENT SERVER special register is the local DB2. If the 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.

Figure 8 illustrates two examples of setting the CURRENT SERVER register.

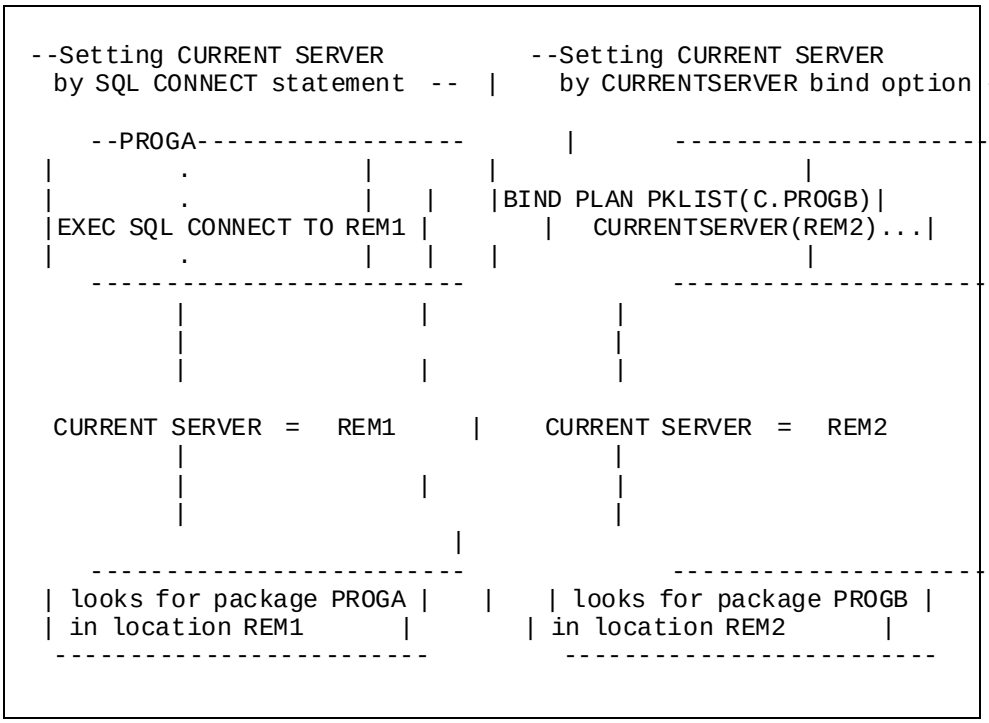

Figure 8. CURRENT SERVER Register and Package Search

1.3.2 CURRENT PACKAGESET

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. PACKAGESET stands for collection_id.

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 register is blank, DB2 first searches for the appropriate executable DBRMs directly bound into a plan and then goes through the packages listed in PKLIST. However, when the value in the CURRENT PACKAGESET register is not blank, DB2

searches only for the package list entry in PKLIST that has the collection_id set in this special register, without searching for the DBRMs or other package list entries with a different collection_id. More details on the package search algorithm are given in Section 2.2, “Package List Search” on page 32.

To set the CURRENT PACKAGESET register in your program, use the following SQL statement:

- SET CURRENT PACKAGESET

which will remain active until the next SET CURRENT PACKAGESET statement or the deallocation of the plan. Figure 9 illustrates an example of setting the CURRENT PACKAGESET register.

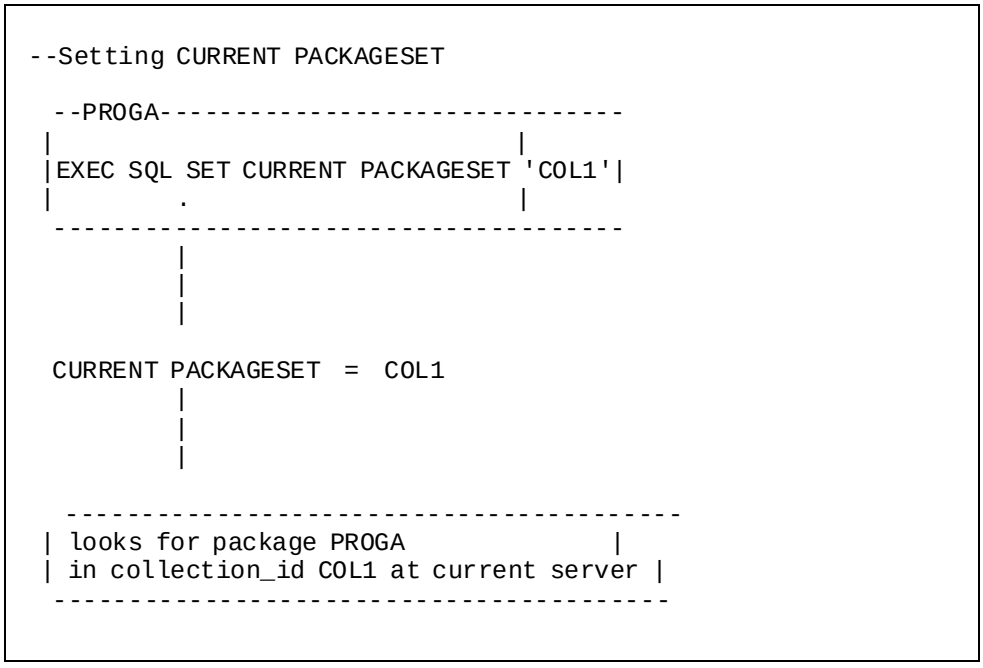

Figure 9. CURRENT PACKAGESET Register and Package Search

1.4 Bind Keywords

This section describes the main keywords used to bind packages and plans. Please refer to the *DB2 Command and Utility Reference* for the BIND subcommand syntax.

1.4.1 PKLIST in BIND PLAN

PKLIST is a list of packages to be included in a plan. Here is an example of a BIND PLAN subcommand.

```
BIND PLAN(planname) PKLIST(loc1.col1.pack1, loc1.col2.*)   .....
```

An asterisk can be used in place of location_name, collection_id, or package_id. Table 1 explains the meaning of the asterisk in each field.

Table 1. Using an Asterisk in PKLIST

	Meaning
*** for location_name (*.col1.pack1)**	Location_name must be supplied at run time by the value in the CURRENT SERVER special register. CURRENT SERVER is set with: • SQL CONNECT statement in the program • CURRENTSERVER option when binding the plan • Default local DB2 location name, or blank if local DB2 has no location name.
*** for collection_id (*.pack1)**	Collection_id must be supplied at run time by the value in the CURRENT PACKAGESET special register. CURRENT PACKAGESET is set with: • SET CURRENT PACKAGESET statement in the program • Default is blank. (In this case you get an error message at run time.)
*** for package_id (col1.*)**	All packages within the specified collection are to be included.

Specifying a PKLIST entry with an asterisk for the package_id eliminates the need to bind a plan again when there is a new package entry in the same collection. The new package entry is automatically included in the plan:

```
BIND PLAN(PLANA) PKLIST(COL1.*) ...
                 |
need to add a new program(NEWPRGM) to 'PLANA'
                 |
BIND PACKAGE(COL1) MEMBER(NEWPRGM)
```

There will be no need to bind the plan again.

1.4.2 QUALIFIER in BIND PLAN and BIND PACKAGE

The value specified in the QUALIFIER keyword (see Figure 10) is the qualifier for all unqualified table names, views, synonyms, and aliases contained in the package. Before DB2 V2.3, the qualifier and the owner of the plan had to be identical, which was restrictive because the owner had to be either the primary or secondary authid of the binder. This QUALIFIER option enables the owner of the package or plan and the qualifier for the object to be different. If QUALIFIER is not present, the OWNER value becomes the qualifier; if OWNER is also not present, the primary authid of the binder becomes the qualifier. QUALIFIER does not have to be the primary or secondary authid of the owner or binder.

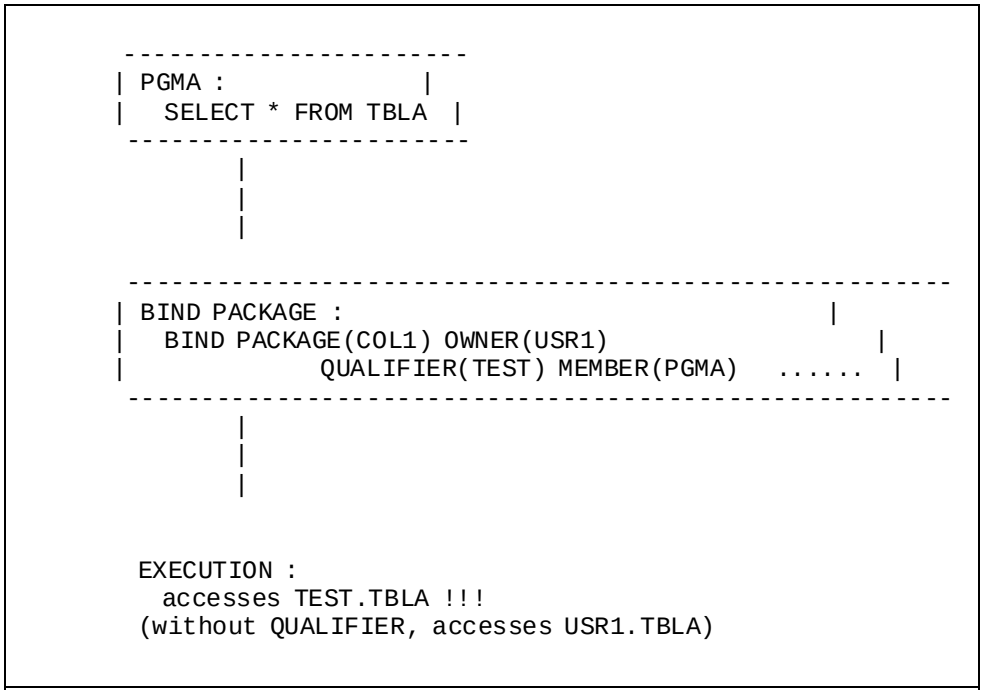

Figure 10. Qualifier Keyword

1.4.3 ENABLE and DISABLE in BIND PLAN and BIND PACKAGE

To help ensure security, the execution of packages and plans can be restricted on certain system connection types. ENABLE allows a package or plan to be executed only on the connection type specified, and DISABLE prohibits a package or plan from being executed on the system connection type specified, thus leaving the other connection types open to execution:

```
BIND PLAN(plana) PKLIST(col1.*) ENABLE(BATCH, DB2CALL) ....
```

or

```
BIND PACKAGE(col1) MEMBER(proga) DISABLE(CICS) CICS(cics1) ....
```

The ENABLE and DISABLE keywords are valid only for local bind operations.

Refer to Section 1.4.7, “Keywords Common to BIND PLAN and BIND PACKAGE” on page 16 to see what happens when there is a conflict between the plan and package ENABLE/DISABLE options.

1.4.4 COPY in BIND PACKAGE

You can make a new package by copying an existing package. Copying packages goes through the full bind process, reading the input from SYSIBM.SYSPACKSTMT instead of the DBRM. Because the copy function is a bind process after all, the access path is optimized for the copied package at bind time. It is recommended that you check, using EXPLAIN, the access path of the package obtained with COPY and compare it to the access path of the original package.

Copied packages have the same bind package options as the original packages. To override the bind options, specify the appropriate keyword when copying. However, remember that 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.

A package can be bound to a local or remote system, but the original package must reside in the local DB2. If you want to copy a package within the same system, the new package must be bound to another collection because both packages will have identical package-ids, version_ids, and consistency tokens.

The COPY feature might enable you to eliminate the use of DBRM libraries, provided that the precompile and BIND PACKAGE process occur in the same job stream and the DBRMs are never bound into a plan. A temporary data set for DBRM can be used to pass the data from precompilation to the BIND PACKAGE process. If that DBRM is required later to make another package, the COPY feature of BIND will do it and avoid binding the package from the DBRM member.

1.4.5 REPLVER in BIND PACKAGE

If the precompiler option VERSION is used, you can specify which version to replace when performing BIND PACKAGE ACTION(REPLACE) by using the REPLVER keyword. The version_id of the replaced package becomes the version_id embedded in the associated DBRM, as Figure 11 on page 15 illustrates.

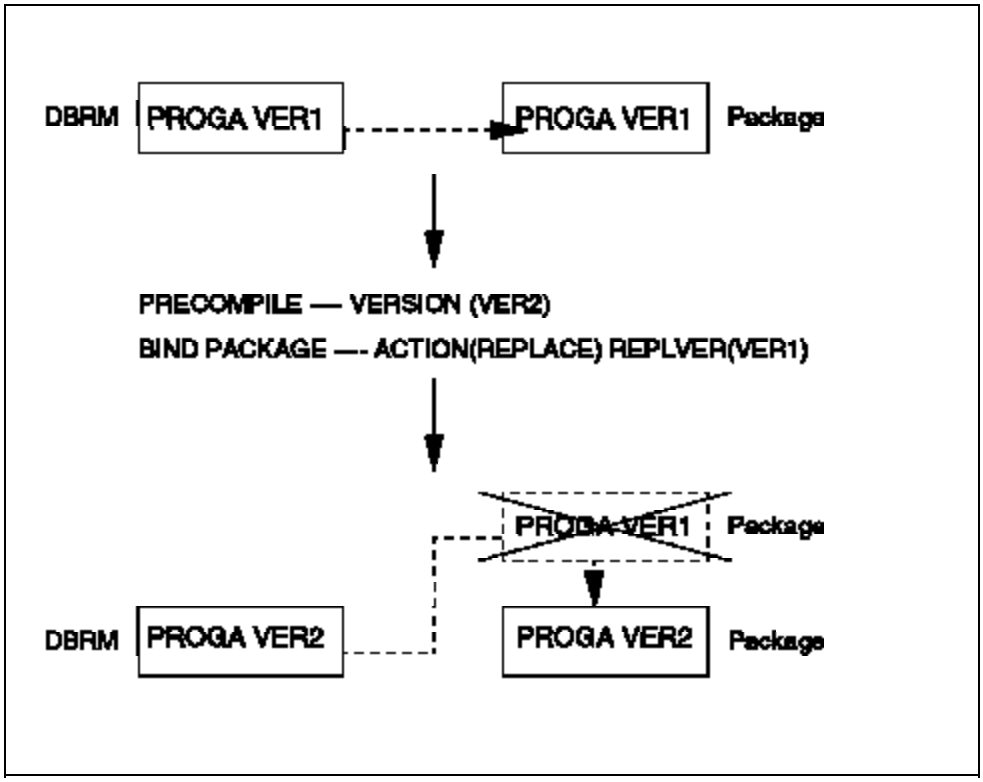

Figure 11. REPLVER Keyword

1.4.6 SQLERROR in BIND PACKAGE

SQLERROR indicates whether or not to create a package if SQL errors are encountered. This keyword is primarily for distributed applications, which are designed to access multiple remote sites. Such programs include syntax that is unique to each unlike environment, thus generating an error when bound at a DB2 site.

The current VALIDATE(RUN/BIND) option does not provide a solution for this. The SQLERROR keyword provides control for handling such errors.

SQLERROR(CONTINUE) indicates that:

- A package is created even if there is an SQL error
- With VALIDATE(BIND)
  - Statements in error are not executable, and an immediate SQL error code is issued when execution is attempted.
- With VALIDATE(RUN)
  - Attempts incremental bind at execution for statement in error (-204 and -555, not for syntax errors)
  - Gives SQL error code if incremental bind fails.

SQLERROR(NOPACKAGE) indicates that BIND fails if there are any SQL errors.

For a detailed description of this keyword, see “SQLERROR” on page 115.

1.4.7 Keywords Common to BIND PLAN and BIND PACKAGE

Several BIND keywords are common to BIND PLAN and BIND PACKAGE subcommands. This section describes the options that are retained at execution time when the BIND PLAN and BIND PACKAGE keywords do not have identical values. For more information on the relationship between package and plan bind options, please refer to the *DB2 Command and Utility Reference* or the *DB2 Application Programming and SQL Guide*.

ENABLE and DISABLE

For an application to run on the DB2 system, the system connection type must be enabled in both package and plan. If the package and plan options differ, the most restrictive option will prevail at execution time (see Table 2).

Table 2. ENABLE and DISABLE Options at Execution Time

		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.

The system connection types that can be specified for these keywords are the same for packages and plans except for the type REMOTE, which can only be specified for packages.

QUALIFIER

The value specified at package bind time is valid. In other words, the qualifier specified when binding a plan has no effect on the package because the package has already been optimized with its qualifier value.

ISOLATION and RELEASE

When the package and plan options differ, the package options take precedence over the plan options. If the package ISOLATION and RELEASE options are not specified, they default to those of the plan. If the plan does not mention the ISOLATION and RELEASE options, the default is ISOLATION(RR) and RELEASE(COMMIT). Table 3 on page 17 summarizes the ISOLATION option retained at execution time.

Table 3. ISOLATION Options at Execution Time				
	Plan			
Package	ISOLATION Option	RR	CS	Blank
	RR	RR	RR	RR
	CS	CS	CS	CS
	Blank	RR	CS	RR

Table 4 summarizes the RELEASE option retained at execution time.

Table 4. RELEASE Options at Execution Time				
	Plan			
Package	Release Option	COMMIT	DEALLOCATE	Blank
	COMMIT	COMMIT	COMMIT	COMMIT
	DEALLOCATE	DEALLOCATE	DEALLOCATE	DEALLOCATE
	Blank	COMMIT	DEALLOCATE	COMMIT

Note that when remote packages do not specify the ISOLATION and RELEASE options, they default to the options of the DISTSERV plan under which the remote packages run. Default options for the DISTSERV plan are ISOLATION(RR) and RELEASE(COMMIT).

1.5 Package Benefits

A number of significant benefits are associated with the use of packages. Implementation of packages provides solutions for problems users have encountered with plans. In addition, packages provide enhancement in application enabling. This section discusses the possible benefits users can derive from packages.

1.5.1 Reduced Bind Time

Problem Before packages, every time a program was modified (with or without an SQL change), it was necessary to do a BIND PLAN ACTION(REPLACE), which resulted in reoptimizing all DBRMs in the plan, even the unmodified DBRMs. This tended to cause:

- Much resource and time consumption
- Undesired access path change for the unmodified DBRMs.

Solution BIND PACKAGE allows a single DBRM to be optimized. So when there is a change in a program, the package associated with it is the only package that needs to be rebound (see Figure 12 on page 18). None of the plans that include this package needs to be rebound. Any other packages in the plan are not accessed at all. This is very beneficial in an environment where the DBRM is shared in many plans, or where there are

a considerable number of DBRMs in one plan. Contention for the DB2 catalog, the workload, the amount of time the plan is unavailable, and the time needed for rebinding is likely to be reduced.

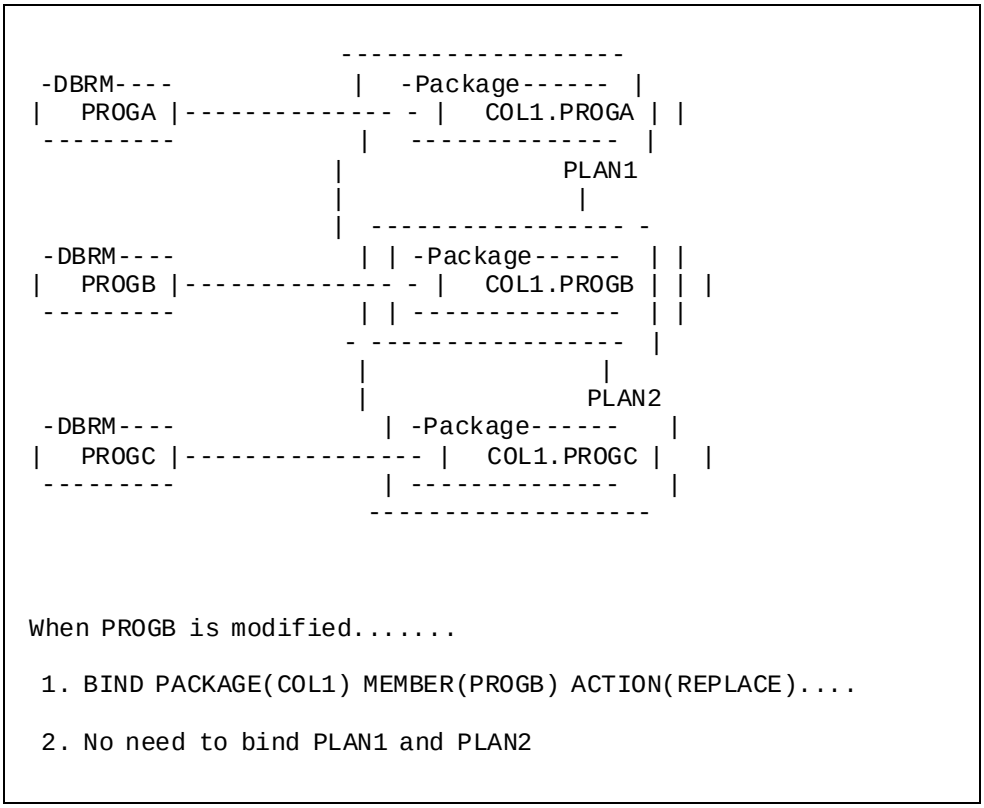

Figure 12. Rebinding Packages

1.5.2 Support for Multiple Versions of Programs

Problem Although load modules could maintain different versions by linking them into different libraries, there was no way to do the same thing with plans, because the plan name had to be unique in a DB2 subsystem. When two versions of the same program (test and production) had to reside on the same DB2 subsystem, the only solution was to bind each DBRM into a different plan. The lack of multiple versions support might have been a big problem in case of fallback, for example, which requires binding the plan again.

Solution With the use of the package attribute, version_id, you do not need to change the collection_id or package_id. Each version of the program must be precompiled with a different VERSION value and put into a separate DBRM library, and each load module must be linked into a separate library. By binding a package with two different DBRM libraries, two separate versions of the package are produced. DB2 picks up the appropriate package version that matches the executing load module. Fallback is much easier and less time consuming, because DB2 can keep the backup version and the new version of a package.

Multiple versions of a program also can be supported by controlling collections. DB2 accepts identical package_ids if they are in different collections. Therefore, instead of changing the VERSION value for each version at precompile time, the collection can be changed for each version when binding a package.

For further discussion of the actual use of collection and version, see Chapter 3, “Using Collections” on page 43 and Chapter 4, “Using the VERSION Keyword” on page 69.

1.5.3 Flexible Naming Conventions

Problem The method of providing a qualifier for an unqualified table name, view, and synonym was limited to using the authid of the plan owner. The qualifier had to be identical for each DBRM in a plan.

Solution With the use of the QUALIFIER keyword, you can explicitly specify a qualifier for each single package. This function can also simplify the transaction design for tables with different qualifiers, such as mirror tables, which are discussed in Section 1.5.6, “Simplified Access to Mirror Tables” and Section 3.2, “Mirror Tables” on page 45.

1.5.4 Increased Bind Availability

Problem Binding a plan could be done only when the plan was not being executed, and a plan was not executable when it was being bound. The bigger the plan, the more a user was concerned about when to bind.

Solution As long as a package is not being allocated, it can be rebound even if the associated plan is still allocated. This increases the availability of packages for binding.

1.5.5 Granularity in Bind Options

Problem All DBRMs included in a plan had to execute with the same bind options.

Solution You can specify bind options at the package level by specifying them at bind package time. This provides the flexibility of allowing each program in a plan to use different options. One program may be bound with ISOLATION(RR), and another program with less restrictive requirements, with ISOLATION(CS). Without packages, all programs in a plan must use the same option.

1.5.6 Simplified Access to Mirror Tables

Problem Mirror tables are tables that have an identical structure with different qualifiers. Before DB2 V2.3, applications had to duplicate code in programs to access mirror tables using either the fully qualified table for each statement or dynamic SQL.

Solution With the use of the QUALIFIER keyword and the new SQL statement SET CURRENT PACKAGESET, it is possible to simplify mirror

table access. Only one source is needed, with its SQL statement having an unqualified table name. You bind the DBRM into multiple packages, each having the necessary qualifier and a different collection_id. In the program, you use the SET CURRENT PACKAGESET statement to direct DB2 to choose a package from a particular collection dynamically. There is thus no need to duplicate the source code with a fully qualified table name. Please refer to Section 3.2, “Mirror Tables” on page 45 for more details on the use of mirror tables.

1.5.7 Restrictions of Remote Access Eliminated

Problem Before DB2 V2.3, all SQL statements for a remote site executed dynamically, so they always required an incremental bind, even in queries where static SQL was preferable from a performance point of view. These remote accesses were limited to DB2-to-DB2 connections through the DB2 private protocol and were system directed.

Solution With DB2 V2.3, remote SQL statements can be executed statically, if they are bound to a package at a remote site. Packages are required for implementing remote unit of work (RUW) or distributed unit of work (DUW) using DRDA.

1.6 Package Authorization Overview

There are some authorization enhancements in DB2 V2.3. The overall concept of these enhancements is to separate system administration from data access. The same applies to new privileges related to packages: privileges of administration are separated from execution. This section summarizes the authorizations required to administer and use packages.

1.6.1 Explicit Privileges

The explicit privileges related to packages are briefly described below.

Package Privileges

The following three privileges are recorded in 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 grant option can grant them on a specific package, for example:

```
GRANT BIND ON PACKAGE COL1.PROGA....
```

Users with SYSADM, SYSCTRL, and PACKADM WITH GRANT OPTION authority can grant 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.* ....
```

SYSCTRL cannot specify the EXECUTE privilege in this case because it does not have the EXECUTE privilege over a package.

System Privilege

The BINDAGENT privilege is introduced:

BINDAGENT To bind, rebind, copy, or free a package or plan on behalf of the owner who granted the BINDAGENT privilege.

BINDAGENT is designed only for operations related to binding, so it does not have EXECUTE privilege.

BINDADD has been extended to include packages as well as plans.

These privileges are recorded in SYSIBM.SYSUSERAUTH.

Collection Privilege

As a collection is considered a DB2 resource, one must have the following privilege in order to bind a package into the collection:

CREATE IN To create packages in the designated collections.

CREATE IN privileges are recorded in SYSIBM.SYSRESAUTH.

1.6.2 Administrative Authorities

SYSCTRL and PACKADM are new administrative authorities that include package privileges. Each authority is described below.

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 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 authids
- Grant DBADM, SYSADM, and PACKADM authorizations.

PACKADM

PACKADM privileges are recorded in the SYSIBM.SYSRESAUTH table.

PACKADM is designed to be a package administrator in a given collection.

PACKADM has all explicit package privileges (BIND, COPY, EXECUTE, FREE) 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 * TO user1
```

Table 5 summarizes the privileges held by the administrative authorities.

Table 5. Package-Related Privileges: Administrative Authorities

	BINDADD	BIND	COPY	EXECUTE	CREATE IN
SYSADM	Y	Y	Y	Y	Y
	All packages in all collections	All packages in all collections	All packages in all collections	All packages in all collections	All collections
SYSCTRL	Y	Y	Y	N	Y
	All packages in all collections	All packages in all collections	All packages in all collections	N/A	All collections
PACKADM	N	Y	Y	Y	Y
	N/A	All packages in specified collection	All packages in specified collection	All packages in specified collection	Specified collection

1.6.3 Privileges Required for Package-Related Operations

Table 6 on page 23 summarizes the privileges—explicit, administrative authority, implicit— required for package-related operations. Users (authids executing the process) must have one of the three privileges.

Table 6 (Page 1 of 2). Privileges Required to Perform Operations Related to Packages

Operation	Which Authid	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 the 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 authid values for the owner depend on the authorizations of the binder: • If the binder has BINDAGENT authority, the owner must be the authid that granted bindagent authority to the binder. • If the binder has SYSADM or SYSCTRL authorities, the owner can be any authid. • For all other cases, the owner must be one of the secondary authids of the binder.			
Rebinding or bind replacing a package	Primary or any secondary authid of the binder	BIND or BINDAGENT	PACKADM SYSCTRL SYSADM	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.			
Freeing a package	Primary or any secondary authid of the user	BINDAGENT	PACKADM SYSCTRL SYSADM	Package owner

Table 6 (Page 2 of 2). Privileges Required to Perform Operations Related to Packages				
Operation	**Which Authid**	**Explicit Privilege**	**Administrative Authority**	**Implicit Privilege**
Copying a package	Primary or any secondary authid of the binder or target package owner	COPY on the source package, or BINDAGENT from the owner of the source package Same privileges required for binding a new package	SYSADM SYSCTRL PACKADM	Source package owner
Executing a plan	Primary or any secondary authid 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 authid 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 authid 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 will be 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, authid checking will be 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 run time. At run time the plan owner should have authorizations either on each package that is going to be executed or on COL.*.

It is strongly recommended that the plan owner have the EXECUTE privilege on each package list entry before binding the plan to avoid authid checking at execution time. A detailed description of authid checking is presented in Section 7.1.1, “Package List Organization” on page 165.

Chapter 2. Package Processing

Packages cannot be used without collections nor can they be executed without a plan. The packages that may be executed under a specific plan are defined in the PKLIST statement, and it is essential that the way in which this list is used by DB2 in searching for a package be clearly understood before using packages. This chapter covers the processing of packages in the EDM pool and the search algorithm that is used to locate the packages at execution time.

2.1 EDM Pool Processing

This section discusses the changes that have been made to the plan header to accommodate 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.

2.1.1 Overview of New Plan Structure

It may be useful at this point to review the new plan structure and in particular the types of entries that can be specified using the PKLIST parameter. Figure 13 on page 26 gives an overview of the possible components of a plan in DB2 V2.3, showing:

- **DBRMs**

  DBRMs are still supported. However, the use of DBRMs is not recommended because the conditions that require a rebind of the plan are unchanged. At run time, if the CURRENT PACKAGESET register is not set by the application program (in which case DB2 skips the DBRM checking), DB2 checks the plan header first to see whether DBRMs are present. If they are, DB2 scans the 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 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.

- **PKLIST entries**

  The following entry types can be included in the plan definition using the PKLIST parameter:

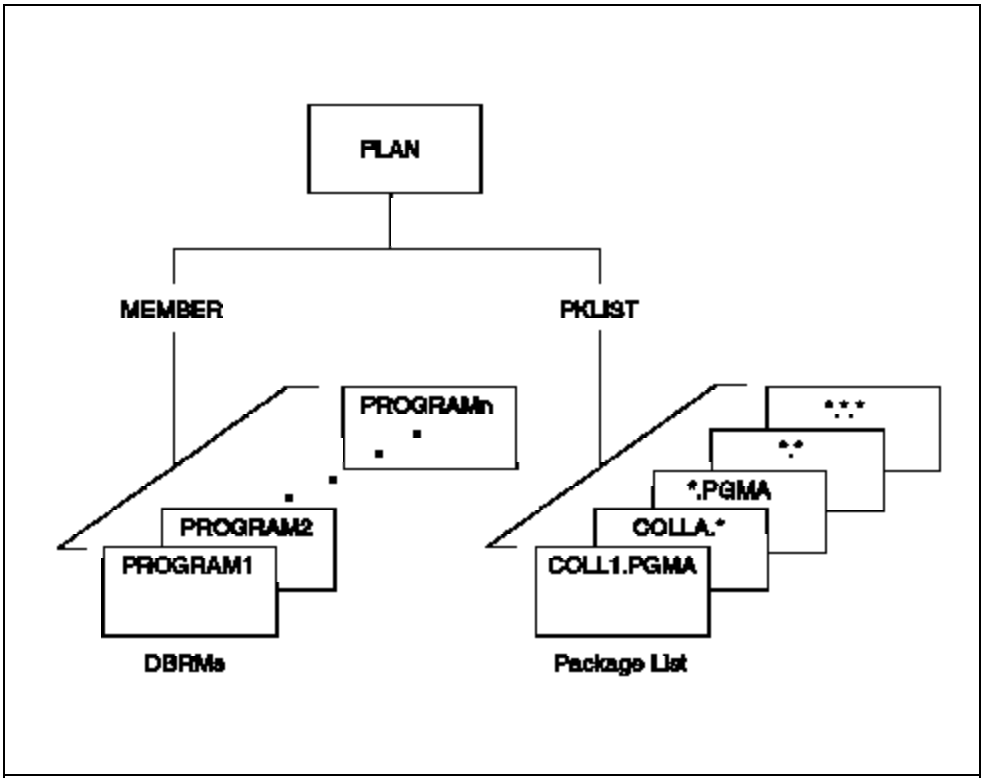

Figure 13. Overview of Plan Structure

Entry Types	Description
COLL1.PGMA	**Specific packages**

Specific packages, identified by collection_id and package_id (DBRM name), can be included in the PKLIST parameter in the BIND PLAN command. As with DBRMs, adding new specific packages requires a rebind of the plan.

Specific packages could be used in the first stage of migration to packages for the common modules that are used in many plans. This would allow these modules to be recompiled and rebound without the cost of rebinding all of the plans in which they are used.

The sequence in which specific packages are specified in the package list should reflect their usage rate so that those with highest utilization are at the front of the list.

COLL1.*	**Collections**

Packages can also be included in the package list in the form of collection_id.* when a plan is bound. This specification allows all packages bound into the specified collection to be executed under the plan. The advantages of using a wild card, that is, an asterisk (*), for the package_id are that the specific packages do not need to be included individually, and new packages can

be added to the collection without the need for a rebind of the plan. As mentioned previously, the sequence in which the collections are specified within the package list is important when the PACKAGESET register is blank because DB2 searches all entries, that is, both specific packages and collections, in the order specified. Performance may be adversely affected where the number of entries in the package list is high. Specific entries and collections may be intermixed within the package list.

***.PGMA** This entry indicates that all collections that contain a package with a package_id of PGMA are valid for this plan. DB2 will return a -812 SQL code if this entry is used without a value being set in the CURRENT PACKAGESET register.

. This entry indicates that all packages in all collections are valid for this plan. Entries such as these also require that the CURRENT PACKAGESET register be set in the program.

Of course, when searching for a package during a run, only one collection_id is used, and it is based on the contents of the CURRENT PACKAGESET 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. 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.

Again at run time, only one location_name and one collection_id are used, and they are based on the values in the CURRENT SERVER and CURRENT PACKAGESET registers, respectively.

Plan Header Contents

The addition of package lists to the plan has altered the contents of the plan header as shown in Figure 14 on page 28. The contents in the header related to DBRMs are unchanged, with the package list specified at plan bind time added at the end. The major points of interest in this figure are:

- The DBRM directory containing a sorted list of the DBRMs bound into the plan.
- The PKLIST entries specified in the PKLIST parameter. Note that these entries are stored in the sequence in which they were specified at the time the plan was bound.

When there are no DBRMs associated with a plan, section information is not stored in the header, thus reducing the DASD requirement. The format

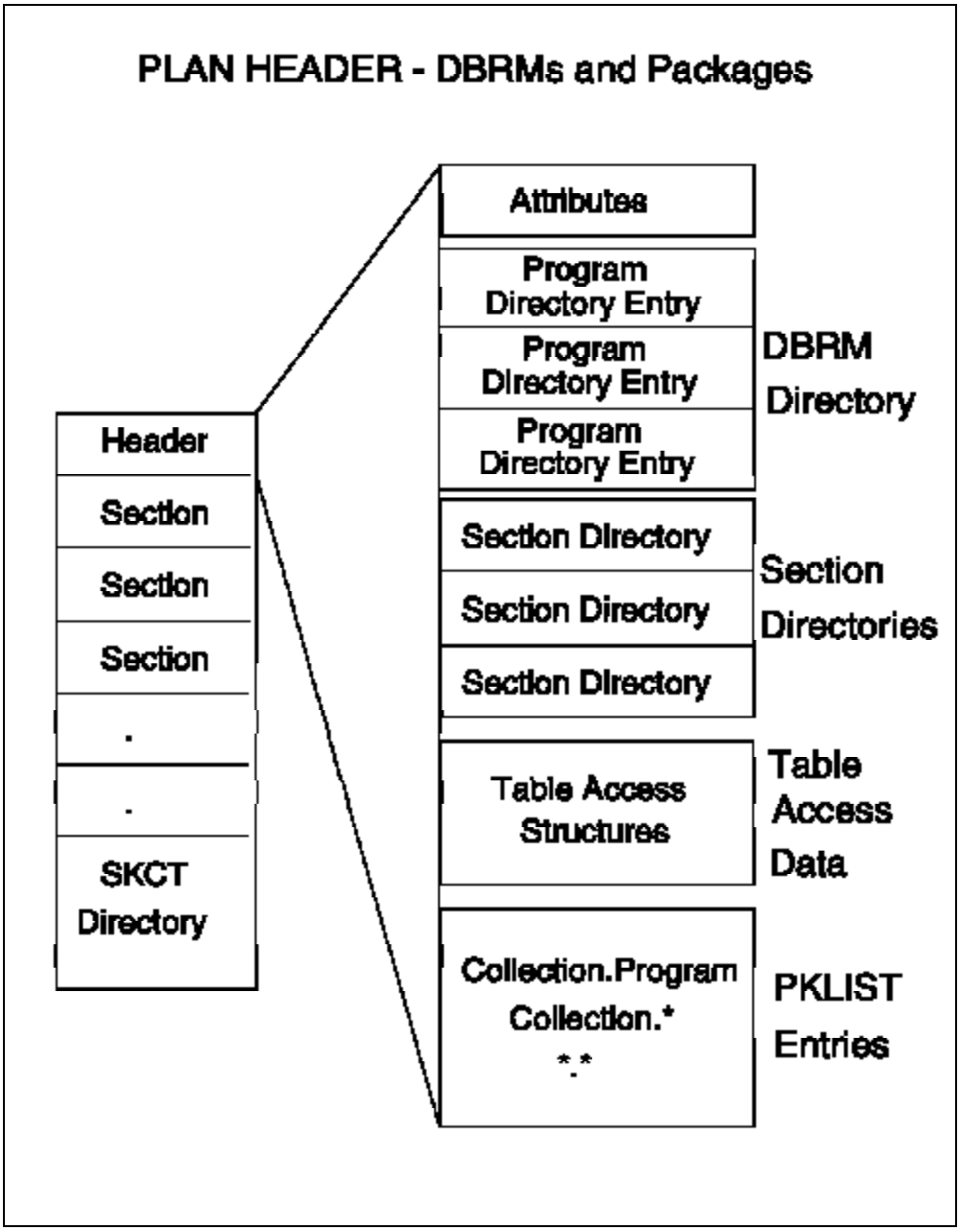

Figure 14. Plan Header When Both DBRMs and Packages Are Present

of the plan header when there are no DBRMs associated with the plan is shown in Figure 15 on page 29.

DASD space is also saved when the same package is used in more than one plan, as the package data is stored only once. The DBRM corresponding to this package would have its data stored in each of the plans in which it was bound. This situation might also lead to savings in the EDM pool. If a DBRM is used in many different plans, there will be multiple skeleton cursor tables (SKCTs) in the EDM pool for that DBRM if different plans are using it concurrently. In contrast, only one skeleton

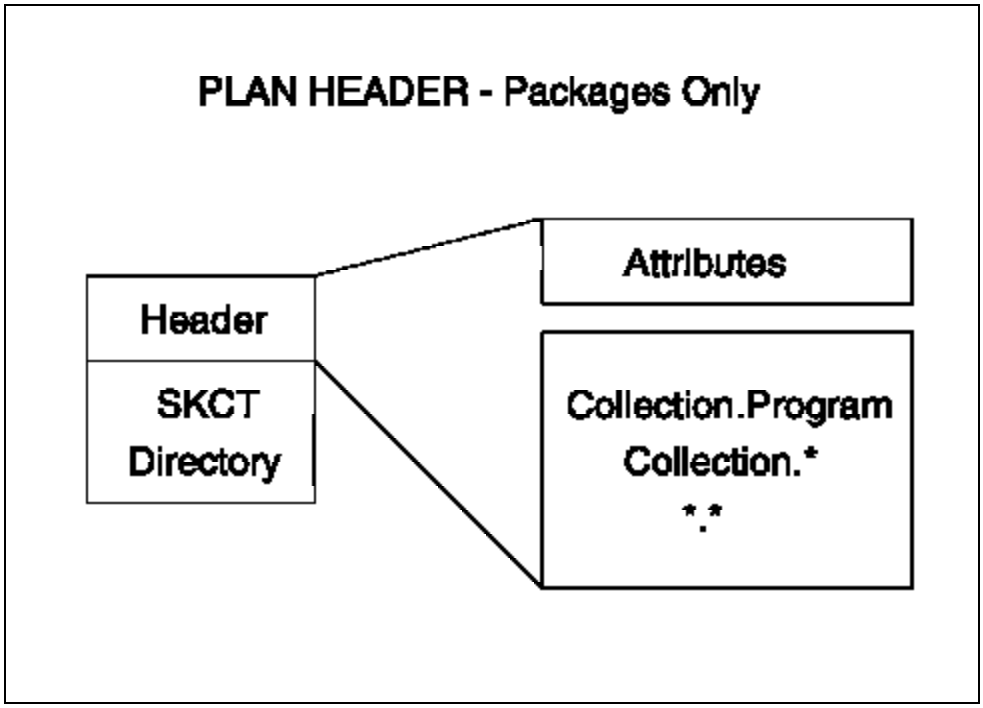

Figure 15. Plan Header When Packages Only Are Present

package table (SKPT) is allocated in the EDM pool for a package regardless of both the number of plans in which it is bound and the number of concurrent users from different plans.

2.1.2 Package Loading in the EDM Pool

Figure 16 on page 30 gives an overview of the processing of packages in the EDM pool. The figure assumes that there are no DBRMs associated with the plan and shows:

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 will be 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 Section 2.2, “Package List Search” on page 32.

   The SKPT directory for this package is loaded from the Skeleton Package Tablespace (SPT01) and forms part of the skeleton copy.
3. The package header is read from the Skeleton Package Tablespace (SPT01).
4. A copy of the package header is made for the skeleton copy.
5. The relevant section is read from the Skeleton Package Tablespace (SPT01).

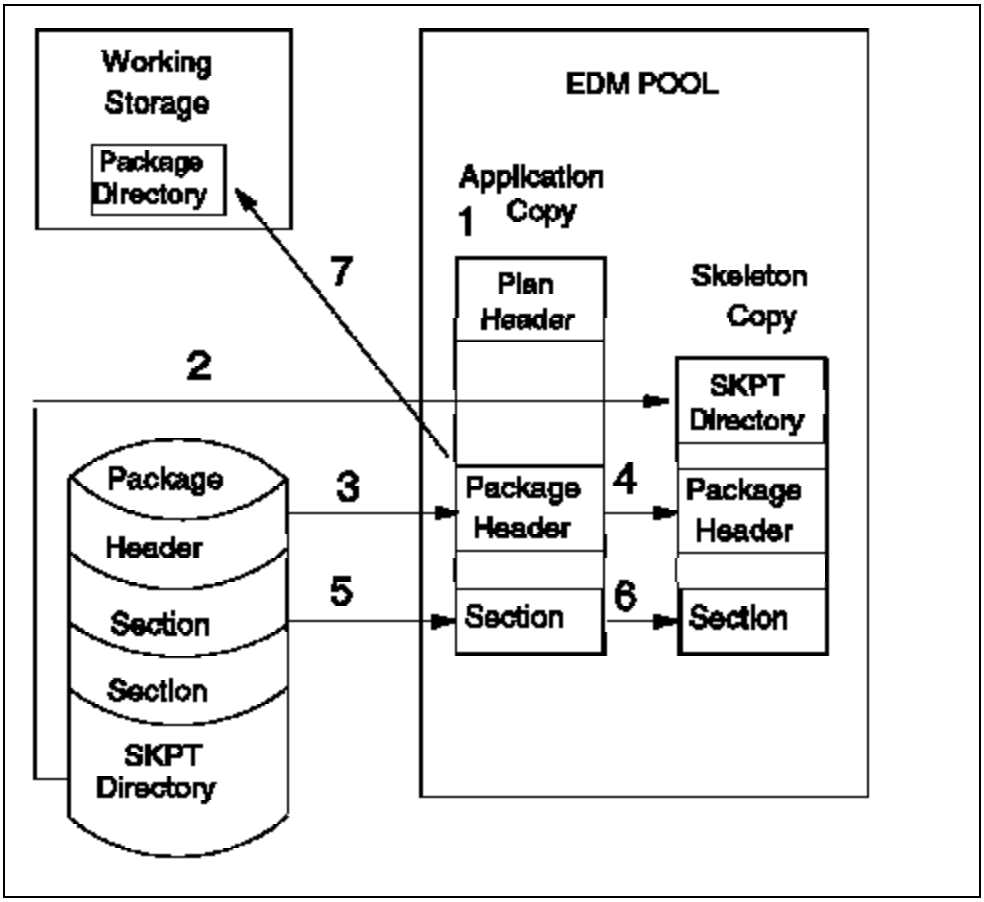

Figure 16. Package Loading in the EDM Pool

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_id, 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_id, 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 will remain in the EDM pool and may 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.

2.1.3 Package Rebind and the EDM Pool

An IMS* 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 will result in a resource unavailable message. Note that this restriction applies to instances of a package; that is, a version of a package could be rebound while another version is allocated to a plan.

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 will require 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, special action may be required to allow it to be rebound. This could include stopping one or more threads to enable the plan to be deallocated. When the package is rebound, the skeleton is marked invalid, and a request for it will require a read from the DB2 Directory as before.

2.1.4 Reuse of Packages in the EDM Pool

When another application using a different 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 17 on page 32 shows the processing involved in this reuse.

The figure shows the processing that occurs in the EDM pool when a package that Application 1 is using has been identified among 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:

1. The plan header is loaded from the Skeleton Cursor Tablespace (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 EDM pool manager. The EDM pool manager always checks the packages already allocated in the EDM pool before initiating a read from SPT01. The required package is already in the pool and 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.

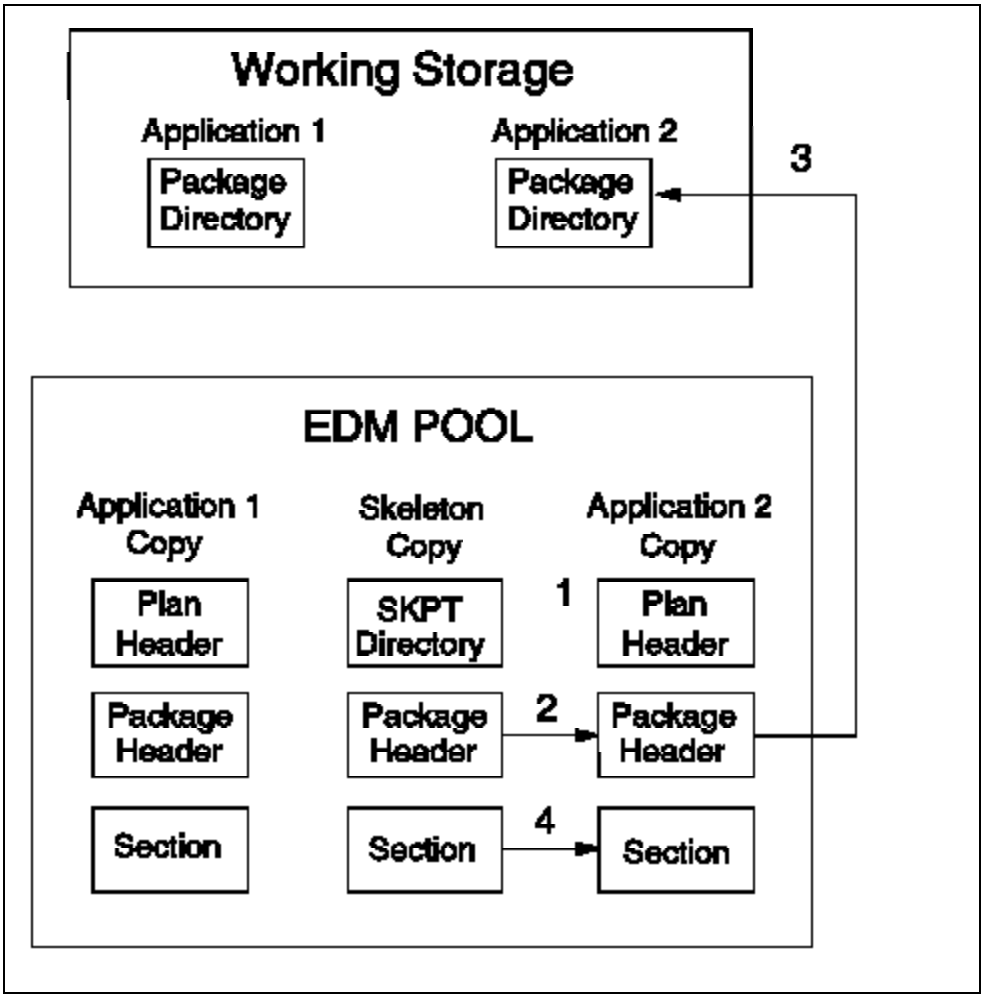

Figure 17. Package Replication in the EDM Pool

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.

2.2 Package List Search

A plan is now a dynamic structure. DBRMs can still be included in the plan using the MEMBER keyword, and they are processed as in previous releases. The flexibility comes from the package list specified at bind time. Packages executed at run time 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_id 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_id are specified, or specific collections, where location_name and collection_id are specified. Or, the specification of the collection can be deferred until run time using an asterisk for either the location or collection qualifiers, or both.

2.2.1 Consistency Token

It is perhaps useful at this time to review the concept of the consistency token, as DB2 uses the token in conjunction with the location name, collection_id, and package_id 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_id, and version_id. This index ensures that each version of a package within a collection has a unique version identifier.

The other index is based on location name, collection_id, package_id, and consistency token. This index ensures that each version of a package within a collection has a unique consistency token.

The consistency token that DB2 uses is based on a system-supplied time stamp except where the LEVEL precompiler option is used. This option was introduced to save on the cost of rebinding after a recompile where the SQL statements were unchanged. Use of the LEVEL option is not usually recommended because the consistency checks are less valuable and the cost of rebinding is much lower with packages than with plans using DBRMs.

The important point to note is that there will always be 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.

2.2.2 Package Search Overview

When an SQL statement is issued in a program, three parameters are passed to DB2 to identify the appropriate DBRM or package to be used. These are the DBRM name, the section number, and the consistency token. Before DB2 V2.3, a DBRM could occur only once in a plan so that the DBRM name alone uniquely identified the DBRM to be used. The consistency token was used to ensure that the load module being used and the DBRM in the plan were both generated in the same run of the precompiler.

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 run time depends on three factors, namely, the value in the

CURRENT SERVER special register, the value in the CURRENT PACKAGESET special register, and the sequence of entries in the package list.

Figure 18 on page 35 gives an overview of the package search and allocation process.

The processing that is detailed in Figure 18 on page 35 is described below. The numbers associated with the description match the numbers in the figure.

1. The DBRM name and consistency token passed from the program and the contents of the CURRENT PACKAGESET register 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 call came from the same module as the previous SQL call 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 will be blank if there is no distributed processing, or it will be the name in the SYSIBM.SYSLOCATION 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 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.

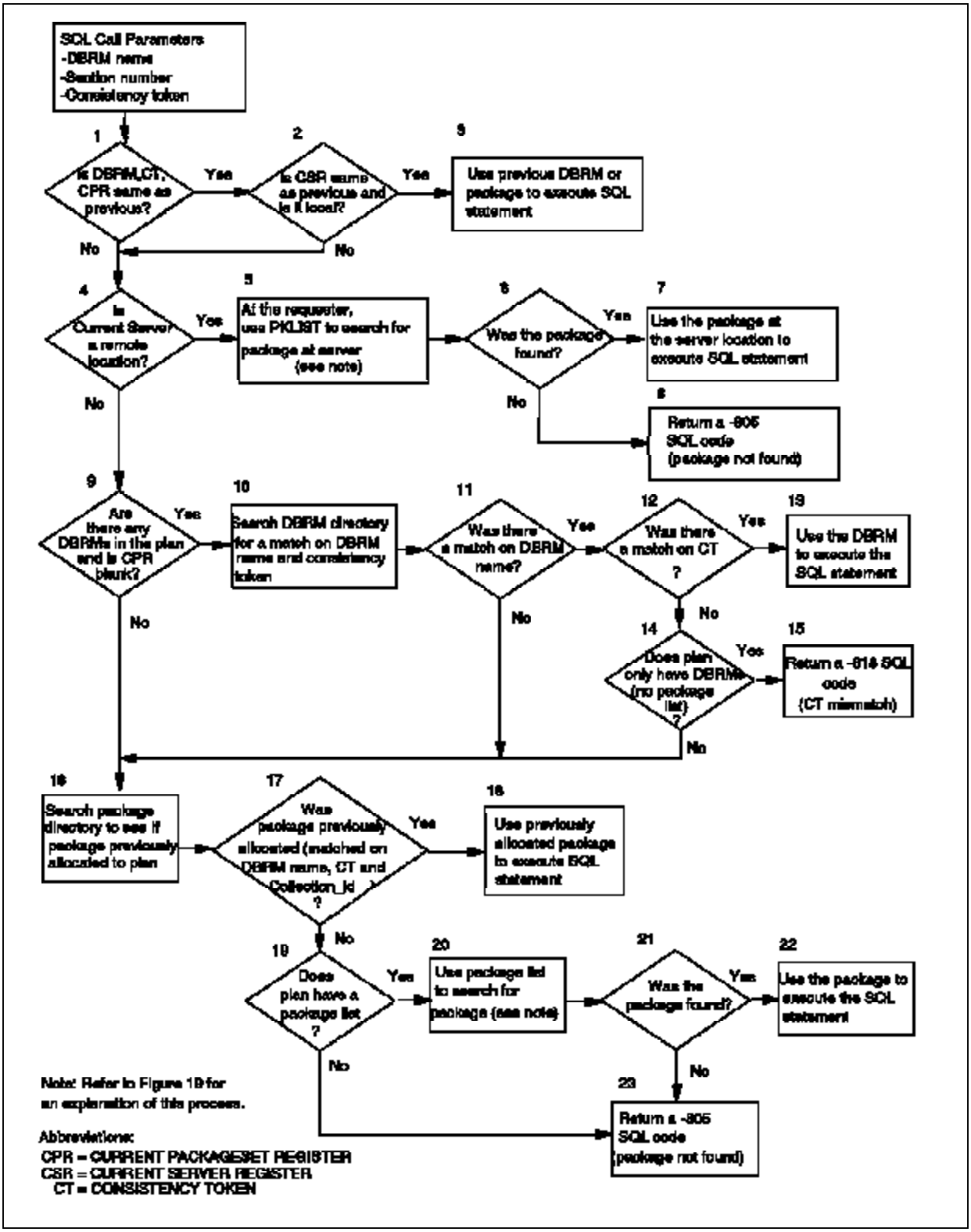

Figure 18. Package Search Process: Overview

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, 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 19 on page 39.

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 is set, the name of the package required is fully qualified.

   If it is not set, a number of packages may 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 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_id, 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) 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 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. As 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.

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. As it is known that the location_name refers to the local location, this check requires matching on three variables, namely collection_id, package_id (DBRM 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 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 19 on page 39.

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.

2.2.3 Package List Search

DB2 uses package list search (Figure 19 on page 39) 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 below. The numbers in the description match the numbers in the figure.

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_id for the package list entry. If they match or if the package_id 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 nonblank 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.

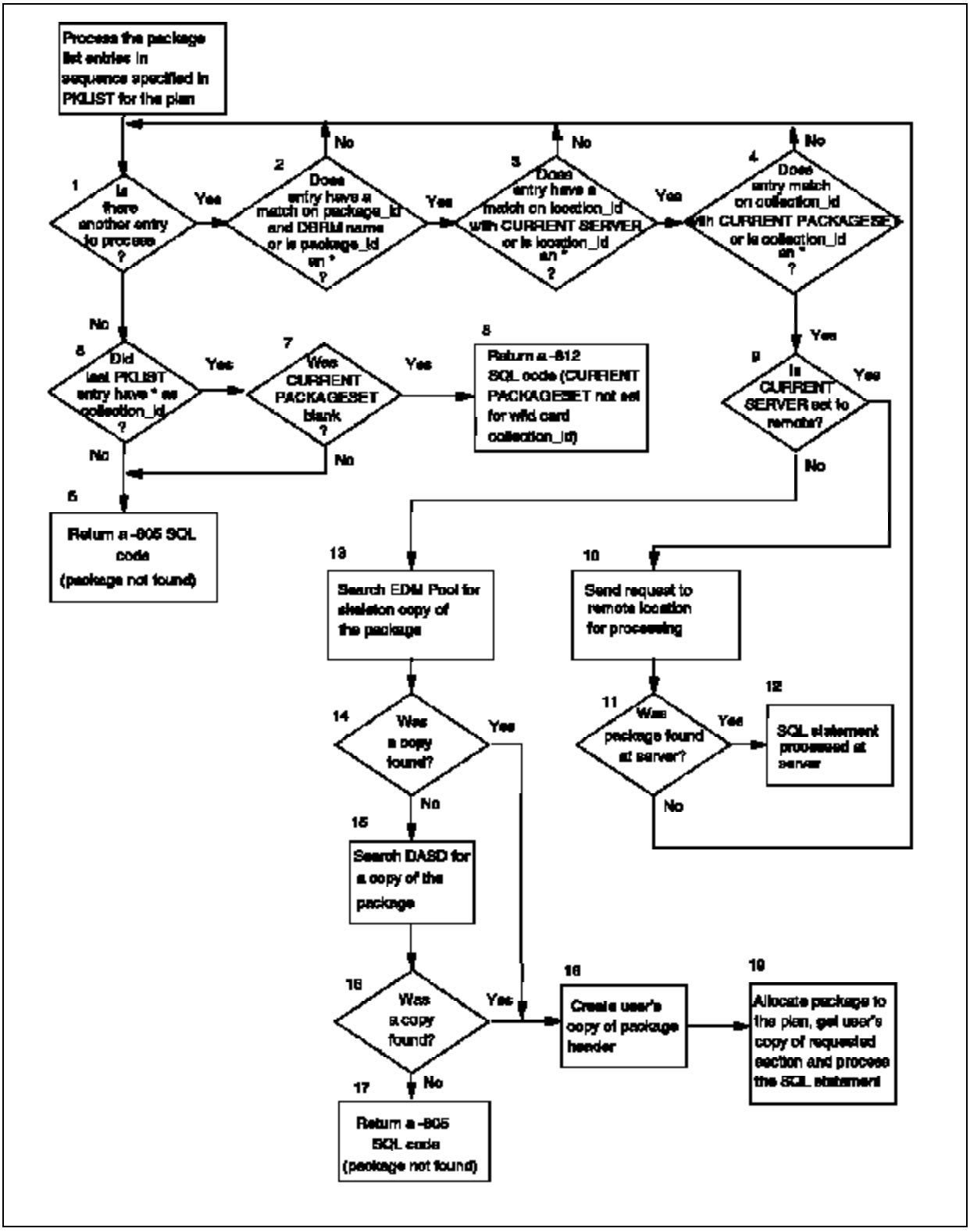

Figure 19. Package List Search

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_id are shown, is summarized below.

   - If the content of the CURRENT PACKAGESET register 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 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. 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 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_id, 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 skeleton package tablespace (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 call is allocated, and the SQL request is processed.

2.2.4 Implications of the Search Process

Two conclusions can be drawn from the package search process:

- DB2 does not return a -818 SQL code when packages are bound into a plan. 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. Determining the reason for this return code may be much more difficult than for a -818 SQL code.

- When the CURRENT PACKAGESET register is not set in a program, the use of multiple wild card entries such as COLL1.* in the package list may degrade performance. For example, use of a package list such as:

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

  in the worst case would result in 100 accesses to SPT01 if the required package is in COLL100.

Chapter 3. Using Collections

DB2 collections provide a method for logically grouping packages. In this chapter we look at the various ways of grouping packages into collections. We also explain how to use collections to support:

- Mirror tables
- Testing of a new package using the try-out collection
- Package versioning.

3.1 Guidelines for Collection Usage

Naming standards and guidelines for the use of 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 a new administrative task that is critical to the successful implementation of packages.

The approaches available for using collections range from one per environment, such as production, to one per DBRM. The latter is not recommended because of the administrative and possible performance overheads. The manner in which packages are grouped into collections depends to a large degree on the procedures used in the existing test and production environments.

The approaches for using collections include:

- One collection per environment

  Using one collection for all packages in an environment offers great simplicity in terms of organizing packages. For example, a CICS* production subsystem has a single collection called CICSPROD into which all CICS program packages are bound. The plans used by the CICS transactions are bound with PKLIST(CICSPROD.*). One advantage of this arrangement is that big plans could be used by more transactions and therefore facilitate thread reuse. With thread reuse, CICS packages should be bound with the RELASE(COMMIT) option to ensure adequate resource release. Another advantage is that less administrative effort is required to maintain the packages.

  This option 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, there is little granularity in security and ownership of the packages.

  With more packages in the plan, it becomes more difficult to understand and manage the performance impact. You should test this approach carefully in your own environment before implementing it.

- One per business area

  Using one collection for each business area, such as finance, marketing or personnel, would involve having many applications in one collection

with authorizations limited in much the same way as above. Typically, the applications would share a common set of tables and common program modules. The administrative load for this approach is less than that for applications as only one collection is involved and common modules would need to be bound only once. However, greater effort would be required to identify the packages that must be included when applications are promoted.

Using collections in this way would be suitable for use in both the test and production environments where a plan was created for each collection. Where one plan is used for an environment, the number of collections in the package list may impact performance unless the programs set the CURRENT PACKAGESET register.

- One collection per application

  Having one collection per application provides greater security and administrative control as it allows the collection, package, and plan privileges required for using packages to be limited to the application group. It also allows easy identification of the packages involved in the application. This method would be 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.

  Grouping of packages in this way would be applicable in the different test environments. In the unit test environment, plans containing a single collection of the form APPL_ID.* could be set up. For the system test environment where performance is not an issue, the plan could be defined to include all of the application-based collections. This approach would not be suitable if system testing included performance monitoring unless the CURRENT PACKAGESET register is set in each of the programs. The reason for this is that the performance of individual applications would be impacted by their position on the plan package list.

  Similar restrictions would apply to the use of this approach in the production environment.

- One collection per plan

  In migrating to packages, it may be desirable to use plans as they were used in previous releases (that is, one plan for each batch program or job step, or for each online transaction). All DBRMs from a plan are bound into a collection, and there is one collection for every plan (that exists in DB2 V2.2). This organization provides the greatest similarity to the existing DB2 V2.2 environment, but it does not exploit the benefit of having a bigger plan, which is now possible with DB2 packages.

  The administrative work involved in creating and maintaining plans at this level is greater than for the approaches outlined above both in identifying the packages that should be included in a plan and in the authorizations necessary.

- Special-purpose collections

  Collections can be used for special purposes.

  One example would be a collection used to provide fallback in production where a program modification fails. This is discussed in

detail in Section 3.4, “Package Versioning Using Collections” on page 49.

Another example would be a collection used to try out a new package. The new package is bound using the copy option from the older package, but it may have different attributes, such as different access paths, and qualifiers. This is discussed in Section 3.3, “Try-Out Collection” on page 47.

Figure 20 illustrates some of the points regarding the approaches to using collections as discussed above.

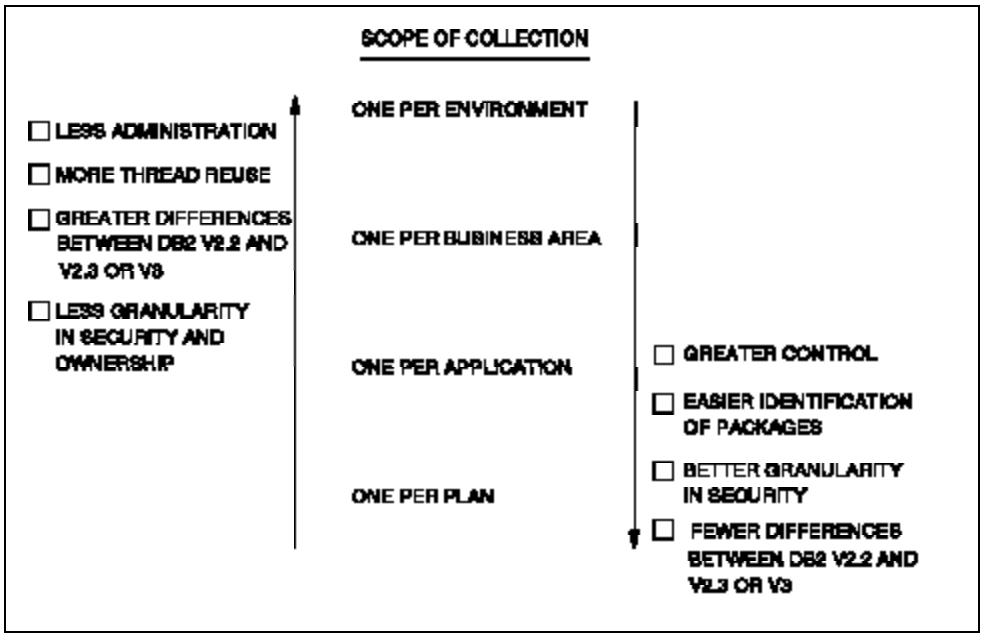

Figure 20. Approaches to Using Collections

3.2 Mirror Tables

Collections in package versioning 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. In previous releases of DB2, a single authid was used as a qualifier on all unqualified table names in static SQL. As a result, any program that had to access tables other than those qualified by the plan owner authid had to specify fully qualified names when using static SQL or use dynamic SQL. This situation often led to duplication of programs to support mirror tables.

Figure 21 on page 46 shows the principles involved in supporting mirror tables using collections.

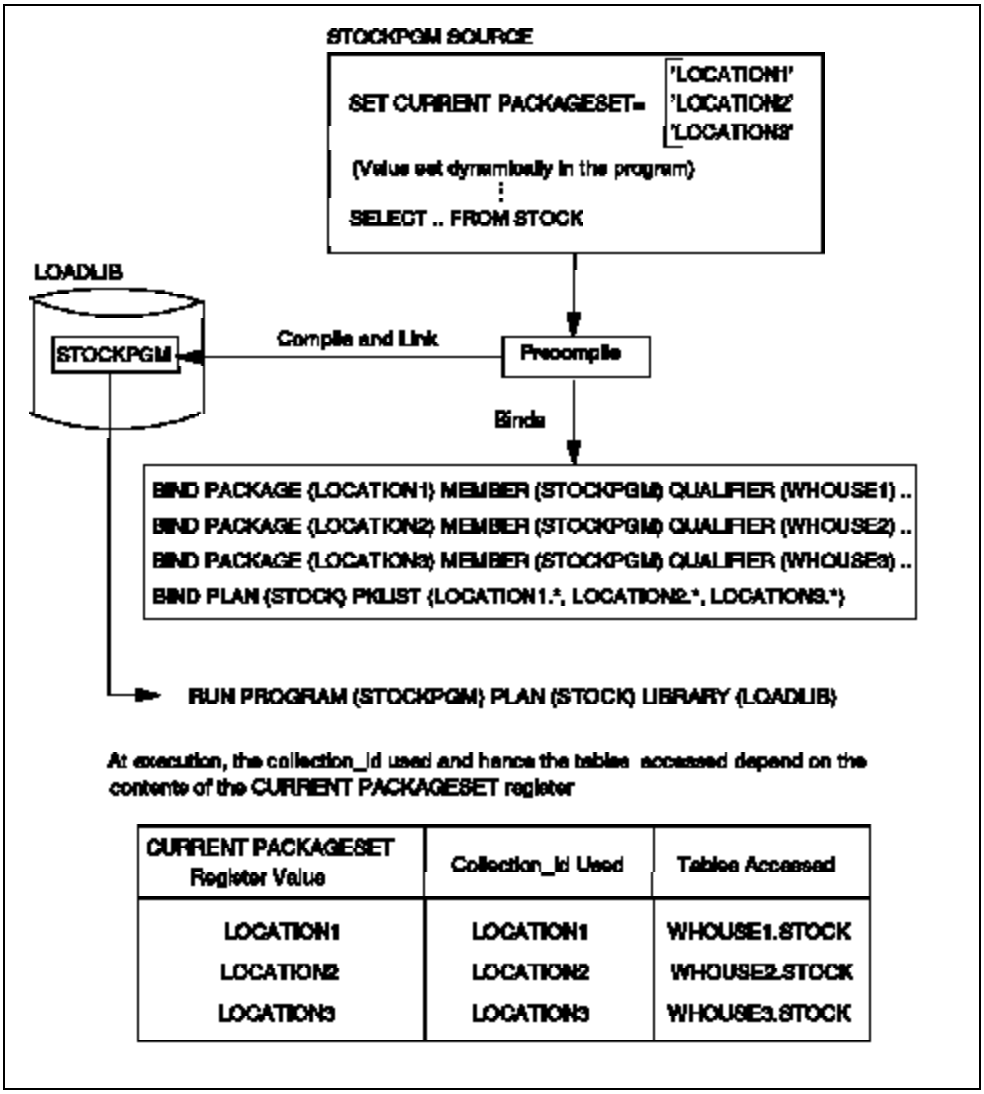

Figure 21. Collection Support of Mirror Tables

There are two points to note in the source program shown. The first is the SET CURRENT PACKAGESET statement, which indicates the collection to be used. In this example, there are three sets of tables containing stock data for warehouses at three locations, namely, LOCATION1, LOCATION2, and LOCATION3. The second point to note is that the table names in the program are unqualified.

The program involved, STOCKPGM, 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 collection_ids and different values for the QUALIFIER parameter that identifies the set of tables to be used. For example, in this case when the package in collection LOCATION1 is used, the table that would be accessed is WHOUSE1.STOCK. An alternative to using the DBRM for subsequent binds is to use the COPY option of the bind package command. This approach is preferable where DBRMs are not retained.

At run time, the collection to be used is identified by the value in the CURRENT PACKAGESET register set dynamically by the program. The program could also access data for all locations in one execution by looping to set the register value in turn to each of the collections. The CURRENT PACKAGESET register is local to the agent (user) executing the plan. Once the register is set to a collection_id using the SET CURRENT PACKAGESET register statement, it remains effective until it is reset by another SET CURRENT PACKAGESET statement or until the agent terminates.

Figure 22 illustrates the programming requirement when the SET CURRENT PACKAGESET statement is used. Program A calls program B. Program B is a common routine, and the package resides in a common collection called COMMON. For performance, program B issues SET CURRENT PACKAGESET = 'COMMON'. Before this, program B must save the previous value of the CURRENT PACKAGESET register so that, just before it exits, it can reset the register to this value.

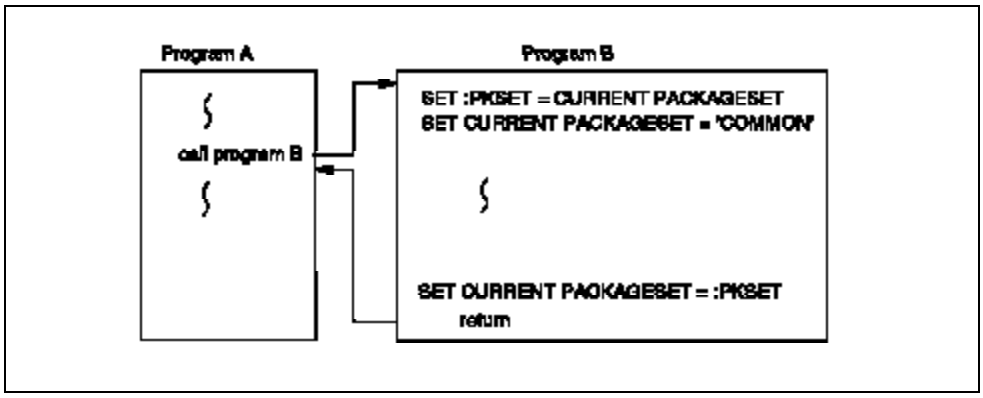

Figure 22. Saving CURRENT PACKAGESET Register

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.

3.3 Try-Out Collection

The try-out technique enables users to test or try out a new package without touching the currently used package. For this technique to work, the consistency token of the new package must be the same as the consistency token of the package it is to replace.

This technique can be used, for example, to evaluate the impact of a new index. The existing package is bound into a try-out collection using the BIND COPY option. The consistency token is the same, but this time, the access path may include the new index. This package can then be used to evaluate the impact of the new index. Note that the original package is untouched and remains accessible through the plan if there is a need for fallback.

In Figure 23 on page 48, the original package is PARTINQ in collection CARDEAL. The plan is bound with both the try-out package and the original package, but the try-out package must be coded first in the package list:

```
BIND PLAN(planname) PKLIST(TRYOUT.PARTINQ, CARDEAL.PARTINQ)
```

The TRYOUT collection remains empty most of the time and is only used when a try-out is required.

Assume that a new index has been created that affects the access path of the PARTINQ package. To test out the impact of this index, the CARDEAL.PARTINQ package is bound into the TRYOUT collection (using its DBRM as input or the bind package with COPY option). The consistency tokens for the two packages are the same. Reoptimization occurs during bind time and may include a new access path that uses the new index.

Assume also that CARDEAL.PARTINQ has been executing all this while. As long as the package is allocated in the plan, the new package will not be used. When CARDEAL.PARTINQ gets deallocated from the plan or when the plan gets deallocated, the next execution of the program uses the new TRYOUT collection package that has the new access path. This is because DB2 scans the PKLIST when it attempts to allocate the package. It sees the TRYOUT.PARTINQ package first and allocates it, overriding the original package in the second collection with the same consistency token. If the new package performs badly, fallback to the original package simply requires freeing the new package from the TRYOUT collection.

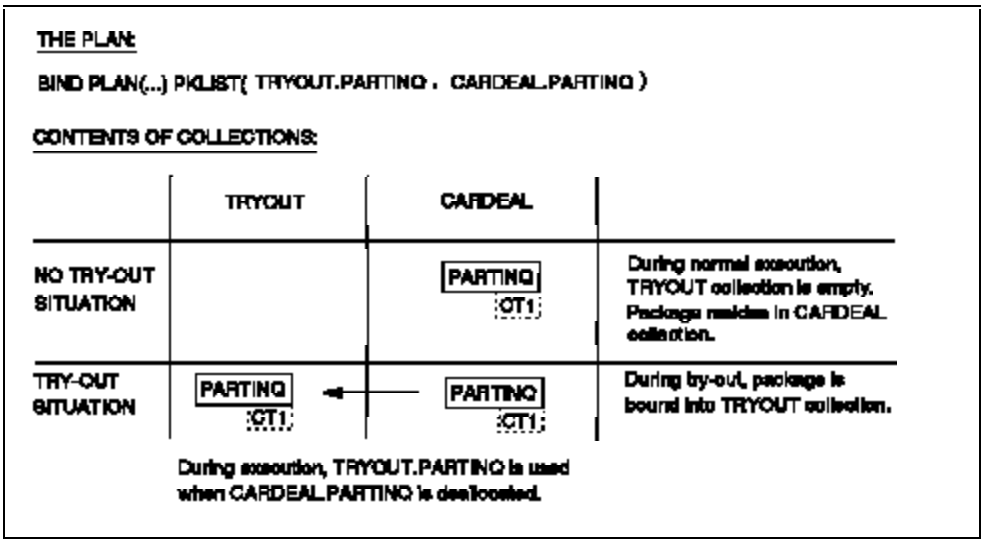

Figure 23. Try-out Collection

Note the following:

- Cutover to the new package does not occur as long as the original package remains allocated in the plan. The duration depends on the RELEASE option used to bind the old package. If RELEASE(COMMIT) is used, the cutover occurs when the PARTINQ program commits. If RELEASE(DEALLOCATE) is used, the cutover occurs when the thread terminates.
- SET CURRENT PACKAGESET is not used in the program. Selection of package is based on the PKLIST sequence. Hence, the try-out collection must be coded first.

The try-out technique incurs a performance overhead during normal program execution because DB2 scans the try-out collection every time the plan is executed. Use of the technique is justifiable only if the flexibility it offers is greatly desired.

3.4 Package Versioning Using Collections

This section discusses the organization of packages using DB2 collections as a means of versioning. Versioning in this context refers to the general concept of versioning and not to the DB2 precompiler VERSION option. A scenario is presented to show how versioning using collections can be implemented in the unit test, system test, and production environments.

3.4.1 Organizing Collections for Versioning

One of the uses of versioning is to keep a backup version of a package. The backup version is used in the same way as a backup load module during a fallback. Since a package_id must be unique within a collection, the backup package has to be bound into a “backup”collection. Figure 24 on page 50 shows the association between the program load libraries and the collections used. The CURRENT.* collection stores the current packages used by the production load library. The BACK.* collection stores a backup copy of the current package. This backup is performed by copying the package from the current collection before it is replaced during program promotion.

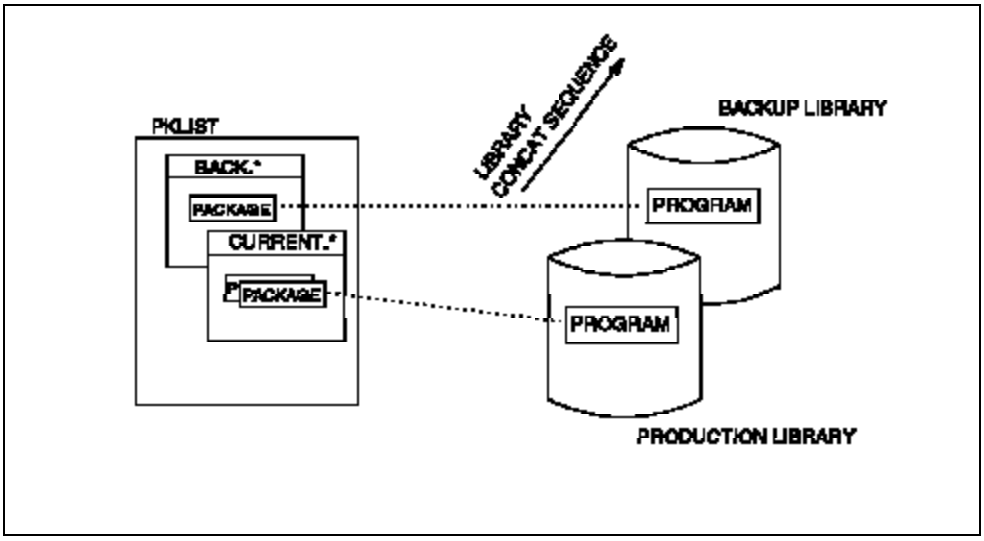

Figure 24. Current and Backup Versions Using Collections

Using collections to store multiple versions of a package is another method of managing different versions of load libraries. Figure 24 shows how the production and backup libraries are organized. The backup library is concatenated after the production library. Hence, the search sequence of a load module is production library followed by backup library.

Assume that the package_id for the program is PARTINQ. The plan is bound with both the current package and the backup package in the package list. By coding the BACK collection in the plan, the plan would have access to the backup packages in a fallback. In a fallback, this arrangement allows instantaneous access to the backup package without rebinding the plan or packages. The plan can be bound as follows:

```
BIND PLAN(planname) PKLIST(CURRENT.*, BACK.*)
```

or

```
BIND PLAN(planname) PKLIST(CURRENT.PARTINQ, BACK.PARTINQ)
```

Using a wild card for the package_id in the package list reduces administration when new packages are added to the collection.

For performance reasons, the packages that are most likely to be used should be coded first in the package list. Hence, the CURRENT collection is coded before the BACK collection. During normal execution of the program, DB2 can find the package from the first entry in the package list.

Note that the SET CURRENT PACKAGESET statement cannot be used to set the CURRENT PACKAGESET register to any value (other than blanks). Versioning using collections is a nice way to achieve a fallback. The reason for this (as will be explained again later) is that the backup package resides in a different collection from the current package. In a fallback, the package required by the program (backup version) is in the backup collection instead of the current collection. If the program had set the CURRENT PACKAGESET register to any nonblank value, DB2 could not find

the backup package in a fallback. Programs that access mirror tables using SET CURRENT PACKAGESET cannot use this method of versioning.

Below we show how a program can be promoted and how easy it is to fall back to the backup version of the program and package.

Promote Program

Assume that the program is called PARTINQ. PARTINQ currently exists in the system. Figure 25 on page 52 illustrates the three stages of promoting a new version of PARTINQ into the system.

Before Promotion

The PARTINQ load module executes from the current library. The corresponding package resides in the CURRENT collection. Both program and package have consistency token CT1. The backup load library and backup collection are empty.

Backing Up

To back up the library and package for fallback:

1. Copy PARTINQ from the current library to the backup library. Alternatively, copy the entire data set depending on requirements.
2. Bind currently used PARTINQ package from the current collection to the backup collection using the bind COPY option.

   Note: The access path of the backup package may be different because bind copy reevaluates the access path. Therefore you must check the EXPLAIN output and compare it with the previous access path.

Promote PARTINQ

Figure 25 on page 52 shows the promote actions. The new load module and package have consistency token CT2.

1. Precompile, compile, and link-edit the application into the current library.
2. Bind from DBRM into the current collection.

When the new load module executes, DB2 uses the package from the current collection rather than the backup collection because of the matching consistency token CT2.

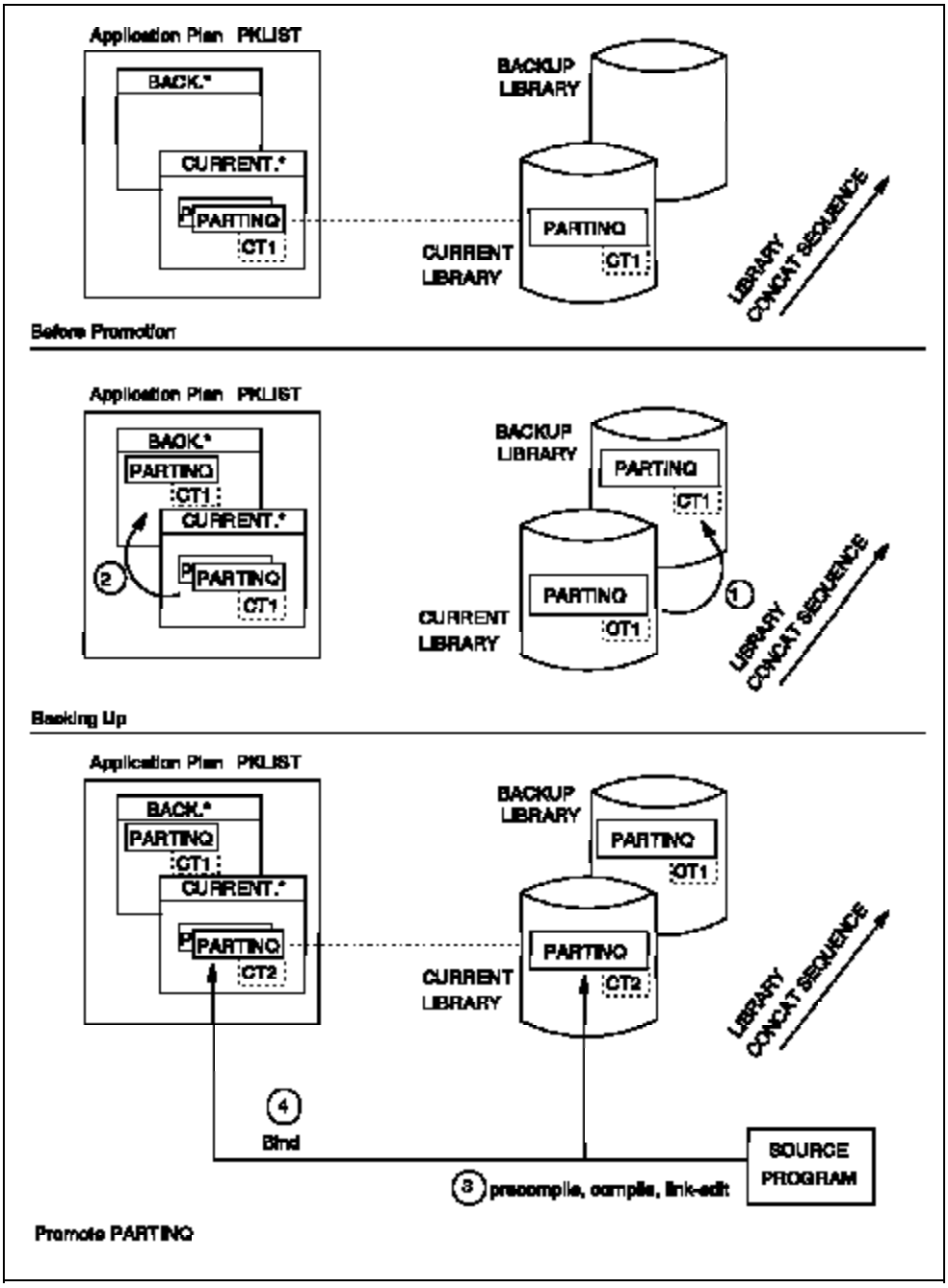

Figure 25. Promoting New Version of PARTINQ Program

Fallback

If the new version of PARTINQ fails for some reason, fallback to the old version is easy and fast. Figure 26 shows what happens at fallback. The failed program module is deleted from the current library (or renamed). The next invocation of PARTINQ executes the old version in the backup library because the library is concatenated to the current library. The old version of PARTINQ executes with consistency token CT1, and DB2 will use the package in the backup collection rather than the current collection because of the matching consistency token CT1.

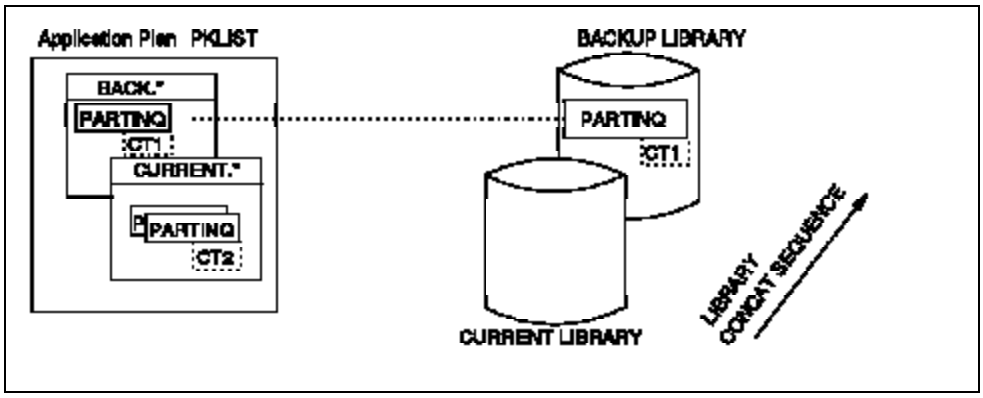

Figure 26. Fallback to Old Version of PARTINQ Program. For fallback, PARTINQ is deleted from the current library. PARTINQ from the backup library will execute at the next program invocation.

Note that the program does not use SET CURRENT PACKAGESET to select the collection it wants. Instead, it allows DB2 to pick up the package based on the consistency token embedded in the program. SET CURRENT PACKAGESET cannot be used to fix the collection_id to any collection (other than blank) within the program because the matching PARTINQ package could reside in either collection, depending on which load module is executing.

Note: Programs that use SET CURRENT PACKAGESET to select mirror tables cannot use collections for program versioning. The precompiler option VERSION must be used instead.

The performance impact of not using SET CURRENT PACKAGESET depends on the sequence of entries in the package list. The current collection must be coded before the backup collection, for example, PKLIST(CURRENT.*, BACK.*). Assuming there is no fallback, there is no performance impact if the CURRENT PACKAGESET register is not set, because DB2 can always find the package in the current collection, which is the first entry in the package list. If the backup collection is first in the package list, DB2 has the additional performance overhead of searching the backup collection first and then proceeding to search in the current collection before finding it there.

When the program has to fall back to the old version, however, DB2 has to execute the package in the backup collection. Since the CURRENT PACKAGESET register is not set, DB2 has to search both collections coded in the package list. When DB2 searches in the current collection, it searches in the EDM pool and then in the DB2 Directory (SPT01) before

realizing that the package in the current collection does not have a matching consistency token. Then it searches the backup collection and finds it there. Additional overhead is incurred for the DB2 Directory (SPT01) search in a fallback.

3.4.2 Using Collections for Versioning: A Scenario

This section describes a possible scenario for using collections for versioning in DB2. Assume that there are three environments: unit test, system test, and production. The test environments and the production environment belong to different DB2 subsystems.

The scenario describes the setup in each environment for a car dealer application. The PARTINQ program is part of the application.

The following aspects are covered in each environment:

- Use of collections in the environment
- People responsible for the application and their responsibilities
- Privileges for the people
- BIND examples.

Unit Test Environment

The following are assumptions regarding the unit test environment:

- The programmer
  - Works on the PARTINQ program in the car dealer project
  - Owns the test tables
  - Codes, precompiles, compiles, and link-edits the program
  - Binds the package into the UT_CARDEAL collection
  - Shares the UT_CARDEAL collection with other programmers
  - Executes the program using the UCARDEAL application plan.
- The project leader
  - Manages the car dealer project in the unit test environment
  - Owns the UCARDEAL application plan.

It was decided that the application should have its own collection in the unit test environment. This approach allows control at the application level, but it involves more administration compared to having a collection of a wider scope. It was also decided that common modules are not to be duplicated but shared.

To ensure that unit test developers do not have privileges over the packages in the system test environment (since they are in the same DB2), the unit test packages and system test packages for the application do not share collections. Hence, there is one collection per application per environment.

The collections for the application in the unit test environment are:

- UT_CARDEAL collection
  - Stores all packages related to the car dealer project in the unit test environment
  - Used by all programmers working on the car dealer project
  - Managed by the project leader.
- UT_COMMON collection
  - Stores common or shared SQL routines used by all projects in the unit test environment
  - Not managed by project leader.

There is only one application plan for the car dealer project in the unit test environment:

- UCARDEAL application plan
  - Plan used to execute any program in the car dealer project in the unit test environment only
  - Used by all programmers
  - Owned by the project leader.

In the unit test environment, packages are grouped into collections at the project level. The project in this case is the car dealer application. Programmers working on this project bind their packages into the project collection, which is UT_CARDEAL. Many packages can be shared among all the projects; these packages are bound into a common collection, UT_COMMON.

The programmers test their program through a common plan, UCARDEAL, which has both collections in the package list. Since the unit test and system test are within the same DB2 system, the UCARDEAL plan name will be different from the system test plan name so that plan execute privileges can be segregated.

Unit Test Privileges: The project leader has the following privileges:

- PACKADM ON COLLECTION *UT_CARDEAL*
- EXECUTE ON PACKAGE *UT_COMMON.**
- BINDADD WITH GRANT OPTION.

The project leader grants the programmer the following privileges:

- BINDADD for creating new packages
- CREATE IN COLLECTION *UT_CARDEAL*
- EXECUTE ON PLAN *UCARDEAL*.

The programmer also has implicit privileges over the test tables he or she owns and should be granted any explicit privileges over other tables the program may access.

Unit Test Binds: The project leader, who owns the plan, issues the bind plan command. A CLIST example:

```
PROC 0
CONTROL MSG
  DSN SYSTEM(DB23)
    BIND PLAN(UCARDEAL)                        -
         PKLIST(UT_CARDEAL.*,UT_COMMON.*)       -
         ACQUIRE(USE) RELEASE(COMMIT)           -
         ACTION(ADD)  VALIDATE(BIND)
  END
EXIT
```

Binding the DB2 plan with collection.* ensures that the plan need not be rebound every time a new package is added to the collection. Also, dropping the package from the collection does not invalidate the plan. This is preferred from an administration point of view.

The programmer performs the package binds. A CLIST example:

```
PROC 0
CONTROL MSG
  DSN SYSTEM(DB23)
    BIND PACKAGE (UT_CARDEAL)     MEMBER(PARTINQ)       -
         ACTION(REPLACE)                               -
         QUALIFIER(UTPGMR1) OWNER(UTPGMR1)              -
         RELEASE(COMMIT)   EXPLAIN(YES)                -
         ISOLATION(CS) SQLERROR(NOPACKAGE) VALIDATE(BIND) -
         LIBRARY ('STDB2C.UTEST.DBRMLIB')
  END
EXIT
```

Because there is only one collection (UT_CARDEAL), only one version of any package can be kept. The programmer will have to bind replace the existing package every time there is a modification. This is acceptable in the development environment.

One concern about privileges is that when programmers are given the BINDADD privileges for creating new packages, they also can create plans, which might compromise security. An alternative to this would be to grant the programmer bind privilege on the specific package and have the project leader bind the package for the first time:

```
BIND PACKAGE (UT_CARDEAL) DBRM(PARTINQ) ....
GRANT BIND ON PACKAGE UT_CARDEAL.PARTINQ to <programmer>
```

Thus, the programmer can bind and rebind the PARTINQ package but has no authority to create any new plans or packages.

System Test Environment

The system test environment mirrors the production environment. The discussion that follows describes the environment and assumptions.

The people involved in the car dealer application at the system test level are:

- Application tester
  - Executes and tests the program

- System test administrator

  - Promotes from unit test to system test
  - Owns the tables
  - Owns the application plan
  - Manages the collections ST_CARDEAL_CURR and ST_CARDEAL_BACK
  - Binds packages and plans for this application.

Four collections are used in the car dealer project at the system test level:

- ST_CARDEAL_CURR

  - Stores packages belonging to the car dealer project in the system test environment
  - These packages are “current.” They have matching consistency tokens with the programs in the current load library for the car dealer project.

- ST_CARDEAL_BACK

  - Stores backup copies of the packages in the ST_CARDEAL_CURR for the car dealer project in the system test environment
  - Stores the backup packages temporarily (as long as fallback may be required).

- ST_COMMON

  - Stores packages that are common to all projects in the system test environment.

- ST_COMMON_BACK

  - Stores backup copies of the packages in the ST_COMMON collection.

Implied in the use of these collections is that only two versions for the same program can be kept in the system test environment: the current and backup versions.

There are two load libraries for the car dealer project at the system test level:

- ST.CURR.LOADLIB (current load library)
- ST.BACK.LOADLIB (backup load library).

The backup library is concatenated after the current load library. The prefix “ST” is used for the collections and load libraries to reflect that they belong to the system test environment.

There is only one application plan for the car dealer project in the system test environment:

- SCARDEAL application plan

  - Plan used to execute any program in the car dealer project in the system test environment only
  - Owned by the system test administrator.

System Test Privileges: The system test administrator should have the following privileges:

- PACKADM ON COLLECTION *ST_CARDEAL_CURR*
- PACKADM ON COLLECTION *ST_CARDEAL_BACK*
- EXECUTE ON PACKAGE *ST_COMMON.**
- EXECUTE ON PACKAGE *ST_COMMON_BACK.**
- BINDADD.

For testing the application, the system test administrator who owns the application plan grants the execute privilege to the testers:

- EXECUTE ON PLAN *SCARDEAL*.

System Test Plan Bind: The system administrator binds the application plan specifying all four collections:

```
PROC 0
CONTROL MSG
  DSN SYSTEM(DB23)
    BIND PLAN(SCARDEAL)                                    -
        PKLIST(ST_CARDEAL_CURR.*, ST_COMMON.*,                -
              ST_CARDEAL_BACK.*, ST_COMMON_BACK.*)            -
        ACQUIRE(USE) RELEASE(COMMIT) VALIDATE(BIND) ACTION(ADD)
  END
EXIT
```

This plan bind is done only once. Subsequent rebinds when the program changes are not required because packages are used.

The sequence of coding the package list entries is important for performance. Recall that using the collection method means that the programs cannot use the SET CURRENT PACKAGESET statement to select either the current or backup collection. DB2 picks the package from the correct collection based on the consistency token. Without the SET CURRENT PACKAGESET statement, DB2 searches each collection in the order in which it is coded in the PKLIST. Putting the collection that is most used first in the package list minimizes DB2 search overhead. The assumption here is that the ST_CARDEAL_CURR packages are the most frequently accessed followed by the ST_COMMON packages.

Promotion and Fallback (System Test): The promote and fallback process for the PARTINQ program is described below. The PARTINQ is promoted from unit test. An assumption here is that the program is subject to precompile, compile, link-edit, and bind when it is moved from unit test to system test. The modules and package are not copied from the unit test environment.

Promote

The procedure to promote a modified version of PARTINQ into system test consists of the following steps:

1. Back up the current load library to the backup library (copy from ST.CURR.LOADLIB to ST.BACK.LOADLIB).

2. Back up the current package into the backup collection. The system test administrator binds the package. A CLIST example:

```
PROC 0
CONTROL MSG
  DSN SYSTEM(DB23)
    BIND PACKAGE (ST_CARDEAL_BACK)                        -
         COPY(ST_CARDEAL_CURR.PARTINQ)                    -
         ACTION(REPLACE)                                -
         QUALIFIER(STADMIN)    OWNER(STADMIN)             -
         RELEASE(COMMIT)      EXPLAIN((YES)              -
         ISOLATION(CS) SQLERROR(NOPACKAGE) VALIDATE(BIND)
  END
EXIT
```

3. Precompile, compile, and link-edit the modified version into the current load library (ST.CURR.LOADLIB).

4. Bind the package into the current collection specifying the qualifier for the system test tables. A CLIST example:

```
PROC 0
CONTROL MSG
  DSN SYSTEM(DB23)
    BIND PACKAGE (ST_CARDEAL_CURR)                          -
         MEMBER(PARTINQ)     ACTION(REPLACE)               -
         QUALIFIER(STADMIN)   OWNER(STADMIN)                -
         RELEASE(COMMIT)     EXPLAIN(YES)                  -
         ISOLATION(CS) SQLERROR(NOPACKAGE) VALIDATE(BIND)    -
         LIBRARY ('STDB2C.STEST.DBRMLIB')
  END
EXIT
```

5. Test the program.

Fallback

Having the backup collection means that the user need not repeat the procedure of precompile, compile, bind, and link-edit when there is a need to fall back to the version before the change.

During fallback, only one action needs to be performed :

- Delete (or rename) the failed program module from the current library (ST.CURR.LOADLIB).

Recall that the current and backup program libraries are concatenated. Deleting the program load module from the current library would result in the old program module (in the backup library) being used at the next invocation of the program. This provides a very fast and efficient way of restoring execution to the old version in a fallback. This may not be as important in the system test as in the production environment.

For both performance and control reasons, the backup collection should never be used in the long term. To restore everything to what it was before the promotion:

1. Restore the program modules from the backup library to the current library.

2. Restore the package from the backup collection:

```
BIND PACKAGE (ST_CARDEAL_CURR) COPY(ST_CARDEAL_BACK.PARTINQ).
```

Production Environment

The production environment setup is similar to the system test environment setup. The environment and assumptions are described below.

The people involved in the car dealer application in the production environment are:

- End users
  - Execute the car dealer programs.
- Production administrator
  - Promotes car dealer programs from system test to production
  - Owns the tables
  - Owns the application plan called PCARDEAL
  - Manages the collections PD_CARDEAL_CURR and PD_CARDEAL_BACK
  - Binds the plan and packages for the application.

Four collections are used in the car dealer project in the production environment:

- PD_CARDEAL_CURR
  - Stores packages belonging to the car dealer project in the production environment
  - These packages are “current.” They have matching consistency tokens with the programs in the current load library for the car dealer project.
- PD_CARDEAL_BACK
  - Stores backup copies of the packages in the PD_CARDEAL_CURR for the car dealer project
  - Stores the backup packages temporarily only (as long as fallback may be required).
- PD_COMMON
  - Stores packages that are common to all projects in the production environment.
- PD_COMMON_BACK
  - Stores backup copies of the common packages in the production environment.

There are two load libraries for the car dealer project in the production environment:

- PD.CURR.LOADLIB (current load library)
- PD.BACK.LOADLIB (backup load library).

The prefix “PD” is used for the collections and load libraries to reflect that they belong to the production environment.

There is only one application plan for the car dealer project in the production environment:

- PCARDEAL application plan
  - Plan used to execute any program in the car dealer project in the production environment
  - Owned by the production administrator.

Production Privileges: The production administrator should have the following privileges:

- PACKADM ON COLLECTION *PD_CARDEAL_CURR*
- PACKADM ON COLLECTION *PD_CARDEAL_BACK*
- EXECUTE ON PACKAGE *PD_COMMON.**
- EXECUTE ON PACKAGE *PD_COMMON_BACK.**
- BINDADD.

If the application already exists, end users would have the execute privilege on the plan. Otherwise, the production administrator grants end users:

- EXECUTE ON PLAN *PCARDEAL*.

Production Plan Bind: The production administrator binds the application plan, specifying all four collections:

```
PROC 0
CONTROL MSG
  DSN SYSTEM(DB2B)
    BIND PLAN(PCARDEAL)                                          -
         PKLIST(PD_CARDEAL_CURR.*, PD_COMMON.*,                  -
              (PD_CARDEAL_BACK.*, PD_COMMON_BACK.*)              -
         ACQUIRE(USE) RELEASE(COMMIT) VALIDATE(BIND) ACTION(ADD)
  END
EXIT
```

As in system test, the package list coding sequence is important. The least frequently used collection should be coded last in the sequence. The plan does not contain DBRMs, and hence it is bound only once. Subsequent changes to programs do not require a plan rebind.

One important point to note regarding this plan is the performance consideration of having packages in collections PD_CARDEAL_CURR and PD_COMMON. The reason for having the common packages in a separate collection is to prevent duplication. However, accessing the common packages in the above plan incurs extra overhead because the SET CURRENT PACKAGESET statement is not used in the common programs for the same reason it is not used in the car dealer programs as discussed in Section “Fallback” on page 53. Every time a common package is requested, DB2 incurs the overhead of searching the PD_CARDEAL_CURR collection first. This involves a DB2 Directory (SPT01) access. Users should test this arrangement in their own environments to see whether it is acceptable. Alternatives to coding the plan that might alleviate this problem are discussed in Section “Performance” on page 65.

Promotion and Fallback (Production): The process to promote the PARTINQ program from system test to production is described below. A fallback procedure is also described. The assumption here is that the source code is ported over to production and recompiled and bound. Object or load modules from system test are not used.

Promote

Assume that PARTINQ has been changed and has gone through system testing and is now waiting to be promoted to production.

The steps to promote a modified version of PARTINQ into system test are:

1. Back up the current production load library.
2. Back up the current package into the backup collection:

```
PROC 0
CONTROL MSG
  DSN SYSTEM(DB2B)
    BIND PACKAGE (PD_CARDEAL_BACK)                        -
        COPY(PD_CARDEAL_CURR.PARTINQ)                      -
        ACTION(REPLACE)                                  -
        QUALIFIER(PDADMIN)    OWNER(PDADMIN)              -
        RELEASE(COMMIT)       EXPLAIN(YES)               -
        ISOLATION(CS) SQLERROR(NOPACKAGE) VALIDATE(BIND)
  END
EXIT
```

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

3. Precompile, compile, and link-edit the modified version into the production load library.
4. Bind the package into the current collection, specifying the qualifier for the tables:

```
PROC 0
CONTROL MSG
  DSN SYSTEM(DB2B)
    BIND PACKAGE (PD_CARDEAL_CURR)                        -
        MEMBER(PARTINQ)     ACTION(REPLACE)               -
        QUALIFIER(PDADMIN)   OWNER(PDADMIN)               -
        RELEASE(COMMIT)     EXPLAIN(YES)                 -
        ISOLATION(CS) SQLERROR(NOPACKAGE) VALIDATE(BIND)    -
        LIBRARY ('STDB2C.PROD.DBRMLIB')
  END
EXIT
```

5. Execute the program.

Fallback

The backup collection package remains in the collection until it is deemed that the modified program is stable. In a fallback, DB2 automatically picks

up the package in the backup collection when the old version of the program executes. The procedure is similar to that described for the system test environment:

- Remove the failed program load modules from the current production load library.

The next invocation of the program will execute from the backup library (because it is concatenated to the production library). DB2 does its part and subsequently allocates the package in the backup collection, which has the old consistency token.

If the old version is to run for some time, it is better, for performance reasons, to restore both the old program module and package from the backups to the current program library and collection. The procedure is:

1. Restore the old program modules from the backup to the current program library.
2. Restore the old package to the current collection:

```
BIND PACKAGE (PD_CARDEAL_CURR) COPY(PD_CARDEAL_BACK.PARTINQ)
```

Figure 27 on page 64 summarizes the setup and procedures to promote a change in program from unit test to system test and finally to production.

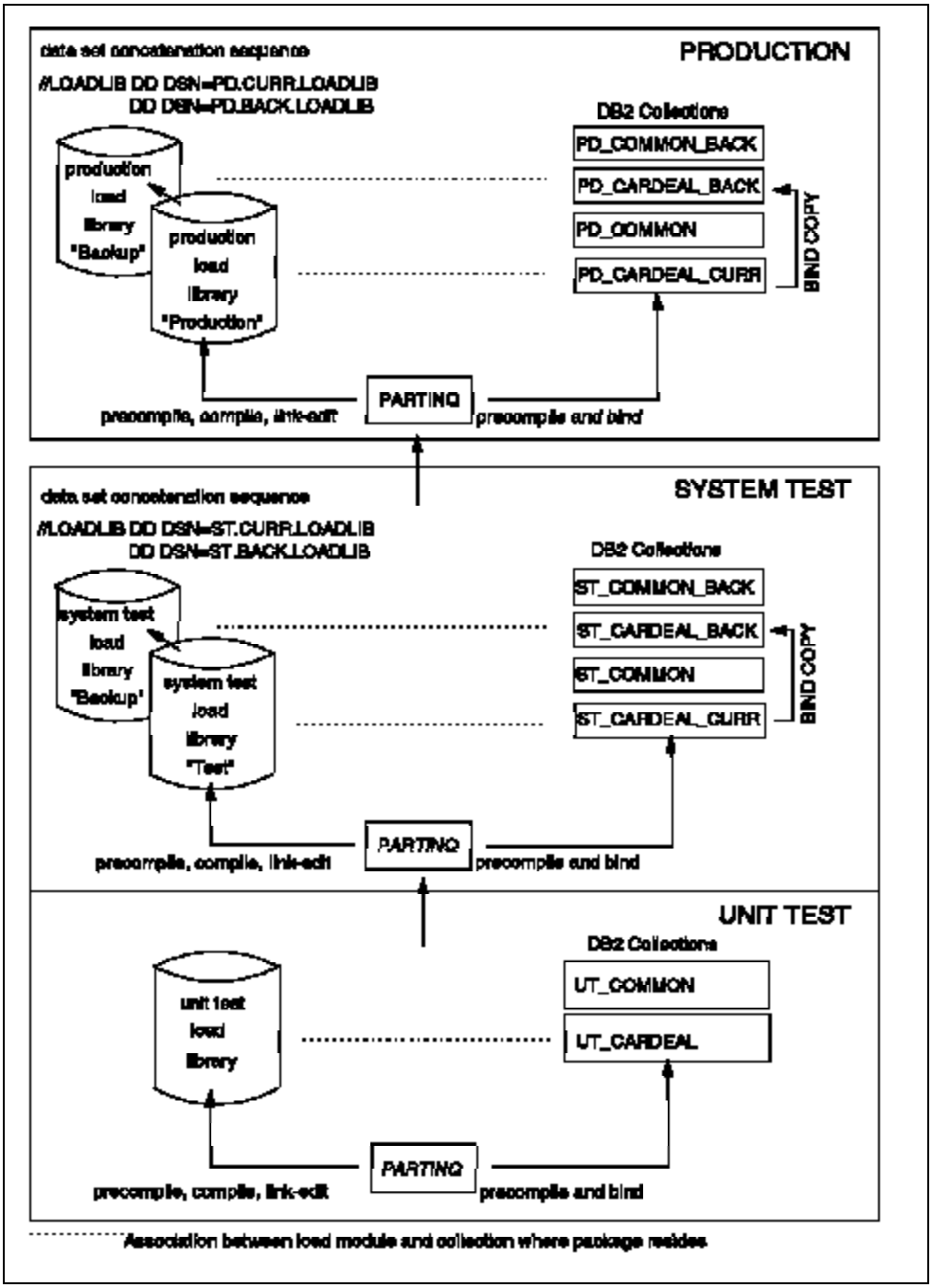

Figure 27. Promotion Path to Different Environments for PARTINQ Program

3.4.3 Considerations in Using Collections for Versioning

The user should evaluate the benefits and drawbacks of using collections as a form of versioning. The issues the user will face when using collections are discussed below.

Number of Versions

The number of versions allowed for a package is clearly limited to the number of collections. The package_id must be unique to the collection. There cannot be duplicates. In the scenario discussed in Section 3.4.2, "Using Collections for Versioning: A Scenario" on page 54, the user has only two versions to handle: the version currently in use and the backup version. Some customers may require more collections than that.

For example, while PARTINQ undergoes a scheduled test in the system test environment, there may be an emergency requirement to test out a corrected programming defect for PARTINQ. This is an additional version on top of the current and backup versions. To avoid impacting the ongoing test for PARTINQ in the system test environment, another collection may be required to hold the emergency package.

Compare the above with the VERSION precompiler option, which allows practically an unlimited number of versions (limited by the 64-character version identifier). This option may be more suitable for an environment where many versions of the same package need to be kept.

Performance

The user should be familiar with the performance considerations when coding multiple collections in a package list. Chapter 7, "Performance Implications" on page 165 discusses in detail performance issues related to packages. For the scenario in this chapter, the SET CURRENT PACKAGESET statement is not used, and the implications should be understood. The PKLIST entries must be carefully coded to ensure optimal performance. The user has many choices, some of which are:

1. PKLIST(COL.*, <back up packages> ...

   There is no common collection. Common packages are bound into the application collection COL. All package references are found in the COL collection, which is the first in the package list. There is no impact on performance due to SET CURRENT PACKAGESET not being used. Performance degrades only in a fallback when the packages referenced are in the second and subsequent entries in the package list. The compromise here is that packages cannot be shared. When a common package changes, it must be bound into all collections using it.

2. PKLIST(COL.<package_ids>, <back up packages> ...

   There is no common package. Instead of using a wild card for the package_ids, the package_ids are specifically coded. In a fallback, performance degradation is **similar** to the degradation incurred by using "*" for package_id. DB2 can find a package in COL collection with a matching package_id, but it has to access the DB2 Directory (SPT01) before finding out that the consistency token does not match (unless

that package was allocated in the plan in the EDM pool-which may be unlikely in the case of a fallback). Then DB2 proceeds with the rest of the package list and finds the backup package with the matching consistency token. There may also be a practical restriction, namely, the number of packages in the collection. That depends on the scope of the collection as discussed in Section 3.1, “Guidelines for Collection Usage” on page 43.

3. PKLIST(COL.*, COMMON.*, <back up packages>) ...

   This is the PKLIST entry used in the scenario. The user must test whether the performance overhead when using common packages is acceptable. Whenever a common package is referenced, DB2 does not know which collection it is in and consequently has to search the first collection before searching the common collection. The former search involves a DB2 Directory (SPT01) access. In the case of a fallback, there is the double overhead of searching the first two entries of the package list.

4. PKLIST(COL.<package_ids>, COMMON.*, <back up packages> ...

   Coding the package_ids specifically for the application would solve the performance overhead when common packages are accessed. DB2 can check from the package list entry that the package_id for the common package is not found in the COL collection, and therefore it need not search the DB2 Directory (SPT01) for packages in the COL collection. (This assumes that the package_ids for common programs are different from the package_ids for the applications.)

5. PKLIST(<high use packages>, COL.*, COMMON.*, <back up packages> ...

   To further minimize the performance impact of using “*,” the user can selectively pick frequently used common packages and code them specifically at the beginning of the package list.

6. PKLIST(COMMON.*, COL.*, <back up packages>) ...

   The user must determine whether the plan frequently invokes the common packages, in which case the common collection entry should be coded first in the package list.

The user must also be aware of the performance degradation in a fallback as discussed in Section “Fallback” on page 53.

Change in Access Path

There is an element of risk in copying the in-use package to the backup collection. The bind with copy version reevaluates the access path just like a normal bind. Therefore, a backup copy of a DB2 package may not be an exact duplicate of the original package. There is no way to bypass the access path evaluation. In a fallback, when the backup collection is used, the access path may be different from the original if, for example, new indexes have been added or catalog statistics have changed. The access path may also change when software fixes are applied to the DB2 optimizer since the package was last bound. It is thus crucial that access paths be checked after the bind copy. Therefore, history information must be kept for the EXPLAIN output for the package so that future EXPLAIN output can

be compared. Certain DB2 tools may help in this case. One such product is Candle Corporation's !DB/EXPLAIN Version 230**.

A new package, for example, for a promoted program can be bound into a temporary collection rather than directly into the current collection. Once it is verified that the program performs correctly, the package is bound into the permanent collection, replacing the old version. For example, the bind for the plan would be:

```
BIND PLAN(planname) PKLIST(APPL.*, APPL_TEMP.*)
```

APPL is the permanent collection for the packages belonging to the application. All packages in the APPL collection are currently used by the programs. When a new program is to be promoted into this environment, it is bound into collection APPL_TEMP, which temporarily stores the new package. Once the package is tried and tested, it is bound into the APPL collection, replacing the old version. This method does not require the old version to be backed up using bind copy. Therefore, in a fallback, there is no change in access path because the old version is untouched. However, when the package is eventually bound into the APPL collection when it is tried and tested, the access path may be different from the access path of the package in the APPL_TEMP collection. Any difference will of course depend on what has changed between the time of the two binds.

An alternative to versioning using collections would be to use the precompiler option VERSION. This option allows multiple versions of the same package to exist in the same collection. No backing up of a package is required during program promotion (see Chapter 4, “Using the VERSION Keyword” on page 69).

Procedures

Before DB2 2.3, the promotion of a modified program into the production environment typically required that the new version be bound into a new plan (with a different name). This procedure will not be used at all if DB2 collections are used as described in this section.

New procedures have to be set up to implement the collection method as described earlier in this chapter. The procedures include promotion, fallback, and freeing of backups after a successful promotion. The procedure must handle the length of time the package and load modules remain in the backup collection or load library. The backups should be kept until the new modules and packages have been subjected to all processes. For example, if the application is subject to different daily and month-end processes, the backup packages must remain until the month-end run is completed before they are freed.

No Mirror Table Support

This method of program versioning using different collections does not support mirror table programs that use the SET CURRENT PACKAGESET statement. In this case, the precompiler option VERSION may have to be used for versioning support.

Naming Conventions

Naming standards should be set up for easy manageability of collections and packages. A sample naming convention may have all or a subset of the following items:

- Environment (for example, TEST, PROD)
- Business area
- Application name (CARDEAL)
- Subapplication name
- Function (BACK for backup).

Why Use Collections for Application Versioning?

The benefits of using collections for application versioning are:

- The migration procedure for packages is similar to the migration procedures for load modules. Tools can be easily developed to handle the backup of the package into the backup collection.
- There is no VERSION precompiler option to worry about. The option allows the user too much flexibility (unless automated) in creating multiple instances of the same package. This may cause an accumulation of instances of the same package.
- The use of collections offers a way to restrict the number of versions or instances of a package. To create a new instance of a package, the administrator would have to grant CREATE IN *collection_id* to the new collection. The need for an authorization to use a new collection automatically prevents the creation of too many versions of the package.

Chapter 4. Using the VERSION Keyword

Supporting multiple versions of a program is one of the benefits derived from using packages. This chapter describes the concept of program versioning using the VERSION keyword and presents some application promotion scenarios in which this new function is implemented.[1] Using VERSION requires the version_id to be specified at precompile time and not bind time, which is significantly different from using collections to support multiple versions of a program. Note that VERSION is supported only in a DB2 environment. It cannot be implemented in a distributed environment.

4.1 VERSION Keyword

The version_id supplied at precompile time through the VERSION keyword becomes the attribute of the package name. (The version_id is an empty string if VERSION keyword is not used.) Using VERSION allows packages with the same location name, collection_id, and package_id to be stored in the same collection. A change in the version_id requires that the program be precompiled. While the version_id supplied is stored in both the DBRM and the program source, the DBRMs for two versions of the same module must be stored in different libraries if the input to the bind statements is to be retained. There is no version concept with DBRMs.

When packages are stored in DB2, each version of a package has a different consistency token, and each package with a different consistency token has a different version_id. However, it is the consistency token, not the version_id, that DB2 always uses during package selection at execution time.

Use of the LEVEL precompiler option with the VERSION option is not recommended. The LEVEL option was introduced to reduce binding costs by allowing programs with unchanged SQL statements to be recompiled and run without a rebind of the plan. When packages are used, there is no need to rebind the plan as previously and hence a reduced need for the LEVEL option. Table 7 on page 70 shows the relationship between the version_id and consistency token where both options are used.

[1] To avoid confusion, the word *version* in uppercase letters refers to the precompiler option, VERSION, whereas the word *version* in lowercase letters refers to the general concept of versioning. The word *version_id* refers to the package attribute.

Table 7. Interplay of VERSION and LEVEL Keywords			
Precompiler Options		**Resulting Package Values**	
VERSION	LEVEL	Consistency Token	Version_id
Not specified	UNITEST	UNITEST	Empty string
AUTO	UNITEST	UNITEST	UNITEST
CN 12345	UNITEST	UNITEST	CN 12345
Not specified	Not specified	Time stamp	Empty string
AUTO	Not specified	Time stamp (see note)	1992-11-16-14.39.245638
CN 12345	Not specified	Time stamp	CN 12345

Note:

This is the ISO 26-character-format time stamp converted from the consistency token when the VERSION parameter is set to AUTO.

The value assigned to the VERSION keyword should be chosen carefully because the value specified must be reproduced exactly when freeing or rebinding a package. The version_id can be specified in three ways:

- AUTO

  When AUTO is specified, the version_id is generated from the consistency token. If the consistency token is set using the precompiler LEVEL option, the level value is used as the version identifier. If the consistency token is a time stamp, the time stamp is converted to ISO character format and used as the version identifier.

  The use of AUTO with a time stamp consistency token is one of the recommended ways of using the VERSION option because:

  - A new version identifier is generated **automatically** each time the program is prepared.
  - The modified time stamp that is generated allows the versions to be sorted on date and time so that the sequence in which the versions were created can be easily identified. This would assist in identifying the older versions for freeing or dropping.

  The major drawback of this approach is that one cannot control the number of versions of a package that can be generated. A programmer may go through many iterations before getting the program to function correctly. In addition, freeing or rebinding a package requires that the version identifier be entered in its entirety. As the time stamps generated in this way are 26 characters in length, considerable effort would be required to free versions (in the absence of a suitable tool). Another drawback of this approach is that the version identifiers generated do not provide any indication of the reason for the new version.

- A constant

  Constants can be used to set the version identifier in a number of circumstances. For example, you can use a change number that refers to the number assigned to the documentation associated with the change, or a constant such as UNIT_TEST, which refers to the environment to which the version applies. As the version identifier can be up to 64 characters long, text could also be added to provide a better description of the reason for the new version. However, long comments are likely to increase the effort to free or rebind a package when the full version_id can be specified.

  The major drawback of this approach is that only one such version can be bound to a collection. Any attempt to bind a second version without changing the version identifier is rejected.

- A combination of the above

  A combination of the time stamp provided by AUTO and a change number that refers to documentation associated with the change would seem to present the best solution. Such an identifier would provide an automatic change in the version identifier because of the time stamp as well as a means of associating the version with the change documentation.

  This approach would require a procedure that builds the version identifier and appends it to the VERSION keyword when the precompiler is first invoked.

4.2 Package Versioning Using the VERSION Keyword

Developers may wish to support multiple versions of an application within a DB2 subsystem for several reasons. The most obvious one is the support of multiple test environments within the one DB2 subsystem. Before DB2 V2.3, different plan names and different DBRM libraries were used to support and distinguish between two or more environments.

The use of versioning would allow two versions of a program, say PROGRAM1, to be stored in the same collection, one for unit test and the other for system test. Whether this represents a realistic use of versioning depends on the existing environment. The two test environments would require different load libraries and DBRM libraries if the DBRMs are to be retained. For versioning to work, the programs must be precompiled, compiled, and linked in each of the application environments. For the unit test environment, for example, the version-id could be UNITV1R01 with the last five digits varying as new versions of the program code are developed. Similar values for the version parameter would be supplied at precompile time for each environment and for each program.

Different plan names with the same structure could be used for each environment; that is, the plans for the different environments would have the same package list, and within a collection there would be packages for the same program with different version_ids of the form UNITV1R01 and SYSTV1R01. At execution time, DB2 identifies the appropriate package from the package_id and the consistency token because only one of possibly

many versions will have a value that matches that of the executing program.

The use of versioning is more applicable to the support of multiple versions of a program within one environment. For example, where program versioning and promotion are controlled through a library management product, package versioning provides complementary support. For example, multiple versions of a program and its package may need to be supported during planned program maintenance, and this is best handled using package versioning.

Another use of versioning is in the fallback situation mentioned previously. The load module must be backed up to another library. However, where package versioning is used, there is no need to back up the package itself, because two packages with different version_ids can be stored in the same collection. The use of package versioning removes the need to have a separate fallback collection as both versions of the package can be bound into the same collection. Whichever module, backup or current, is executed, DB2 selects the corresponding package with matching consistency token. (See Figure 28.)

A point to note in the use of package versioning: there is no effective means of limiting the number of versions that can be generated. Thus unwanted growth of the SYSIBM.SYSPACKAGE table may result.

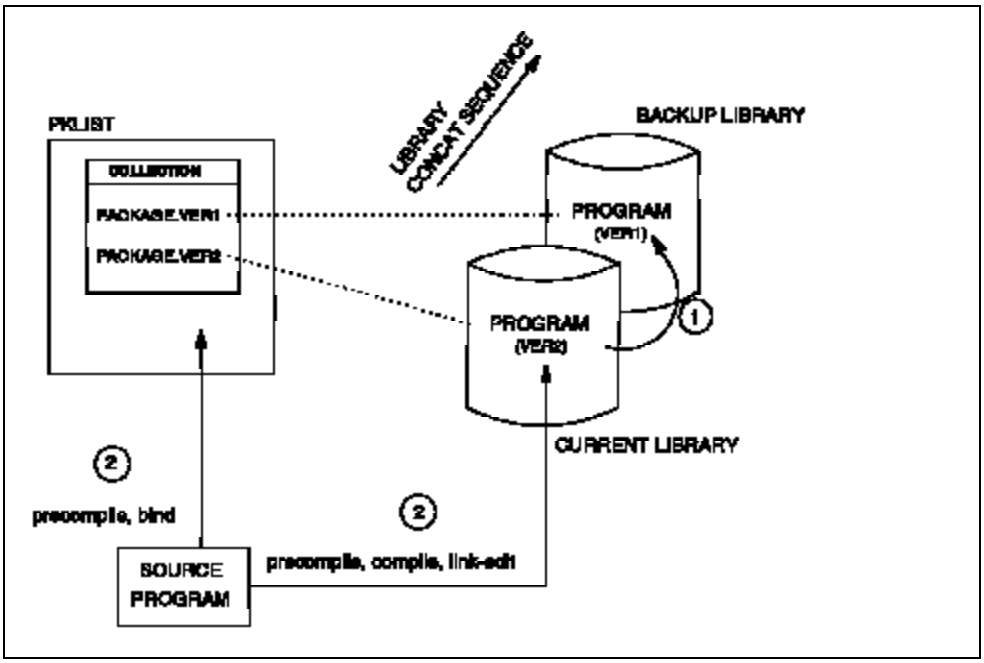

Figure 28. Versions and Program Libraries

4.3 Application Promotion Scenarios

This section describes the sample scenarios that use VERSION in application promotion.

Each scenario has three phases: unit test, system test, and production. Each phase has separate libraries for the current executing program and the backup program. The backup library is concatenated behind the current library at execution time. (See Figure 28 on page 72.) In case of fallback, the program in the backup library can be executed by deleting or renaming the program in the current library, so that DB2 selects the corresponding backup package.

We consider two different perspectives for the version naming convention. One perspective is to use the same constant version_id (for example, VER1) throughout the promotion phases. Using a document or change number as the version_id might be good in this case, in that it is easy to identify to which document the module should be referred. This scenario is referred to as Scenario 1.

The other perspective is to change the version_id as the program is promoted (for example, VER1timestampA, which includes a constant value and a time stamp). This scenario is referred to as Scenario 2.

4.3.1 Scenario 1: Identical Version_id

The environment for this scenario is the same as that in Section 3.4.2, “Using Collections for Versioning: A Scenario” on page 54: one DB2 system is used for unit test and system test, and one DB2 system for production. The sample program is also the same: the PARTINQ program in the car dealer application.

Figure 29 on page 74 illustrates the current environment.

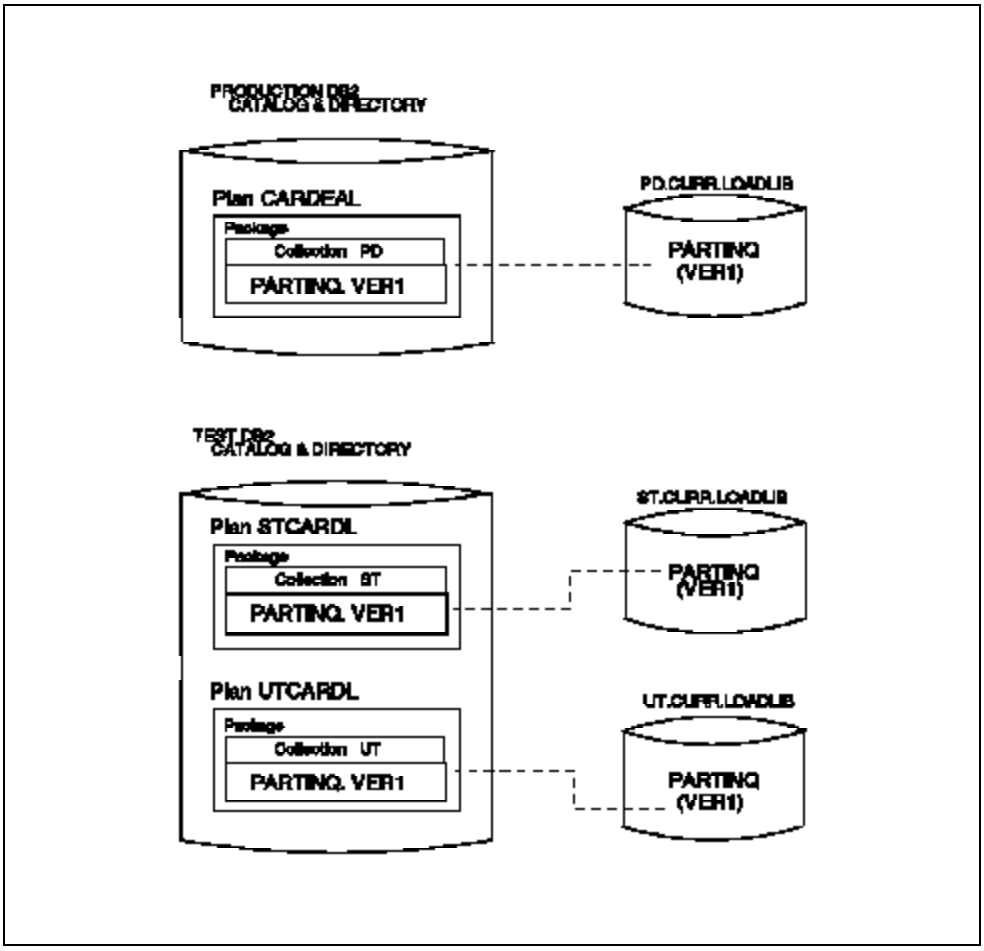

Figure 29. Current Environment for Scenario 1

The version_id of the PARTINQ program is VER1. Because the version_id stays the same throughout the promotion phases, the package in each phase must be bound to a different collection. All programs, including the backup, are bound into collections UT for unit test, ST for system test, and PD for production. The plan name is UTCARDL in the unit test, STCARDL in the system test, and CARDEAL in the production environment. A separate plan for unit test and system test is not mandatory; however, in Scenario 1, for reasons of security, we have separated plans for unit test and system test.

The scenario starts with the modification of the PARTINQ program. The new version_id for the modified program is VER2 for the entire process of promotion from unit test to production. Because multiple load libraries are required to keep the different versions of the load modules, two libraries are set up for each phase (for example, for production, PD.CURR.LOADLIB for the current program, and PD.BACK.LOADLIB for the backup program). However, a backup library may not be necessary in unit the test environment.

Figure 30 on page 75 illustrates the overall environment after promotion. Figure 31 on page 76 presents an overview of the promotion procedures in Scenario 1. Each phase is described in detail below.

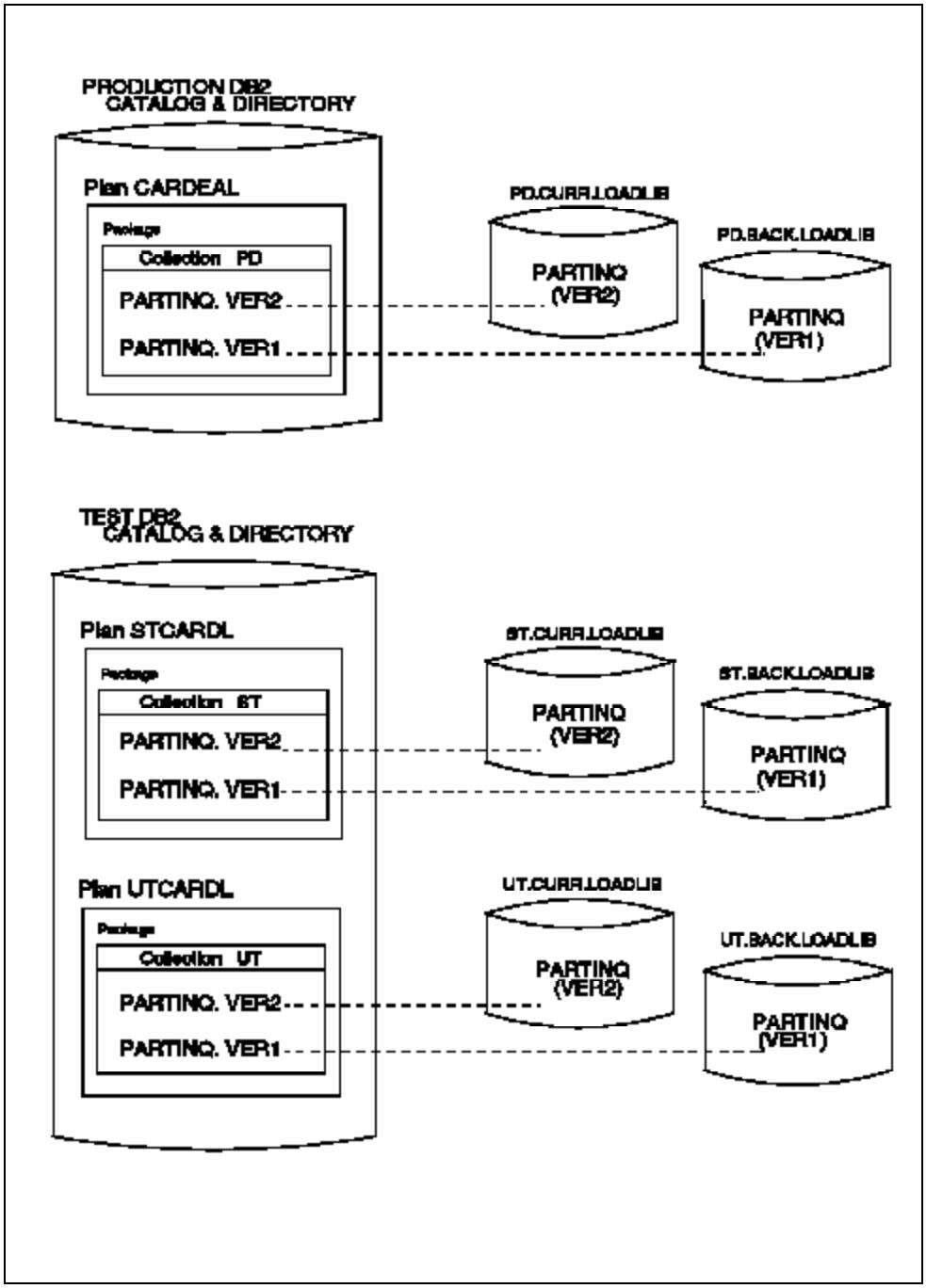

Figure 30. Environment after Modification of Program in Scenario 1

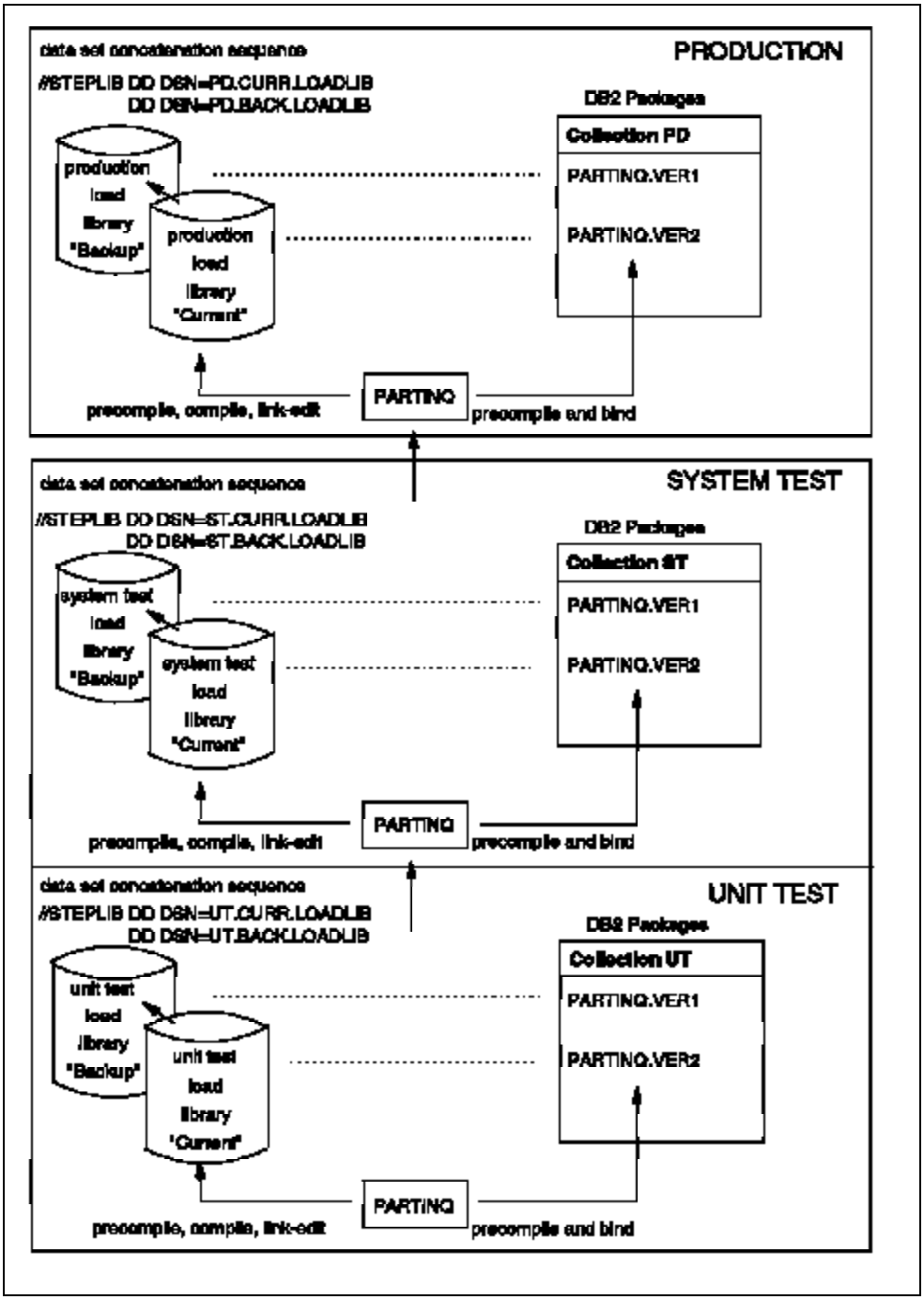

Figure 31. Application Promotion: Scenario 1

Unit Test

Assumption

In the unit test, it is assumed that the programmer and the unit test administrator play the following roles:

- Programmer
  - Owns tables
  - Prepares source code
  - Precompiles, compiles, and link-edits the module
  - Binds package and owns package
  - Executes and tests the program.
- Unit test administrator
  - Binds plan and owns plan.

The programmer has no privileges at this point except for those implicitly given as the table owner.

The unit test administrator has the following privileges:

- PACKADM authority on collection UT with grant option
- BINDADD privilege with grant option.

Promotion Procedure

The promotion procedure to the unit test is to:

1. Copy the previous version of the program to the backup library if necessary.
2. After preparing the source code, precompile the modified PARTINQ program specifying the VERSION keyword as VER2 and the DBRM library as the unit test DBRM library (UT.CURR.DBRMLIB). Then compile and link-edit the program to the unit test load library (UT.CURR.LOADLIB). The CLIST to invoke the precompiler, compiler, and linkage editor is:

```
PROC 0
   %DSNH+
     PRECOMP(YES)+
     COMPILE(YES)+
     LINK(YES)+
     BIND(NO)+
     INPUT('''UT.CURR.SRCELIB(PARTINQ)''')+
     DBRMLIB('''UT.CURR.DBRMLIB(PARTINQ)''')+
     LOAD('''UT.CURR.LOADLIB''')+
     VERSION(VER2)+
     .............
END
```

   The programmer need not have any DB2 authorities up to this point.

3. Bind the package in the collection UT. The bind package subcommand is:

```
BIND PACKAGE(UT) OWNER(programmer) QUALIFIER(programmer)
    MEMBER(PARTINQ) LIBRARY(UT.CURR.SRCELIB)
    VALIDATE(BIND) ACTION(ADD) ......
```

Here the programmer must have the following privileges to create a new version of the package, own it, and execute it:

- BINDADD privilege
- CREATE IN privilege for collection UT
- Privilege to execute all SQL statements in the DBRM
- EXECUTE privilege on the plan UTCARDL.

Of these four required privileges, the programmer already holds the privilege to execute all SQL statements in the DBRM because the programmer owns the tables accessed in the program.

The unit test administrator would grant the programmer the three remaining privileges as follows:

```
GRANT BINDADD TO programmer   ;
GRANT CREATE IN COLLECTION UT TO programmer   ;
GRANT EXECUTE ON PLAN UTCARDL TO programmer   ;
```

If the plan were to be bound, the bind subcommand would look like this:

```
BIND PLAN(UTCARDL) PKLIST(UT.PARTINQ, ....) .......
```

or

```
BIND PLAN(UTCARDL) PKLIST(UT.*) .......
```

However, because this is a modification of an existing program, the plan UTCARDL containing the package name UT.PARTINQ already exists. So there is no need to bind the plan again. The package entry included in the PKLIST of a plan implies all versions of the package.

Figure 32 on page 79 shows the DB2 catalog after the programmer has bound the package for the modified PARTINQ program.

```
 < SYSIBM.SYSPACKAGE >
COLLID|NAME   |CONTOKEN|OWNER  |CREATOR|QUALIFIER|VERSION| ......
------ ------- -------- ------- ------- --------- ------- -------
  UT  |PARTINQ|.M ...ï¥|prgrmr1|prgrmr1| prgrmr1 |  VER1 |
  UT  |PARTINQ|.M «..åf|prgrmr2|prgrmr2| prgrmr2 |  VER2 |    -----
  ST  |PARTINQ|.M   .åâû|stadm  |stadm  | stadm   |  VER1 |


 < SYSIBM.SYSPACKAUTH >
GRANTEE|COLLID|NAME   |BINDAUTH|COPYAUTH|EXECUTEAUTH| ......
------- ------ ------- -------- -------- ----------- -------
prgrmr1|  UT  |PARTINQ|   G    |   G    |    G     |
prgrmr2|  UT  |PARTINQ|   G    |   G    |    G     |     -----
stadm  |  ST  |PARTINQ|   G    |   G    |    G     |


 < SYSIBM.SYSPLANAUTH >
GRANTEE|NAME   |BINDAUTH|EXECUTEAUTH| ......
------- ------- -------- ----------- -------
utadm  |UTCARDL|   G    |    G     |       -----
prgrmr1|UTCARDL|        |    Y     |
prgrmr2|UTCARDL|        |    Y     |       -----
stadm  |STCARDL|   G    |    G     |


 < SYSIBM.SYSRESAUTH >
GRANTEE|QUALIFIER|NAME|USEAUTH| ......
------- --------- ---- ------- -------
utadm  | PACKADM | UT |   G   |    -----
prgrmr1|         | UT |   Y   |
prgrmr2|         | UT |   Y   |    -----
stadm  | PACKADM | ST |   G   |


 < SYSIBM.SYSUSERAUTH >
GRANTEE|BINDADDAUTH| ......
------- ----------- -------
utadm  |    G     |       -----
prgrmr1|    Y     |
prgrmr2|    Y     |       -----
stadm  |    G     |
```

Figure 32. DB2 Catalog: Scenario 1, Unit Test. Prgrmr1 is the programmer for the previous version of the program, VER1; prgrmr2 is the programmer for the modified program, VER2; utadm is the unit test administrator; and stadm is the system test administrator.

After executing and testing the program, if it is necessary to modify the source code again, follow this procedure:

1. Precompile the program with the same version_id, VER2, and compile and link-edit. This replaces the DBRM in the DBRM library and the load module in the load library.

2. Bind the package again with the ACTION(REPLACE) option.

3. Execute the program again.

System Test

Assumption

In the system test, it is assumed that the tester and the system test administrator play the following roles:

- Tester
  - Executes and tests the program.
- System test administrator
  - Promotes from unit test to system test
  - Owns tables.

The tester has no privileges at this point.

The system test administrator has the following privileges:

- PACKADM authority on collection ST with grant option
- BINDADD privilege with grant option.

Promotion Procedure

The promotion procedure from unit test to system test is to:

1. Copy the previous version of the source module, DBRM, and load the module from the current system test libraries to the backup system test libraries.
2. Copy the source module from the current unit test library (UT.CURR.SRCELIB) to the current system test library (ST.CURR.SRCELIB).
3. Precompile the program, specifying the VERSION keyword as the same value specified in the unit test, VER2, and using the current system test DBRM library (ST.CURR.DBRMLIB). Compile and link-edit the program into the current system test load library (ST.CURR.LOADLIB).
4. Bind the package using collection ST:

```
BIND PACKAGE(ST) OWNER(system-test-administrator)
    QUALIFIER(system-test-administrator)
    MEMBER(PARTINQ) LIBRARY(ST.CURR.DBRMLIB)
    ACTION(ADD) VALIDATE(BIND) ....
```

This procedure basically repeats the unit test procedure. It requires precompiling, compiling, link-editing, and binding a new package. So, the required DB2 authorization is the same as that for the unit test procedure. As you can see from the assumption, the system test administrator has all the required authorities and privileges: PACKADM authority on collection ST, the BINDADD privilege, and the table owner privilege. Figure 33 on page 81 shows the DB2 catalog after the new version of the package is bound. In this figure, sttster is the tester.

```
 < SYSIBM.SYSPACKAGE >
COLLID|NAME   |CONTOKEN|OWNER  |CREATOR|QUALIFIER|VERSION| ......
------ ------- -------- ------- ------- --------- ------- ------
  UT  |PARTINQ|.M ...ï¥|prgrmr1|prgrmr1| prgrmr1 |  VER1 |
  UT  |PARTINQ|.M «..åf|prgrmr2|prgrmr2| prgrmr2 |  VER2 |
  ST  |PARTINQ|.M  ?..7 |stadm  |stadm  | stadm   |  VER1 |     -----
  ST  |PARTINQ|.M   .åâû|stadm  |stadm  | stadm   |  VER2 |     -----


 < SYSIBM.SYSPACKAUTH >
GRANTEE|COLLID|NAME   |BINDAUTH|COPYAUTH|EXECUTEAUTH| ......
------- ------ ------- -------- -------- ----------- -------
prgrmr1|  UT  |PARTINQ|   G    |   G    |     G     |
prgrmr2|  UT  |PARTINQ|   G    |   G    |     G     |
stadm  |  ST  |PARTINQ|   G    |   G    |     G     |     -----
stadm  |  ST  |PARTINQ|   G    |   G    |     G     |     -----
```

Note: There are two entries for stadm because stadm owns two versions of the PARTINQ package.

```
 < SYSIBM.SYSPLANAUTH >
GRANTEE|NAME   |BINDAUTH|EXECUTEAUTH| ......
------- ------- -------- ----------- -------
utadm  |UTCARDL|   G    |     G     |
prgrmr1|UTCARDL|        |     Y     |
prgrmr2|UTCARDL|        |     Y     |
sttster|STCARDL|        |     Y     |        -----
stadm  |STCARDL|   G    |     G     |        -----


 < SYSIBM.SYSRESAUTH >
GRANTEE|QUALIFIER|NAME|USEAUTH| ......
------- --------- ---- ------- -------
utadm  | PACKADM | UT |   G   |
prgrmr1|         | UT |   Y   |
prgrmr2|         | UT |   Y   |
stadm  | PACKADM | ST |   G   |    -----


 < SYSIBM.SYSUSERAUTH >
GRANTEE|BINDADDAUTH| ......
------- ----------- -------
utadm  |    G     |
prgrmr1|    Y     |
prgrmr2|    Y     |
stadm  |    G     |        -----
```

Figure 33. DB2 Catalog: Scenario 1, System Test

It is not necessary to bind the plan again, because this scenario is based on the modification of an existing program. It is assumed that the plan had already been bound as follows:

```
BIND PLAN(STCARDL) PKLIST(ST.*) .......
```

To execute the program, the tester must be granted the EXECUTE privilege on the plan STCARDL:

```
GRANT EXECUTE ON PLAN STCARDL TO tester
```

Refer to Figure 32 on page 79 for the related DB2 catalog.

Fallback Procedure

The fallback to the previous version simply requires that the load module in the backup library be executed. DB2 automatically selects the backup package by matching its consistency token in the load module. Executing the backup module can be done easily by deleting or renaming the new version of the program in the current library (ST.CURR.LOADLIB) or replacing it with the backup module in ST.BACK.LOADLIB. It is not necessary to bind the package or plan again. However, be aware that the new version of the package is stored in DB2 until freed or replaced. You might want to copy other data sets as well, such as the source module and DBRM from the backup library, to maintain integrity between the source, DBRM, and load module.

Production

Assumption

Production runs on a different DB2 system from that used for the test environment. Both DB2 subsystems are interconnected through Distributed Data Facility (DDF). In production, it is assumed that the end user, application owner, and system administrator play the following roles:

- End user
  - Executes the program.
- Application owner
  - Owns tables
  - Owns plan for the application
  - Owns package for the application.
- System administrator [2]
  - Promotes from system test to production
  - Binds package
  - Binds plan.

Let us assume that the end user already has the EXECUTE privilege on the plan CARDEAL. The application owner has the implicit privileges of owning the PD.PARTINQ package, CARDEAL plan, related tables, and EXECUTE privilege on PD.*. The system administrator has SYSCTRL authority.

---

[2] The system administrator is simply an administrator for DB2 resources in this scenario and does not necessarily have SYSADM authorization.

Promotion Procedure

The promotion procedure from system test to production is to:

1. Copy the previous version of the source module, DBRM, and load module from the current production libraries to the backup production libraries for fallback.

2. Copy the source module from the current system test library (ST.CURR.SRCELIB) to the current production library (PD.CURR.SRCELIB).

3. Precompile the program specifying the VERSION keyword as the same value specified in the system test, VER2, and using the current production DBRM library (PD.CURR.DBRMLIB). Compile and link-edit the program into the current production load library (PD.CURR.LOADLIB).

4. Bind the package using collection PD:

```
BIND PACKAGE(PD) OWNER(application-owner)
    QUALIFIER(application-owner)
    MEMBER(PARTINQ) LIBRARY(PD.CURR.DBRMLIB)
    ACTION(ADD) VALIDATE(BIND) ....
```

This procedure is the same as that used in the system test promotion procedure.

The system administrator has adequate privilege to bind the package. Figure 34 on page 84 illustrates the DB2 catalog after the package has been bound.

```
 < SYSIBM.SYSPACKAGE >
COLLID|NAME   |CONTOKEN|OWNER  |CREATOR|QUALIFIER|VERSION| ......
------ ------- -------- ------- ------- --------- ------- ------
  PD  |PARTINQ|.NT .+ö.|aplownr|systadm| aplownr |  VER1 |
  PD  |PARTINQ|.NT[.‰Ñ*|aplownr|systadm| aplownr |  VER2 |


 < SYSIBM.SYSPACKAUTH >
GRANTEE|COLLID|NAME   |BINDAUTH|COPYAUTH|EXECUTEAUTH| ......
------- ------ ------- -------- -------- ----------- -------
aplownr|  PD  |PARTINQ|   G    |   G    |    G     |
aplownr|  PD  |  *    |        |        |    Y     |
aplownr|  PD  |PARTINQ|   G    |   G    |    G     |


 < SYSIBM.SYSPLANAUTH >
GRANTEE|NAME   |BINDAUTH|EXECUTEAUTH| ......
------- ------- -------- ----------- -------
enduser|CARDEAL|        |    Y     |
aplownr|CARDEAL|   G    |    G     |


 < SYSIBM.SYSUSERAUTH >
GRANTEE|SYSCTRLAUTH| ......
------- ----------- -------
systadm|    Y     |
```

Figure 34. DB2 Catalog: Scenario 1, Production. Enduser is the end user; aplownr is the application owner; and systadm is the system administrator.

It is not necessary to bind a plan again in production, assuming that the plan already has been bound for the previous version of the program:

```
BIND PLAN(CARDEAL) PKLIST(PD.*) ......
```

After either promotion procedure has completed, the new version of the PARTINQ program is available to end users.

System administrators should free the old version of the package at an appropriate time.

Fallback Procedure

If fallback is required, simply replace the PARTINQ load module in PD.CURR.LOADLIB with the old version of the load module backed up in PD.BACK.LOADLIB, or either delete or rename the new version in PD.CURR.LOADLIB. There is no need to bind the plan or package again. DB2 picks up the appropriate package by matching the consistency token in the load module. Free the failed version of the package at an appropriate time.

Alternative Procedure Using BIND COPY Option

Instead of the precompile, compile, and link-edit process described above, the load module and package can be copied from the previous phase to the next phase using the BIND COPY option. This procedure is possible only because the version_id remains the same throughout all of the promotion phases. Figure 35 on page 86 presents an overview of the promotion procedures using the BIND COPY option. A description of these procedures from unit test to system test and from system test to production follows.

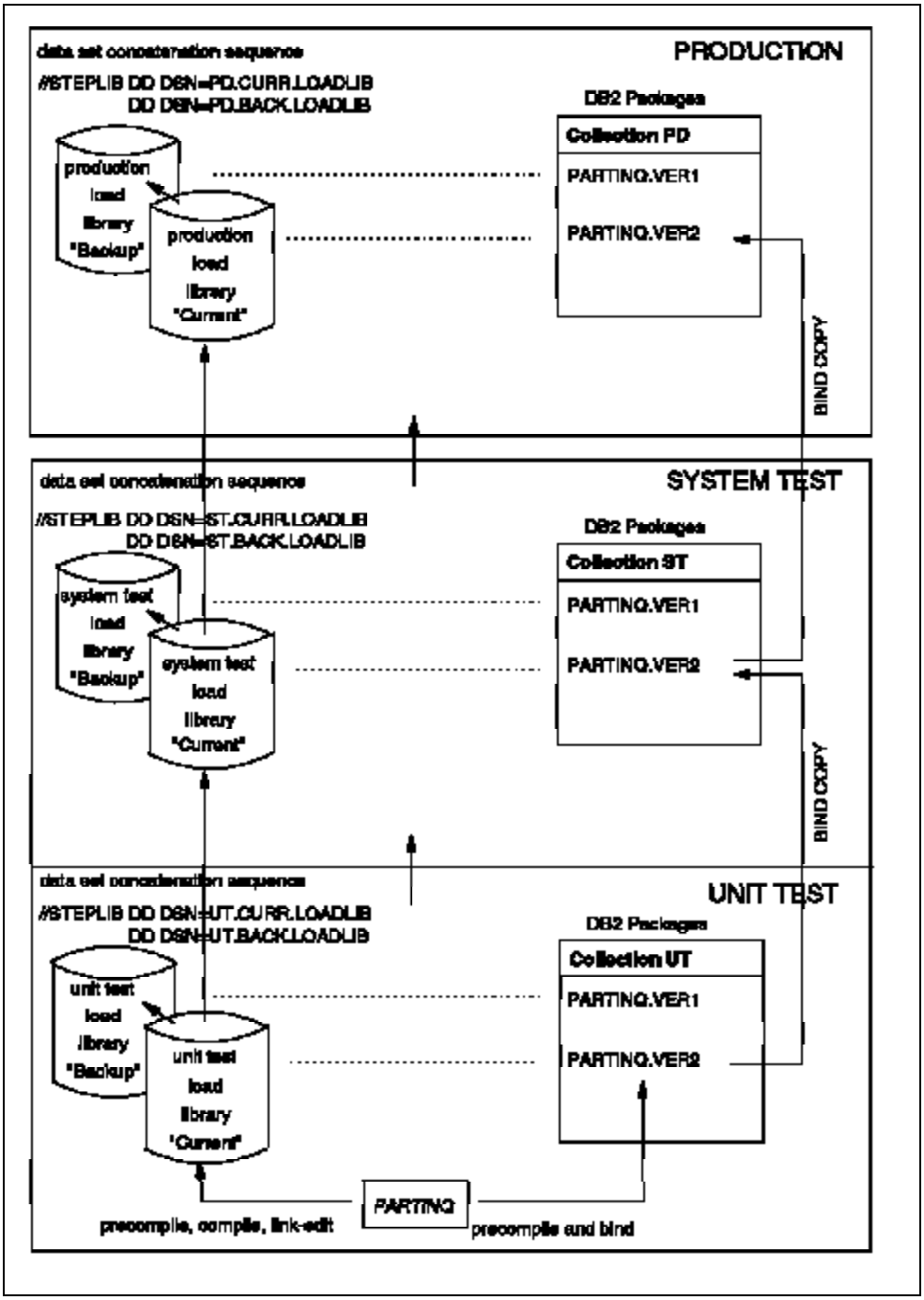

Figure 35. Promotion: Scenario 1, Using BIND COPY Option

The promotion procedure from unit test to system test using BIND COPY is to:

1. Copy the previous version of the load module from the current system test library (ST.CURR.LOADLIB) to the backup system test library (ST.BACK.LOADLIB).

2. Copy the load module from the current unit test library (UT.CURR.LOADLIB) to the current system test library (ST.CURR.LOADLIB).

3. Copy the UT.PARTINQ.VER2 package to ST.PARTINQ.VER2 using the following subcommand:

```
BIND PACKAGE(ST) OWNER(system-test-administrator)
    QUALIFIER(system-test-administrator)
    COPY(UT.PARTINQ) COPYVER(VER2)....
```

4. Copy the source from the current unit test library (UT.CURR.SRCELIB) to the current system test library (ST.CURR.SRCELIB) to store the corresponding source of the load module. (This step is optional.)

In this procedure, the COPY privilege for the UT.PARTINQ package is required in addition to the privileges the system test administrator already holds. (See Section "System Test" on page 80.) So the system test administrator must be granted this privilege:

```
GRANT COPY ON PACKAGE UT.* TO system-test-administrator
```

The related DB2 catalog is the same as that shown in Figure 33 on page 81, with two exceptions: the SYSIBM.SYSPACKAGE table and the SYSIBM.SYSPACKAUTH table. In Figure 36, the value C in the REMOTE column indicates that the new version of the package was created by BIND COPY. A row indicating that the system test administrator was granted the COPY privilege on packages has been added.

```
 < SYSIBM.SYSPACKAGE >
COLLID|NAME   | .... |VERSION|REMOTE| ......
------ ------- ------ ------- ------ -------
  UT  |PARTINQ| .... |  VER1 |  N   |
  UT  |PARTINQ| .... |  VER2 |  N   |
  ST  |PARTINQ| .... |  VER1 |  N   |
  ST  |PARTINQ| .... |  VER2 |  C   |    -----


 < SYSIBM.SYSPACKAUTH >
GRANTEE|COLLID|NAME   |BINDAUTH|COPYAUTH|EXECUTEAUTH| ......
------- ------ ------- -------- -------- ----------- -------
prgrmr1|  UT  |PARTINQ|   G    |   G    |    G     |
prgrmr2|  UT  |PARTINQ|   G    |   G    |    G     |
stadm  |  ST  |PARTINQ|   G    |   G    |    G     |
stadm  |  UT  |   *   |        |   Y    |          |     -----
stadm  |  ST  |PARTINQ|   G    |   G    |    G     |
```

Figure 36. DB2 Catalog: Scenario 1, System Test, Using BIND COPY Option

The promotion procedure from system test to production using BIND COPY is to:

1. Copy the previous version of the load module from the current production library (PD.CURR.LOADLIB) to the backup production library (PD.BACK.LOADLIB).

2. Copy the load module from the current system test library (ST.CURR.LOADLIB) to the current production library (PD.CURR.LOADLIB).

3. From the *test DB2* system, copy the ST.PARTINQ.VER2 package to PD.PARTINQ.VER2 in the production DB2 system using the following subcommand:

```
BIND PACKAGE(production-DB2-location-name.PD)
    OWNER(application-owner)  QUALIFIER(application-owner)
    COPY(ST.PARTINQ) COPYVER(VER2)......
```

4. Copy the source from the current system test library (ST.CURR.SRCELIB) to the current production library (PD.CURR.SRCELIB) to store the corresponding source of the load module. (This step is optional.)

Note that copying a package must be executed at the test DB2 system, not the production DB2 system, because the package being copied must reside in its local DB2. In this case, the system administrator must be given appropriate privileges in the test environment.

The related DB2 catalog in the production DB2 system is basically the same as that shown in Figure 34 on page 84 except for the REMOTE column, which indicates that the new version of the package was bound from the remote location. (See Figure 37.)

```
 < SYSIBM.SYSPACKAGE >
COLLID|NAME   | .... |VERSION|REMOTE| ......
------ ------- ------ ------- ------ -------
  PD  |PARTINQ| .... |  VER1 |  N   |
  PD  |PARTINQ| .... |  VER2 |  Y   |    -----
```

Figure 37. DB2 Catalog: Scenario 1, Production, Using BIND COPY Option

These procedures using the BIND COPY option eliminate the need to precompile, compile, and link-edit the program. Hence, there is no absolute need for an exclusive source library and DBRM library for each phase. The only library that needs to be distinguished is the load library. However, it is recommended that the corresponding source module be kept in a separate library set up for each environment to avoid loss or confusion of the integrity between the load module and source module.

These procedures are useful for customers who want to save precompile time, as well as those who distribute the same application program to multiple DB2 sites.

4.3.2 Scenario 2: Different Version_id for Each Environment

In Scenario 1, it was possible to use the same version_id throughout the entire promotion process by changing the collection_id between phases. However, there are cases where users do not want to change the collection_id but prefer to keep multiple versions of a program. Some users who have only a small number of packages may want to use the same collection during the test to simplify the testing environment, and some users using programs that explicitly set the CURRENT PACKAGESET register may not want to change the source code to point to the appropriate collection relating to each phase of the promotion process. Using the VERSION keyword allows keeping several versions of a program in the same collection by appending a different version_id to each version of the program.

The basic environment of Scenario 2 is same as that of Scenario 1, with the following exceptions:

- Every PARTINQ package throughout the promotion process is bound to the same collection COL1.
- Version_id includes:
  - Constant value and UT in unit test (for example, VER1UT)
  - Constant value and ST in system test (for example, VER1ST)
  - Constant value and time stamp in production (for example, VER1yymmddhhmm).

Note:

During unit test and system test, users may go through many iterations before completing the test. Using a time stamp as a part of the version_id in these phases may cause unwanted growth of packages in the DB2 catalog and directory. Hence this scenario fixes one version_id per version, per program, and per environment, so that users can replace the previous package as long as they are testing the same version of a program. (For example, the original PARTINQ program in unit test has version_id VER1UT, and the modified program has VER2UT regardless of the number of precompilations.)

In the production phase, where precompilation and binding are usually done on a plan base, using a time stamp as a part of the version_id does not lead to the problems mentioned above. Moreover, using a time stamp ensures that a unique version_id is generated each time the program is prepared and assists in identifying the older versions for freeing. We used the combination of a constant value and a user-generated time stamp rather than VERSION(AUTO) because we wanted a value in the version_id that is passed from the unit test that adds some meaning to the version name.

In addition, using the combination of a constant value and a time stamp assists you when you have to find out whether the package and the load module match. The constant value helps you in searching the version_id embedded in the load module, and the time stamp ensures that the version_id is unique for each precompilation. Therefore, you can easily see whether the load module and the package match just by

checking the version_id in both, instead of searching for a consistency token in the load module with a hex dump.

- The plan is shared between the unit test and system test, and its name is CARDEAL.

  Note: From a security point of view, it is better to have separate plans, but here, we assumed the case where users want to simplify the plan configuration (plan name and package list entries). The plan configuration stays the same regardless of the phase or the DB2 subsystem.

Because the version_id in Scenario 2 is a combination of a constant value throughout the promotion process and a unique value for each phase, three different CLISTs are used to invoke the precompiler. These CLISTs require the user to key in only the constant value to be passed throughout the promotion process regardless of the phase. Each CLIST creates the version_id appropriate for each phase, including the constant value specified as a part of the version_id.

For the unit test, the CLIST in Figure 38 is used. This CLIST precompiles, compiles, and link-edits the program with the version_id as *constant-version-specified.UT*. For example, suppose the name of the CLIST is UTCLIST. By entering the following TSO command:

```
UTCLIST VER2
```

a load module and DBRM with version_id *VER2.UT* are created.

```
PROC 1 VER                    UTCLIST
  %DSNH +
    MACRO(NO)+
    PRECOMP(YES)+
    BIND(NO)+
    COMPILE(YES)+
    LINK(YES)+
    RUN(NO)+
    COPTION('XREF LIB')+
    DBRMLIB('''UT.CURR.DBRMLIB''')+
    DECIMAL(PERIOD)+
    DELIMIT(APOST)+
    HOST(COB2)+
    INPUT('''UT.CURR.SRCELIB(PARTINQ)''')+
    LLIB('''ISP.V3R3M0.ISPLOAD''')+
    LOAD('''UT.CURR.LOADLIB''')+
    OUTNAME(TEMP)+
    PRINT(TEMP) +
    SOURCE(YES)+
    SQLDELIM(APOST)+
    SYSTEM(DB23)+
    VERSION(&STR(&VER..UT))                                        --
END
```

Figure 38. Sample CLIST to Generate Version_id in Unit Test

A similar CLIST named STCLIST is set up for the system test environment to generate the version_id as *constant-version-specified.ST*. (See Figure 39 on page 91.)

```
PROC 1 VER                    STCLIST
  %DSNH +
    MACRO(NO)+
    PRECOMP(YES)+
    BIND(NO)+
    COMPILE(YES)+
    LINK(YES)+
    RUN(NO)+
    COPTION('XREF LIB')+
    DBRMLIB('''ST.CURR.DBRMLIB''')+
    DECIMAL(PERIOD)+
    DELIMIT(APOST)+
    HOST(COB2)+
    INPUT('''ST.CURR.SRCELIB(PARTINQ)''')+
    LLIB('''ISP.V3R3M0.ISPLOAD''')+
    LOAD('''ST.CURR.LOADLIB''')+
    OUTNAME(TEMP)+
    PRINT(TEMP) +
    SOURCE(YES)+
    SQLDELIM(APOST)+
    SYSTEM(DB23)+
    VERSION(&STR(&VER..ST))                                        --
END
```

Figure 39. Sample CLIST to Generate Version_id in System Test

For the production environment, the version_id consists of a constant value and time stamp. To generate this version_id, the sample CLIST shown in Figure 40 on page 92 is used. This CLIST precompiles, compiles, and link-edits the program with the version_id as *constant-version-specified.date-of-precompile.time-of-precompile* . For example, suppose the name of the CLIST is PDCLIST. By entering the following TSO command:

```
PDCLIST VER2
```

a load module and DBRM with version_id *VER2.921201.1530* are created.

```
PROC 1 VER                    PDCLIST
  %DSNH +
    MACRO(NO)+
    PRECOMP(YES)+
    BIND(NO)+
    COMPILE(YES)+
    LINK(YES)+
    RUN(NO)+
    COPTION('XREF LIB')+
    DBRMLIB('''PD.CURR.DBRMLIB''')+
    DECIMAL(PERIOD)+
    DELIMIT(APOST)+
    HOST(COB2)+
    INPUT('''PD.CURR.SRCELIB(PARTINQ)''')+
    LLIB('''ISP.V3R3M0.ISPLOAD''')+
    LOAD('''PD.CURR.LOADLIB''')+
    OUTNAME(TEMP)+
    PRINT(TEMP) +
    SOURCE(YES)+
    SQLDELIM(APOST)+
    SYSTEM(DB2B)+
    VERSION(&STR(&VER..+
    &SUBSTR(7:8,&SYSDATE)&SUBSTR(1:2,&SYSDATE)&SUBSTR(4:5,&SYSDATE).+
    &SUBSTR(1:2,&SYSTIME)&SUBSTR(4:5,&SYSTIME)))
  END
```

Figure 40. Sample CLIST to Generate Version_id with Time Stamp

Figure 41 on page 93 illustrates the current environment.

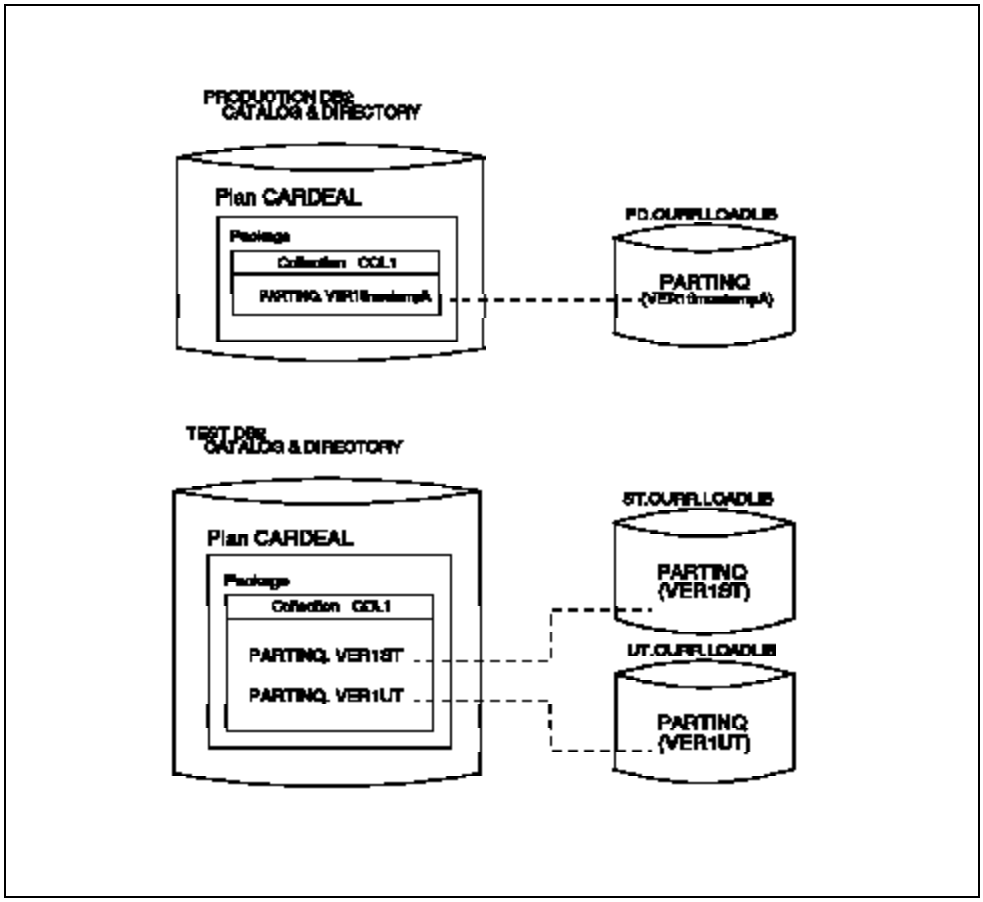

Figure 41. Current Environment for Scenario 2

Scenario 2 also starts with a modification of the PARTINQ program. Figure 42 on page 94 shows the change in version_id through each phase after the new version of the program has been promoted. Figure 43 on page 95 presents an overview of the promotion procedures in Scenario 2. Each phase is described in detail below.

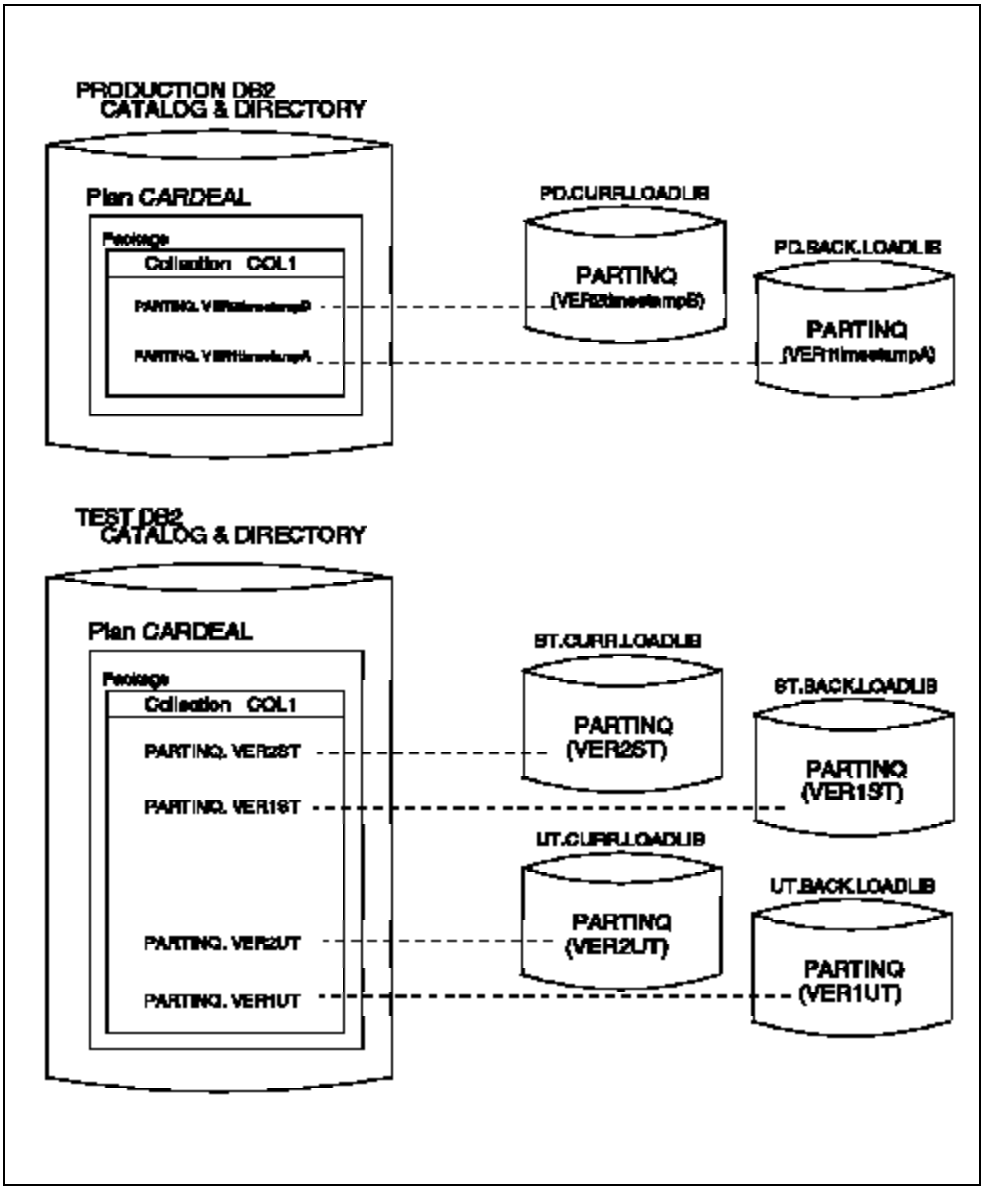

Figure 42. Environment after Modification of Program in Scenario 2

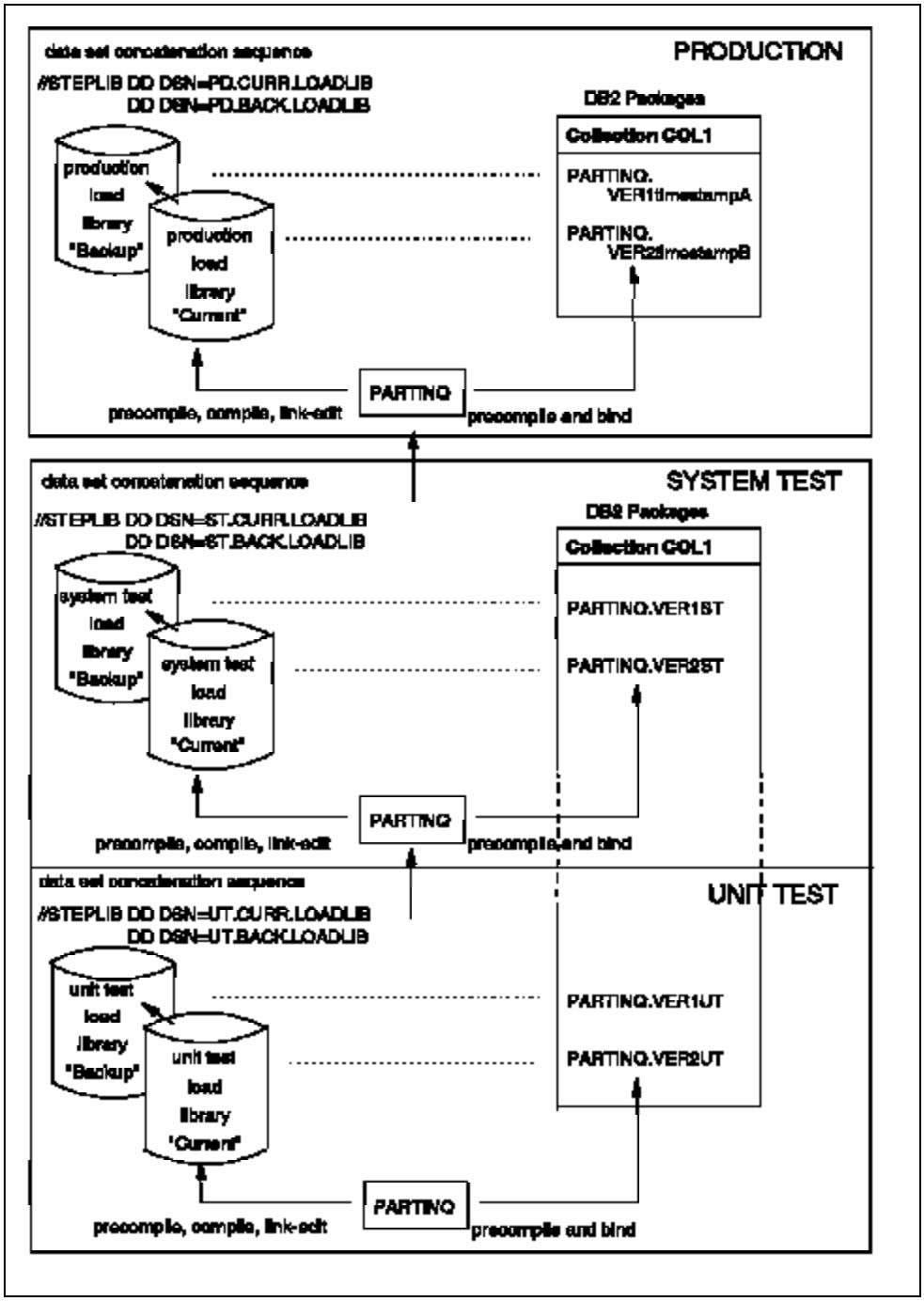

Figure 43. Promotion: Scenario 2

Unit Test

Assumption

In the unit test, the programmer and the unit test administrator play the same roles as those indicated in Section “Unit Test” on page 77. The programmer has the following privileges:

- BINDADD privilege
- CREATE IN privilege for collection COL1
- Implicit privilege as table owner
- EXECUTE privilege on the plan CARDEAL.

The unit test administrator has the following privileges:

- PACKADM authority on collection COL1 with grant option
- BINDADD privilege with grant option
- Implicit privilege as plan owner.

Promotion Procedure

The promotion procedure to the unit test is to:

1. Copy the previous version of the program to a backup library if necessary.
2. Prepare the source code.
3. Precompile, compile, and link-edit the program using the sample CLIST shown in Figure 38 on page 90. Specify VER2 as the constant value. The CLIST generates the version_id as VER2.UT.
4. Bind the package into collection COL1. The bind package subcommand is:

```
BIND PACKAGE(COL1) OWNER(programmer) QUALIFIER(programmer)
    MEMBER(PARTINQ) LIBRARY(UT.CURR.DBRMLIB)
    VALIDATE(BIND) ACTION(ADD) ......
```

The unit test administrator can use the following bind subcommand to bind the plan:

```
BIND PLAN(CARDEAL) PKLIST(COL1.PARTINQ, ....) .......
                or
BIND PLAN(CARDEAL) PKLIST(COL1.*) .......
```

However, the CARDEAL plan containing the package name COL1.PARTINQ already exists, so there is no need to bind the plan again.

System Test

Assumption

In the system test, the tester and the system test administrator play the same roles as those indicated in Section “System Test” on page 80. The tester has the following privilege:

- EXECUTE privilege on the plan CARDEAL.

The system test administrator has the following privileges:

- PACKADM authority on collection COL1 with grant option
- BINDADD privilege with grant option
- EXECUTE privilege on plan CARDEAL with grant option
- Implicit privilege as table owner
- Implicit privilege as package owner.

Promotion Procedure

The promotion procedure for the system test administrator is to:

1. Copy the previous version of the source module, DBRM, and load module from the current system test libraries to the backup system test libraries.

2. Copy the source module from the current unit test library (UT.CURR.SRCELIB) to the current system test library (ST.CURR.SRCELIB).

3. Precompile, compile, and link-edit the program using the sample CLIST shown in Figure 39 on page 91. Specify VER2 as the constant value. The CLIST generates the version_id as VER2.ST.

4. Bind the package using collection COL1:

```
BIND PACKAGE(COL1) OWNER(system-test-administrator)
    QUALIFIER(system-test-administrator)
    MEMBER(PARTINQ) LIBRARY(ST.CURR.DBRMLIB)
    ACTION(ADD) VALIDATE(BIND) ....
```

Figure 44 on page 98 shows the DB2 catalog after the package has been bound.

```
 < SYSIBM.SYSPACKAGE >
COLLID|NAME   |CONTOKEN|OWNER  |CREATOR|QUALIFIER|VERSION| ......
------ ------- -------- ------- ------- --------- ------- ------
 COL1 |PARTINQ|.M ...ï¥|prgrmr1|prgrmr1| prgrmr1 |VER1.UT|    ---- UT
 COL1 |PARTINQ|.M «..åf|prgrmr2|prgrmr2| prgrmr2 |VER2.UT|    ----
 COL1 |PARTINQ|.M ?..7 |stadm  |stadm  | stadm   |VER1.ST|    ---- ST
 COL1 |PARTINQ|.M  .ååû|stadm  |stadm  | stadm   |VER2.ST|    ----


 < SYSIBM.SYSPACKAUTH >
GRANTEE|COLLID|NAME   |BINDAUTH|COPYAUTH|EXECUTEAUTH| ......
------- ------ ------- -------- -------- ----------- -------
prgrmr1| COL1 |PARTINQ|   G    |   G    |    G     |    ---- UT
prgrmr2| COL1 |PARTINQ|   G    |   G    |    G     |    ----
stadm  | COL1 |PARTINQ|   G    |   G    |    G     |    ---- ST
stadm  | COL1 |PARTINQ|   G    |   G    |    G     |    ----


 < SYSIBM.SYSPLANAUTH >
GRANTEE|NAME   |BINDAUTH|EXECUTEAUTH| ......
------- ------- -------- ----------- -------
utadm  |CARDEAL|   G    |    G     |       ----    UT
prgrmr1|CARDEAL|        |    Y     |       ----
prgrmr2|CARDEAL|        |    Y     |       ----
sttster|CARDEAL|        |    Y     |       ----    ST
stadm  |CARDEAL|        |    G     |       ----


 < SYSIBM.SYSRESAUTH >
GRANTEE|QUALIFIER|NAME|USEAUTH| ......
------- --------- ---- ------- -------
utadm  | PACKADM |COL1|   G   |    ---- UT
prgrmr1|         |COL1|   Y   |     ----
prgrmr2|         |COL1|   Y   |     ----
stadm  | PACKADM |COL1|   G   |    -----ST


 < SYSIBM.SYSUSERAUTH >
GRANTEE|BINDADDAUTH| ......
------- ----------- -------
utadm  |    G     |       ----    UT
prgrmr1|    Y     |       ----
prgrmr2|    Y     |       ----
stadm  |    G     |       -----ST
```

Figure 44. DB2 Catalog: Scenario 2, Unit Test, System Test. Prgrmr1 is the programmer for the previous version of the program, VER1.UT, in the unit test; prgrmr2 is the programmer for the modified program, VER2.UT, in the unit test; utadm is the unit test administrator; stadm is the system test administrator; and sttster is the tester in the system test.

The COPY option of the BIND PACKAGE subcommand is not available for this promotion procedure. Copying a package cannot change the version_id, and in this case, changing the version_id is mandatory.

Because the unit test and system test share the same CARDEAL plan, there is no need to bind the plan.

The tester can execute the program after the system test administrator completes the above procedure.

Fallback Procedure

The fallback to the previous version simply requires that the load module in the load library be copied back from ST.BACK.LOADLIB to ST.CURR.LOADLIB or that the new version of the load module be renamed or deleted.

Production

Assumption

In production, the end user, application owner, and system administrator play the same roles as indicated in Section “Production” on page 82.

The end user has the following privilege:

- EXECUTE privilege on the CARDEAL plan.

The application owner has the following privileges:

- EXECUTE privilege on COL1.*
- Implicit privilege as table owner
- Implicit privilege as COL1.PARTINQ package owner
- Implicit privilege as CARDEAL plan owner.

The system administrator [3] has the following privileges:

- SYSCTRL authority.

Promotion Procedure

The promotion procedure is to:

1. Copy the previous version of the source module, DBRM, and load module from the current production libraries to the backup production libraries for fallback.
2. Copy the source module from the current system test library (ST.CURR.SRCELIB) to the current production library (PD.CURR.SRCELIB).
3. Precompile, compile, and link-edit the program using the sample CLIST shown in Figure 40 on page 92. Specify VER2 as the constant value. The CLIST generates the version_id as VER2.yymmdd.hhmmss.

[3] The system administrator is simply an administrator for DB2 resources in this scenario and does not necessarily have SYSADM authorization.

4. Bind the package using collection COL1:

```
BIND PACKAGE(COL1) OWNER(application-owner)
    QUALIFIER(application-owner)
    MEMBER(PARTINQ) LIBRARY(PD.CURR.DBRMLIB)
    ACTION(ADD) VALIDATE(BIND) ....
```

The COPY option of the BIND PACKAGE subcommand is not available for this procedure. Figure 45 illustrates the DB2 catalog after the package has been bound.

```
 < SYSIBM.SYSPACKAGE >
COLLID|NAME   |CONTOKEN|OWNER  |CREATOR|QUALIFIER|VERSION       |....
------ ------- -------- ------- ------- --------- ---------------- ----
 COL1 |PARTINQ|.NT .+ö.|aplownr|systadm| aplownr |VER1.921210.1410|
 COL1 |PARTINQ|.NT[.‰Ñ*|aplownr|systadm| aplownr |VER2.921210.1649|


 < SYSIBM.SYSPACKAUTH >
GRANTEE|COLLID|NAME   |BINDAUTH|COPYAUTH|EXECUTEAUTH| ......
------- ------ ------- -------- -------- ----------- -------
aplownr| COL1 |PARTINQ|   G    |   G    |    G     |
aplownr| COL1 |  *    |        |        |    Y     |
aplownr| COL1 |PARTINQ|   G    |   G    |    G     |


 < SYSIBM.SYSPLANAUTH >
GRANTEE|NAME   |BINDAUTH|EXECUTEAUTH| ......
------- ------- -------- ----------- -------
enduser|CARDEAL|        |    Y     |
aplownr|CARDEAL|   G    |    G     |


 < SYSIBM.SYSUSERAUTH >
GRANTEE|SYSCTRLAUTH| ......
------- ----------- -------
systadm|    Y     |
```

Figure 45. DB2 Catalog: Scenario 2, Production. Enduser is the end user; aplownr is the application owner; and systadm is the system administrator.

It is not necessary to bind the plan in production, assuming that the plan already has been bound for the previous version of the program:

```
BIND PLAN(CARDEAL) PKLIST(COL1.*) ......
```

After the above procedure has completed, the new version of the PARTINQ program is available to end users.

The system administrator should free the old version of the package at an appropriate time.

Fallback Procedure

If fallback is required, simply replace the PARTINQ load module in the PD.CURR.LOADLIB with the old version of the load module backed up in PD.BACK.LOADLIB, or either delete or rename the new version of the load module in PD.CURR.LOADLIB. Free the failed version of the package at an appropriate time.

4.4 Considerations and Recommendations

In comparison to using collections as a means of application versioning, using VERSION provides users with broader and more flexible choices, leaving collection usage for other purposes, such as mirror table application or try-out collection, and practically eliminating the limit on the number of versions. This flexibility brings up several considerations with regard to using VERSION:

- Use of version_id through the promotion process
- Version naming considerations
- Administration considerations.

4.4.1 Use of Version_id through the Promotion Process

We have looked at two application promotion scenarios that use the version_id differently: Scenario 1 uses the same version_id throughout the entire promotion process; Scenario 2 uses different version_ids. Although version_id usage is dependent on each user's environment, here are some points to consider when deciding which approach to take:

- Using the same version_id throughout the entire promotion process has the following advantages:
  - Versioning is easier to maintain because the version_id represents the program version regardless of the phase.
  - The BIND COPY option is available, which eliminates the necessity of precompilation.

  The drawbacks are:
  - In the same location, the use of a different collection for each phase is mandatory.
- Using a different version_id for each promotion phase has the following advantages:
  - Users can decide whether or not to use collections.
  - The plan configuration (plan name and package list entries) can stay the same regardless of the phase.

  The drawbacks are:
  - Precompilation is mandatory for each phase.
  - When using the same collection, security control is less strict.

4.4.2 VERSION Naming Considerations

The version_id should be kept simple and as short as possible (even though up to 64 characters are allowed) because freeing or replacing a package may require the full version_id.

In the production environment in Scenario 2 a time stamp is used as the version_id. Whether VERSION(AUTO) or a user-generated time stamp is used, the inclusion of a time stamp in the version_id means that a new version_id is generated for each precompilation. Without a tool to automate the administration of versions, it is not recommended that a time stamp be included in the version_id in an environment where unpredictable iterations of precompilation may occur, such as the testing environment. However, in an environment where the occurrence of versions can be controlled, such as the production environment, or test environment where someone cleans up the old versions on a regular basis, including a time stamp in the version_id is very useful, because a time stamp ensures the correspondence between the version name and the consistency token.

Using VERSION(AUTO) may be a reasonable choice when application migration tools are available.

4.4.3 Administration Considerations

Before implementing VERSION, it is important to establish version administration rules.

First and foremost, the number of versions stored in DB2 must be controlled. Figure 46 on page 103 shows some ways to limit the number of versions. Example 1 shows use of the same version_id for the same version of a program in each phase. The package can be easily replaced by BIND COPY ACTION(REPLACE). All phases in Scenario 1 and the unit and system test phases in Scenario 2 use this approach. This may be effective in the testing environment when you do not want a new version_id generated for each bind package unless there is a major modification in a program (however, eventually, the old version has to be freed).

Example 2 shows the approach of replacing the old version whenever a new one is created by specifying the old version_id in the REPLVER keyword when binding a package. This approach may be effective if you want to keep every package for every precompilation with a different version_id.

Example 3 shows the approach of cycling the version_id. This approach works better if upon precompilation cyclic version_ids can be generated automatically by means of some tool.

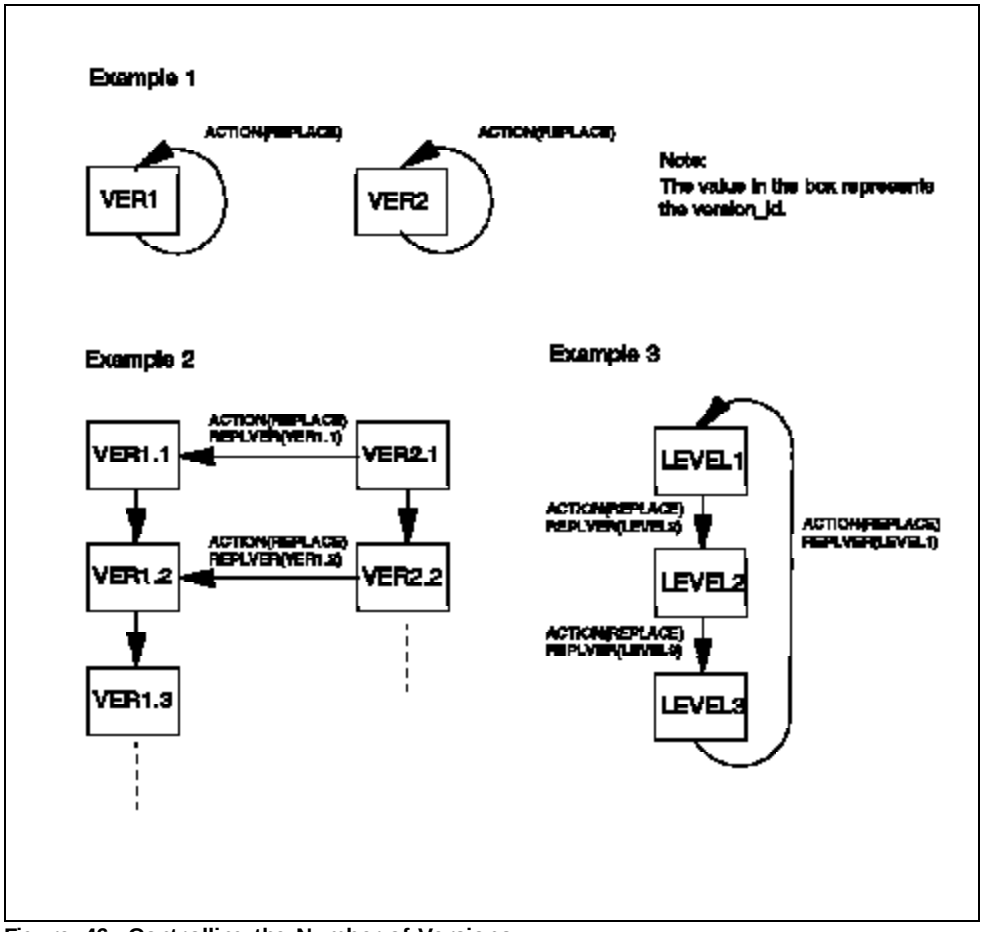

Figure 46. Controlling the Number of Versions

Another point users should be aware of in administering VERSION is that package privileges do not have a version_id as a qualifier. The DB2 catalog table SYSIBM.SYSPACKAUTH where package privileges are recorded does not have a version_id column.

Take a look at Figure 47, for example, which is an extract of SYSIBM.SYSPACKAUTH shown in Figure 32 on page 79 and Figure 33 on page 81.

```
 < SYSIBM.SYSPACKAUTH >
GRANTEE|COLLID|NAME   |BINDAUTH|COPYAUTH|EXECUTEAUTH| ......
------- ------ ------- -------- -------- ----------- -------
prgrmr1|  UT  |PARTINQ|   G    |   G    |     G     |
prgrmr2|  UT  |PARTINQ|   G    |   G    |     G     |
stadm  |  ST  |PARTINQ|   G    |   G    |     G     |
stadm  |  ST  |PARTINQ|   G    |   G    |     G     |
```

Figure 47. DB2 Catalog: SYSIBM.SYSPACKAUTH

Recall that prgrmr1 and stadm were the owners of the old version of the package, VER1, and prgrmr2 and stadm were the owners of the modified version of the package, VER2. Because there is no version concept in the package privilege, prgrmr1 and prgrmr2 have all the privileges for COL1.PARTINQ, regardless of the version_id. And, even if the same owner was specified when binding different versions of a package, a row for each version is inserted in this table. (You can see two rows for stadm in Figure 47 on page 103.)

It is also important to note that the entries are not deleted until all versions of the package are freed. When the following subcommand has been executed:

```
FREE PACKAGE(UT.PARTINQ.(*))
```

or

```
FREE PACKAGE(UT.PARTINQ.(VER1), UT.PARTINQ.(VER2))
```

prgrmr1 and prgrmr2 are deleted from SYSIBM.SYSPACKAUTH. In other words, even if VER1 of the UT.PARTINQ package is freed by the following subcommand:

```
FREE PACKAGE(UT.PARTINQ.(VER1))
```

prgrmr1 is not deleted from SYSIBM.SYSPACKAUTH and still has all of the package privileges on UT.PARTINQ.

This mechanism may lead to unwanted growth of SYSIBM.SYSPACKAUTH because an entry is always added to this table even if the owner is the same as the previous version of the package. One of the simple ways to prevent growth of SYSIBM.SYSPACKAUTH is to FREE all versions of a package occasionally. This may be acceptable in a test environment; however, it may not be realistic in a production environment because you have to bind the necessary packages again after freeing. Another way to prevent this growth is to specify the same owner when binding different versions of a package and use the REPLVER option to replace the old version of the package. Using REPLVER replaces the entry in SYSIBM.SYSPACKAUTH, so it is better than binding a new version with ACTION(ADD) and freeing the old version afterwards. (See Figure 48 on page 105.)

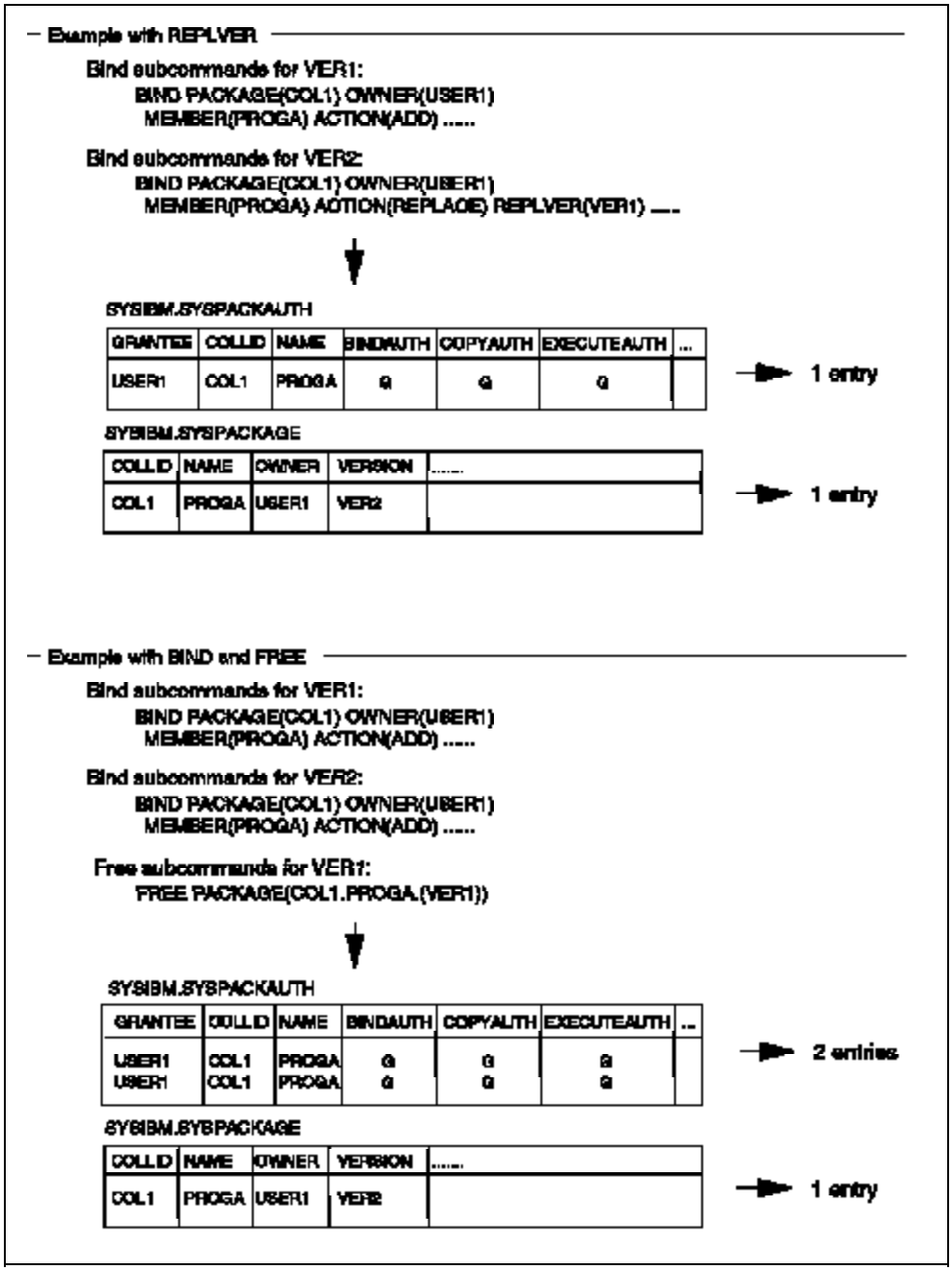

Figure 48. Occurrence of Entries in SYSIBM.SYSPACKAUTH in DB2 V2.3

In DB2 V3, only one entry is recorded in SYSIBM.SYSPACKAUTH if the owner stays the same for multiple versions of a package.

Chapter 5. Distributed DB2 Packages

DB2 provides support for packages in a distributed database environment.

In this chapter we examine the perspective where DB2 is the application server and discuss the use of DB2 packages in the following distributed environments:

- DB2 application accessing a remote DB2
- DB2/2* application accessing DB2
- SQL/DS application accessing DB2
- AS/400* application accessing DB2.

DB2/6000* is not covered because it had only been recently announced at the time of writing this book.

The objective of this chapter is to provide a DB2 trained reader with information on how a DB2 package is created and used in a distributed database environment. The DB2 administrator supporting such an environment might need to consider other platforms that connect to DB2, such as SQL/DS, AS/400, and DB2/2.

This chapter does not cover aspects of distributed programming or application code portability.

If you are developing distributed applications, you should refer to other manuals regarding the implementation of DRDA applications. The documents listed in “International Technical Support Center Publications” on page xix cover application design, program coding and development, security, recovery, and performance considerations in distributed application development.

5.1 DRDA and Distributed Relational Databases

DRDA allows IBM and non-IBM relational databases to access data at unlike relational database systems. Figure 49 shows the possible connectivity. The arrows indicate the flow of user data. A double-arrow line indicates that the database system can act as both a requester and a server. Note that the DB2/2 and DB2/6000 database managers can act only as a requester at this time. DRDA supports connections between like database systems, for example, DB2 and DB2, even though the like database systems usually have their own private protocol for communications.

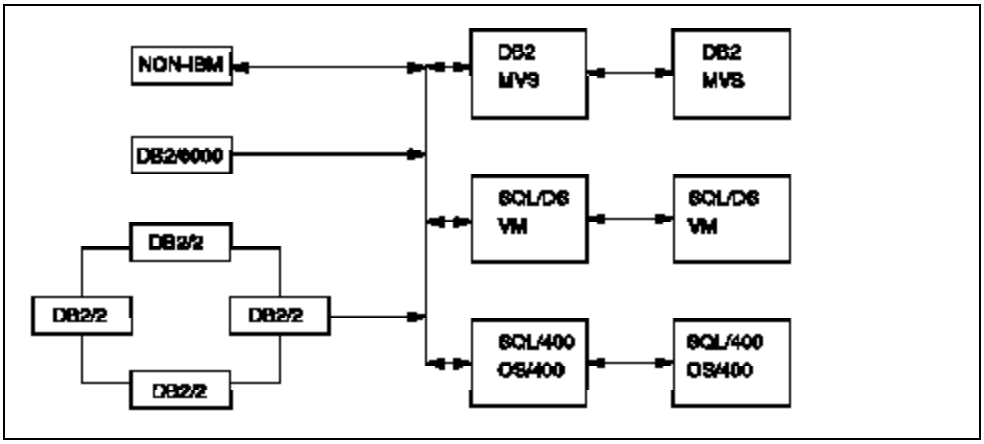

Figure 49. DRDA Connectivity

DRDA supports SQL as the application program interface (API) to access remote databases. The SQL statements to be executed at the application server site are bound and stored at that site in the form of an SQL package.

DRDA has three levels (see Table 8). Each level of DRDA restricts what can be performed within a unit of work (UOW).

Table 8. DRDA Levels

Level	SQL Stmt per UOW	DBMS per UOW	DBMS per SQL Stmt
Remote unit of work	N	1	1
Distributed unit of work	N	N	1
Distributed request	N	N	N

Remote Unit of Work (RUW): This level supports, within a logical unit of work, application access to just one remote database system. Before a program can access another database, it must end the current unit of work before starting another one. This is done by issuing either a COMMIT or ROLLBACK statement. RUW is architected in DRDA-1.

Distributed Unit of Work (DUW): This level extends RUW distributed database capability by allowing, within a single logical unit of work, 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 unit of work and starting another one. However, a single SQL statement cannot access more than one database. DUW is architected in DRDA-2 and DB2 private protocols.

Distributed Request (DR): This is the most flexible and powerful of the DRDA levels. On top of RUW and DUW capability, it allows a single SQL statement to access multiple databases. DR is only conceptually described today.

5.2 DB2 to DB2

DB2 supports two distributed database connection protocols:

- DB2 to DB2 private protocol
- DRDA.

DB2 to DB2 Private Protocol: DB2 V2.2 introduced the DB2 private protocol. This protocol support, within a single unit of work, allows application access to the local DB2 database as well as multiple remote DB2 database systems. Non-DB2 databases are not supported. DB2 V2.3 implements limited DUW capabilities where applications can access multiple locations. TSO applications can update at a single location (local or remote), and IMS and CICS applications can update only the local database. DB2 V3 implements DUW and multisite update capabilities where TSO, IMS, and CICS applications can access and update multiple remote databases. References to remote tables are implemented by means of three-part table or view names (location.qualifier.tablename) defined in DB2. Thus remote connections are system-directed. The DB2 private protocol supports only the dynamic execution of SQL DML statements. DB2 packages are not used.

DRDA: DB2 V2.3 implements RUW, where applications can connect to one remote site only. TSO applications can update at a single site (local or remote), and IMS and CICS applications can update the local database only. DB2 V3 implements DUW and multisite update capabilities where TSO, IMS, and CICS applications can update multiple database locations. With DRDA, connections to remote databases are application-directed using CONNECT statements rather than the three-part names. Remote SQL is bound at the remote site before execution, and hence static SQL is supported.

5.2.1 Remote DB2 Access with DRDA

Figure 50 on page 110 illustrates a program in DB2 V2.3 that accesses a remote DB2 in New York and the local DB2 in Boston. The location names are NEWYORK and BOSTON, respectively. The Boston site where the program executes is the requester, and the New York site is the server.

To access data at the New York server site, the program establishes an SQL connection using the CONNECT TO <location> statement. The CONNECT statement sets the CURRENT SERVER register to the location name for New York, which is NEWYORK. This register value is eventually used to resolve the location_name of the package.

The SQL to be executed must be bound into a package at the server and requester beforehand. To reference the remote package at the server, the DB2 requester passes the package information to the DB2 server. This information includes the location_name, the collection_id, the package_ id, and the consistency token. Figure 50 shows how these are derived. Note that the collection-id could be derived from either the SET CURRENT PACKAGESET statement in the program (if any) or the PKLIST entries in the plan. At the server, DB2 uses a default plan called DISTSERV for serving this remote request (the user need not create this plan). Also, the DB2 server uses an index to locate the right package in the DB2 Skeleton Package Tablespace (SPT01).

To connect back to the local DB2, the program can issue a CONNECT RESET statement. This resets the CURRENT SERVER register to the local location name (BOSTON), which is subsequently used for the location_name of the package. The local package is allocated at the next SQL statement.

Note: In DB2 V2.3, a COMMIT is required to end the remote unit of work before switching databases.

At the DB2 requester, a plan is created that includes in the package list the local and remote package. The location ID of the package denotes where the package resides.

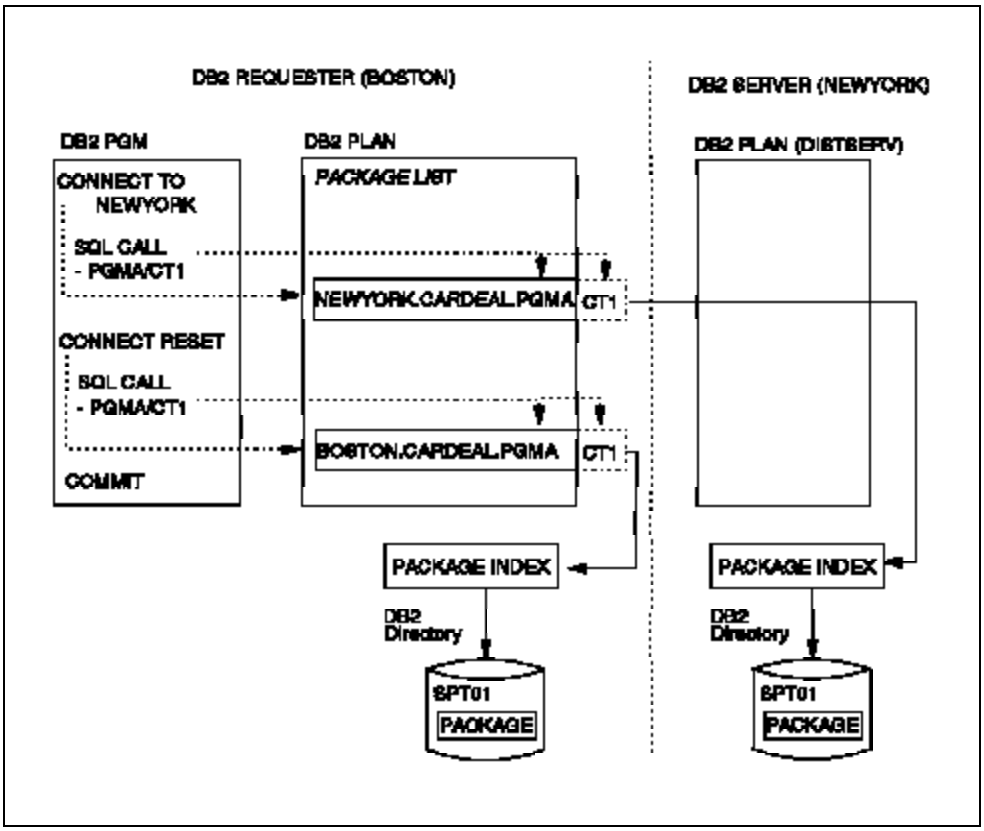

Figure 50. DB2 Distributed Program and Package Resolution

5.2.2 DB2 Remote Bind

DB2 remote bind has the following characteristics:

- Remote bind applies to packages only.
- Remote bind of a package results in a package stored at the remote site.
- The program references a remote package by means of a local plan. The plan includes the required remote package. A remote package is one that has a remote location_name in the package name.

- A default plan, DISTSERV, is automatically allocated by the DB2 server to service the remote request during execution time.
- Authorizations have to be granted at the remote site.

During a remote bind, DB2 performs authorization checking, reads and updates the remote catalog, and creates the package at the remote site. Catalog reads, catalog updates, and authorization checks are not performed at the local site. If the COPY option is used, DB2 validates that the necessary authority exists at the local site.

Figure 51 on page 112 shows the bind process for the program that accesses the local DB2 (BOSTON) and the remote DB2 (NEWYORK). The binds are issued at the requester in Boston and results in a package created at the local and remote DB2s.

To reference the packages, a DB2 plan is created at the requester.

The bind for the local package would be:

```
BIND PACKAGE (BOSTON.CARDEAL) ...
```

The bind for the remote package would be:

```
BIND PACKAGE (NEWYORK.CARDEAL) ...
```

Assuming that only one collection is used at each site, the bind plan would look like this:

```
BIND PLAN(CAR) PKLIST(NEWYORK.CARDEAL.*, BOSTON.CARDEAL.*)
    VALIDATE(BIND)...
```

When binding a plan that refers to a remote package, DB2 does not validate whether the binder has the execute privilege for the remote package. Remote package execute privileges are checked at execution time regardless of whether VALIDATE(BIND) or VALIDATE(RUN) is used. Hence, even though the execute privileges for the remote packages do not exist at bind plan time at the server, the bind still goes through without warning or error messages. The user must ensure that the privileges exist at execution time.

Local packages, however, are validated at bind time if VALIDATE(BIND) is used. In the above bind plan example, Boston is the requester site, and DB2 performs validation at bind time for the packages in the BOSTON.CARDEAL collection.

An alternative to coding the plan is to use a wild card (*) for the location_name. For example:

```
BIND PLAN(CAR) PKLIST(*.CARDEAL.*) ...
```

For an application that requires access to many locations, such a plan need not be rebound every time a new location is added. An example of such an application may be Query Management Facility (QMF*). Two drawbacks to using a wild card for the location name are:

- If one of the locations accessed is the local DB2, the validation cannot be performed at bind time for the local packages because of the wild card location_name.
- There may be a security exposure because the plan can be used to access all locations.

During a remote bind, no messages are produced at the remote site. DRDA does not architect a flow for messages. What flows from the application server to the application requester is an SQLCA. The requester may then choose to map the SQLCA to messages that are meaningful to the user.

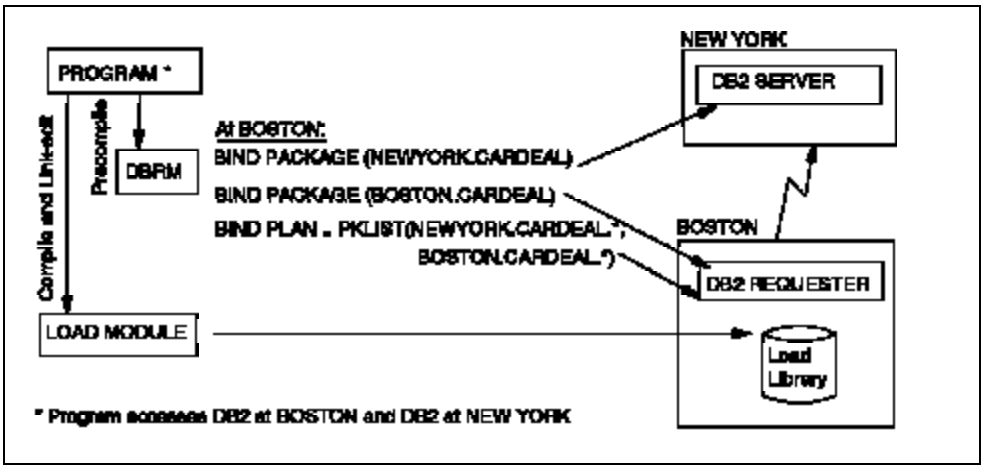

Figure 51. Bind for Packages and Plan for Distributed DB2 Program

No information is stored in the local DB2 catalog for a remotely bound package. There is, however, a reference in SYSIBM.SYSPACKLIST indicating that the remote package is inside a package list for a plan. Figure 52 is an extract from SYSIBM.SYSPACKLIST showing a plan (CAR) at the requester site that references a remote package at the server (DB2REGNDIST01). This is the result of the bind command:

```
BIND PLAN(CAR) PKLIST(DB2REGNDIST01.CARDEAL.*) ...
```

```
 SELECT * FROM SYSIBM.SYSPACKLIST WHERE PLANNAME = 'CAR';
---------+---------+---------+---------+---------+---------+----
PLANNAME   SEQNO  LOCATION        COLLID            NAME
---------+---------+---------+---------+---------+---------+----
CAR          1  DB2REGNDIST01    CARDEAL           *

----+---------+---------+---------+--
 TIMESTAMP               IBMREQD
----+---------+---------+---------+--
 1992-11-18-18.17.30.903965  N
```

Figure 52. Extract from SYSIBM.SYSPACKLIST at Requester Site

Figure 53 on page 113 shows the extract from the SYSIBM.SYSPACKAGE at the DB2 server where a package was remotely bound. Note the following:

- The SYSPACKAGE column for LOCATION is blank and is reserved for future use.
- There is a Y in the remote column indicating that the package was remotely bound. The PDSNAME column, which normally is the pdsname for the DBRM records, contains:
  - A three-character abbreviation that represents the DBMS (DSN for DB2, ARI for SQL/DS, SQL for OS/2, QSQ for OS/400*)
  - The DBMS's version (*vvrrm* or 02030 for DB2 V2.3)
  - Location_name for the remote site (DB2CENTDIST).

The fact that a package is remotely bound does not limit the use of the package to distributed or remote access only. The package created using a remote bind is similar to having the DBRM ported to the remote site and bound there. The package can be accessed by an application that executes either locally at the server site or remotely at the requester site, as long as the load module (with the same consistency token) is ported to the execution site.

```
  SELECT * FROM SYSIBM.SYSPACKAGE  WHERE NAME = 'PARTINQ';
---------+---------+---------+---------+---------+---------+-----
LOCATION        COLLID            NAME     CONTOKEN  OWNER
---------+---------+---------+---------+---------+---------+-----
              CARDEAL           PARTINQ   .L=...'   STDB2C

--+---------+---------+---------+---------+---------+---------+---
 CREATOR   TIMESTAMP                BINDTIME
--+---------+---------+---------+---------+---------+---------+---
 STDB2C   1992-11-18-17.50.57.095646  1992-11-18-17.50.57.095646

------+---------+---------+---------+---------+---------+---------+
 QUALIFIER      PKSIZE     AVGSIZE  SYSENTRIES  VALID  OPERATIVE
------+---------+---------+---------+---------+---------+---------+
 STDB2B          1096          0         0  Y     Y

---------+---------+---------+---------+---------+---------+---
 VALIDATE  ISOLATION  RELEASE  EXPLAIN  QUOTE  COMMA  HOSTLANG
---------+---------+---------+---------+---------+---------+---
 R        S        C       N       N     N

------+---------+---------+---------+---------+---------+-------
 CHARSET  MIXED  DEC31  DEFERPREP  SQLERROR  REMOTE  PCTIMESTAMP
------+---------+---------+---------+---------+---------+-------
 A       N     N     B         N        Y      1992-11-17-1

------+---------+---+---------+---------+
 IBMREQD  VERSION   PDSNAME
------+---------+---+---------+---------+
 N               DSN02030.DB2CENTDIST
```

Figure 53. Extract from SYSIBM.SYSPACKAGE at Server Site

5.2.3 Authorizations for DB2 Requesters

The package owner_id is the authid specified in the bind package option, OWNER. If OWNER is not specified, the package owner_id defaults to the primary authid of the binder. When a bind command is issued from a requester to bind a package at the server, the following privileges have to be granted to the **package owner_id** at the application server site:

- Privileges to execute the package's static SQL statements
- BINDADD
- CREATE IN COLLECTION <collection_id>.

When binding a remote package using the OWNER option, DB2 checks the following at the application requester site where the bind is issued:

- If the binder has the SYSADM privilege, the owner_id can be any authid.
- If the binder has the BINDAGENT privilege, the owner_id must be the authid of the person who granted the BINDAGENT privilege to the binder.
- If the binder does not have the SYSADM or BINDAGENT privilege, the owner_id must be either the primary or secondary authids of the binder.

When binding a plan that includes remote packages, the following privilege must be granted to the **plan owner_id** at the application server site:

```
EXECUTE ON PACKAGE <remote package_ids>.
```

DB2 validates this privilege only at execution time regardless of the bind option VALIDATE. Note that the **user** invoking the application requires the EXECUTE privilege on the application plan at the application requester site.

Authids may be subject to outbound or inbound translation specified in the SYSIBM.SYSUSERNAMES table at both the requester and server end. If translation is used, the authorization grants at the server site must be for the translated authids. Translation occurs for:

- Binder or bindagent's primary authid
- Id specified in the OWNER bind option.

Translation does not apply to the QUALIFIER option.

During the remote execution of a package, the authid of the plan owner is sent to the server. The authid is used to validate at run time whether the plan owner has execute authority on the requested package. The plan owner_id can also be subject to translation. Hence, EXECUTE ON authority must be granted to the translated plan owner_id.

At the server, if the CDB tables indicate that inbound translation is required, the authid is translated to the new authid specified in the SYSIBM.SYSUSERNAMES table. Figure 54 on page 115 shows the extract from SYSIBM.SYSUSERNAMES at the DB2 server. The incoming authid STDB2C will be translated to REMUSER, and correspondingly the authorities should be granted to REMUSER.

```
  SELECT * FROM SYSIBM.SYSUSERNAMES;
---------+---------+---------+---------+--------
TYPE  AUTHID    LUNAME    NEWAUTHID  PASSWORD
---------+---------+---------+---------+--------
I             LUSQLDB2  DEALER01
I             LUS40L02  DEALER02
I             LUOS2S03  DEALER03
I     STDB2C    LUDB23    REMUSER
```

Figure 54. Extract from SYSIBM.SYSUSERNAMES

After translation is done according to the information in the CDB tables, the new authid may go through further translation if the DB2 authorization exits (for example, DSN3@SGN or DSN3@ATH) are executed.

For a discussion of inbound and outbound authid translation, please refer to Chapter 5, “Security and Auditing,” in the *DB2 Administration Guide*.

5.2.4 DB2 Bind Options

This section discusses the bind options relevant to binding a package for a distributed application. For a complete list of all bind options for binding a remote package, please refer to Appendix H, “Bind Options For Remote Packages,” in the *DB2 Application Programming and SQL Guide*.

SQLERROR: The SQLERROR bind option determines whether a package is created when “errors” are encountered during bind time. The “errors” could be due to:

- Unrecognized SQL statements
- Missing objects
- Missing authorizations.

A single distributed application program may reference unlike database servers. Since all database systems do not support the same set of SQL statements, it is possible for the application program to contain SQL statements that are not recognized at all servers. The binding of such programs will result in errors. However, the errors are perfectly acceptable, given the unlike environment in which the distributed application executes. Tables and authorizations at the server may not be in place at bind time, or not all of the tables referenced in the SQL are at the server; they could be at other servers. A package should nevertheless be created despite all of these difficulties.

Figure 55 on page 117 shows how the SQL precompiler option and SQLERROR bind option can be used to resolve syntax problems. It also shows how the VALIDATE and SQLERROR bind options can be used to resolve missing authorizations and objects.

At precompilation time, the SQL precompiler option should be used to distinguish the type of SQL to be expected in the program. If the program is strictly DB2 (local or remote), SQL(DB2) should be used. If the program accesses non-DB2 databases like SQL/DS, SQL(ALL) should be used. The

SQL(ALL) option tells the precompiler to interpret SQL statements for use by non-DB2 databases. Hence, non-DB2 SQL will be tolerated. Without this option, the precompiler defaults to SQL(DB2), and any non-DB2 statements will not result in package creation.

After precompilation, the BIND of the package should include the SQLERROR option. The SQLERROR(NOPACKAGE) option results in no package being created if errors occur during the BIND process. If the package contains non-DB2 SQL statements, SQLERROR(CONTINUE) should be used so that the package can still be created at the DB2 server even when it has non-DB2 SQL statements.

The SQLERROR(CONTINUE) option informs the relevant server to ignore SQL statements that it does not syntactically recognize. If such statements are found, they would be marked nonexecutable, but the package will be bound successfully. For DB2 servers, the BINDERROR column in SYSTEM.SYSPACKSTMT identifies the nonexecutable SQL statements in a particular package.

SQLERROR(NOPACKAGE) is the reverse, and no package is bound when there are invalid statements.

Note: When coding distributed applications, programmers are advised to use SAA* SQL if different types of databases are involved. SAA SQL facilitates portability of applications and data across the different platforms. The current level of SAA SQL is level 2, which encompasses most of the ISO/ANSI SQL86 and SQL89 standards and is a subset of the ISO/ANSI SQL2 standard. SAA SQL level 2 extends support for distributed database access. For more details, refer to *SAA CPI: Database Level 2 Reference*. The precompiler option SQLFLAG(SAA) can be used to enforce compliance for level 2 of SAA SQL level 2. Refer to the *DB2 Application Programming and SQL Guide* for details.

To handle missing authorizations or objects, use VALIDATE(RUN). The server will do an incremental bind at run time to verify the authorizations and existence of previously missing objects. If VALIDATE(BIND) is used, the SQLERROR option determines whether the package can be created if there are missing authorizations and objects. With VALIDATE(BIND) and SQLERROR(CONTINUE), the package can still be created.

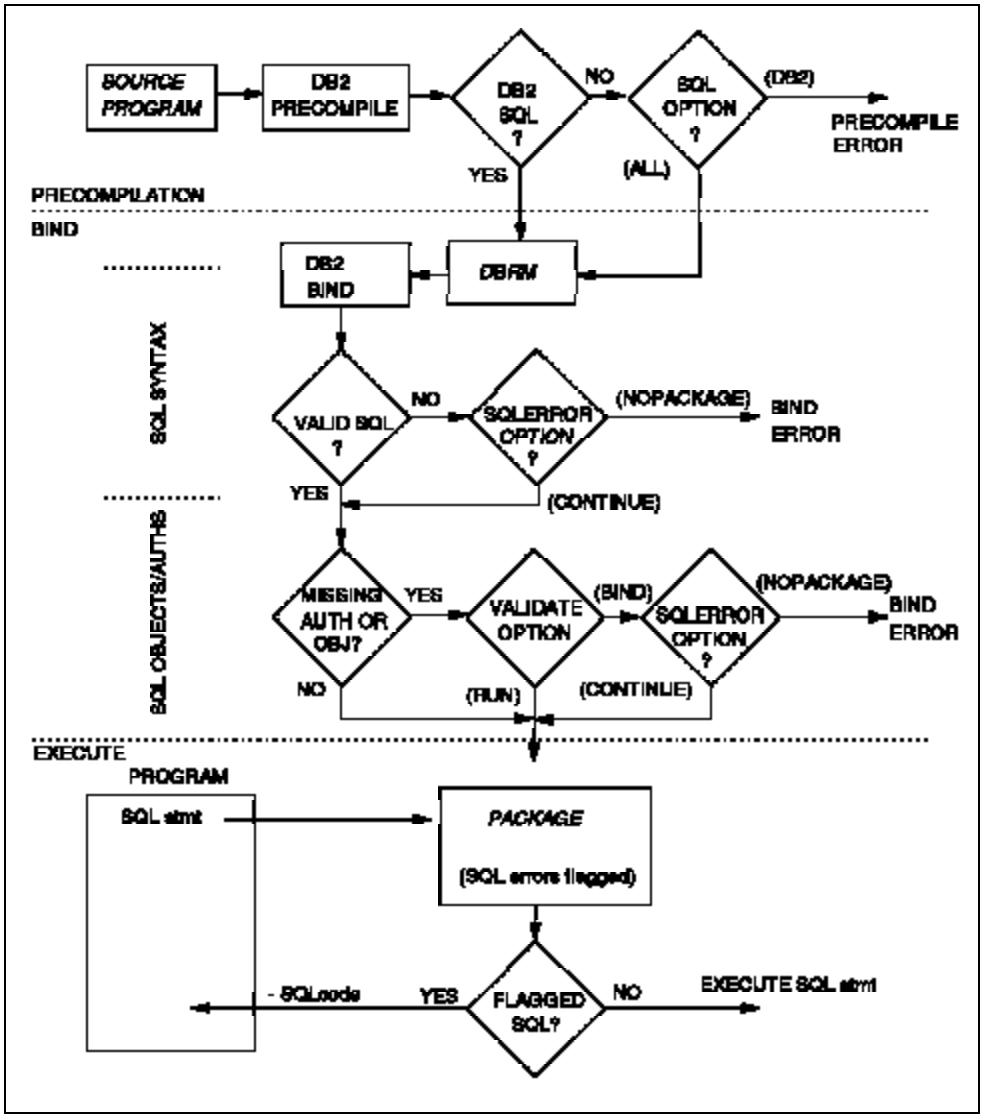

Figure 55. DB2 Remote Precompile and Bind Option

Figure 56 on page 118 shows the possible combinations of the VALIDATE and SQLERROR options to handle different cases in a distributed environment.

CONDITIONS AT SERVER	Bind Options Specified			
	VALIDATE		SQLERROR	
	RUN	BIND	CONT	NOPACK
1. SQL valid All tables and authorizations present		Y		Y
2. SQL valid Some (valid) tables and authorizations missing	Y			Y
3. SQL valid (any SQL associated with missing table or authorization is not executed at this server)		Y	Y	
4. Some SQL statements are invalid at this server		Y	Y	

Figure 56. Choice of VALIDATE and SQLERROR at DB2 Server

The first condition applies to a program accessing DB2-only servers. The program uses DB2 SQL statements only and hence should be valid (unless it is a genuine error). At each server, all referenced tables and required authorizations exist. In this case VALIDATE(BIND) and SQLERROR(NOPACKAGE) can be used.

The second condition also applies to a program accessing DB2-only servers. However, not all of the referenced tables and authorities exist at bind time. The VALIDATE(BIND) option allows package creation even though the objects are missing. Because the program accesses DB2-only servers, SQLERROR(NOPACKAGE) should be specified. The statements that reference the missing objects or authorizations are executable, and DB2 performs an incremental bind during run time. If the objects or authorizations still do not exist at run time, an SQL error code is returned to the program.

The third condition applies to programs accessing DB2-only servers when the following criteria exist:

- All *referenced* objects and authorizations at the server exist at bind time.
- Not *all* objects and authorizations are defined at *every* server. In other words, tables declared in the program exist only at some servers, not all of them. These tables are valid but are missing from some servers.
- Program will not execute invalid statements containing valid but missing tables.

This is the case when the tables are distributed across servers. In other words, the tables are NOT replicated at all of the servers. If the tables that are accessed at execution time exist at bind time, VALIDATE(BIND) should be used. To bypass the error messages due to unresolved tables (that exist at other servers and are referenced by the program), use SQLERROR(CONTINUE).

The fourth condition applies to programs accessing DB2 and non-DB2 servers. In this case, there may be some SQL that is syntactically unrecognizable by DB2. Hence, SQLERROR(CONTINUE) and VALIDATE(BIND) should be used if the tables and authorizations already exist at bind time. Alternatively, use VALIDATE(RUN) if the tables and authorizations do not exist at bind time but will exist at run time.

Caution: VALIDATE(RUN) and SQLERROR(CONTINUE) are useful when binding against unlike database systems as discussed above. However, it is the user's responsibility to distinguish between expected warning messages and messages that are due to real errors.

CURRENTDATA: The CURRENTDATA option applies to distributed applications in which ambiguous cursors are defined. DB2 recognizes each of the following conditions as an ambiguous cursor:

- There are dynamic SQL statements.
- The cursor SELECT declaration does not specify the FOR FETCH ONLY clause.
- The cursor SELECT declaration does not specify UNION, DISTINCT, ORDER BY, GROUP BY operation.
- The cursor SELECT declaration does not specify a join or subquery.
- The cursor SELECT declaration does not specify FOR UPDATE clause.
- The cursor does not have an UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF clause in the SQL statement associated with it.

Ambiguous cursors bound with cursor stability (CS) are potentially updatable in a distributed environment with block fetch operations. Rows of data are returned to the application requester before the FETCH SQL statements in the application are processed. The locks on the data may have been released before the application receives the data. In this case, it is possible for the application to fetch a row of values that no longer exists or has just been recently inserted.

By specifying CURRENTDATA(YES), data currency is ensured, and block fetch is inhibited for ambiguous cursors.

Note: If an ambiguous cursor is actually updated by means of a *dynamic* SQL DELETE or UPDATE WHERE CURRENT OF statement, that update operation will fail.

Disabling block fetch may significantly impact the performance of ambiguous cursors, which are actually intended to be read only. Therefore, to avoid performance issues, the user should code FOR FETCH ONLY in the cursors when data currency is not required.

The default is CURRENTDATA(NO).

COPY: The COPY option can be used in a remote bind. The copied module must reside at the local DB2 where the BIND is issued. The BIND COPY option cannot have a remote package as input for the remote bind operation. The COPY privileges must exist at the local site, not the remote site.

CURRENTSERVER: THE CURRENTSERVER option can be used when binding a plan. If specified, it initializes the CURRENT SERVER register to whatever was specified in the option at execution time. When the program executes, it connects immediately to the server specified. All SQL activities are executed at the server. The SQL CONNECT command is not required until the program needs to access the next server or the local database. This option allows the remote location to be determined at bind time.

Note that with DB2 V3, the usage of CURRENTSERVER means the connect that is done on behalf of the application is treated as a Type 1 CONNECT, which means that the whole application must adhere to the rules associates with a Type 1 CONNECT.

ISOLATION: For packages that execute at the remote site, the default is repeatable read (RR), whereas for local packages that execute at the local site, the default is the same as that of the plan to which they are appended at execution time.

RELEASE: For packages at the server site, the default is COMMIT, whereas the default for local packages is the same as that of the plan to which it is appended at execution time.

There is no ACQUIRE option for packages. It is always ACQUIRE(USE).

EXPLAIN: EXPLAIN is supported for a remote bind at the server. The EXPLAIN output table is stored at the server.

ENABLE/DISABLE: This option is not supported by the DRDA BIND flows. To use it, a rebind issued at the server site is necessary.

The REMOTE option can be used to restrict the package to only certain requesters. The REMOTE option requires the location name for DB2 requesters and lunames for non-DB2 requesters.

If a remote BIND PACKAGE with REPLACE attempts to replace a package that has the ENABLE or DISABLE parameters associated with it, the bind will be unsuccessful, and an error message will be returned.

DRDA-2 Precompiler and Bind Plan Options: DB2 Version 3 introduces new precompiler and bind options to support DRDA-2's ability to connect to several remote databases within a logical unit of work.

In a DRDA-1 implementation, a program can only connect (using SQL CONNECT) to one remote database within a logical unit of work. At any time, the application can have only one SQL connection; any subsequent successful connection will destroy all previous connections (if any). This CONNECT statement is called a type 1 CONNECT. DB2 V2.3 supports only type 1 CONNECTs.

In a DRDA-2 implementation, the application can CONNECT to several remote databases within a logical unit of work. Each subsequent CONNECT statement in the logical unit of work will establish a new connection without

destroying previous connections (if any). This CONNECT statement is called a type 2 CONNECT. DB2 V3 supports both type 1 and type 2 CONNECT statements.

The syntax of the two CONNECT statements is the same. However, the semantics are different and are determined by the CONNECT precompiler options: CONNECT(1) to indicate type 1 and CONNECT(2) to indicate type 2. In DB2 V3, the default is CONNECT(2). The implicit CONNECT associated with the CURRENTSERVER bind option is a type 1 CONNECT.

DB2 V3 also introduces the SET CONNECTION statement, which allows the application to reuse an existing SQL connection, as opposed to CONNECT, which establishes a new connection. This distinction, provided for compatibility with the ISO/ANSI SQL2 standard, is unnecessary in DB2, and hence the SQLRULES bind (plan) option can be used to determine whether a type 1 CONNECT is executed in accordance with the rules of DB2 or the ISO/ANSI SQL2 standard. SQLRULES(DB2) specifies that, if a CONNECT statement identifies with an existing SQL connection, no error will occur; the existing connection will be reused. SQLRULES(STD) specifies that if a CONNECT statement identifies an existing connection, an error will occur. To reuse the connection, the application program must use the SET CONNECTION statement.

The DISCONNECT bind (plan) option specifies which remote connections are destroyed during commit operations. DISCONNECT(EXPLICIT) specifies that only connections in the *released* state are destroyed. A connection can be released using the SQL RELEASE statement. DISCONNECT(AUTOMATIC) specifies that all remote connections are destroyed at commit. DISCONNECT(CONDITIONAL) specifies that remote connections are destroyed provided an open WITH HOLD cursor is not associated with the connection.

5.2.5 Package List and Performance

The search for a package in a distributed environment is expensive because the overhead in communications can be significant. Hence the way the PKLIST is coded can affect performance.

In a distributed environment, the communications overhead of searching is proportional to the number of PKLIST entries through which DB2 has to search. Hence, from a performance point of view, having one collection for all packages, that is, one entry in the PKLIST, is ideal.

Consider the following two alternative methods of coding the package list for a plan that requires remote access:

Alternative 1:

```
BIND PLAN(planname) PKLIST(REMOTE.CARDEAL.*, REMOTE.COMMON.*) ..
```

or

Alternative 2:

```
BIND PLAN(planname) PKLIST(REMOTE.CAR_ALL.*) ..
```

Alternative 1 logically breaks up the packages into two collections for ease of management and to allow sharing of common packages. Assume that SET CURRENT PACKAGESET is not used. Thus DB2 does not know which collection the package is in. As a result, DB2 first searches for the package in the first collection (CARDEAL). If DB2 does not find the package, it searches again using the second collection (COMMON). Each search is across the network. Hence, if the package resides in the second collection, there would be two communication exchanges between the two DB2s.

The preferred way (from a performance point of view) would be to use alternative 2, which involves only one search because there is only one entry in the PKLIST.

If there are not too many packages in the collections, a third alternative would be to specifically code the package_id in the package list. This allows the DB2 requester to determine the collection in which the package resides before requesting a search at the DB2 server. For example,

```
BIND PLAN(planname) PKLIST(REMOTE.CARDEAL.PKG1, REMOTE.CARDEAL.PKG2,...)
```

If you have to use a wild card, make sure that the sequence of coding reflects the frequency of use. The collection that is the most likely to be used should be coded first in the package list.

5.2.6 Performance Monitoring

With DB2 V2.3, accounting information is available at the plan level using DB2PM V2.1. The DB2PM accounting summary provides accounting information on both the requester and server. At the requester, the accounting is reported by PRIMAUTH/PLANNAME/REQLOC. At the server, accounting is similarly reported but under one plan only. The plan name is DISTSERV. All accounting information under this plan can be broken down by PRIMAUTH and REQLOC, where REQLOC is the requester DB2.

With DB2 V3, the accounting information is further subdivided into package_ids. This applies to the DISTSERV plan as well. DB2PM Version 3 supports package accounting.

See the *DB2 Administration Guide* for further details.

5.3 DB2/2 to DB2

This section discusses the use of DB2 packages when developing a DB2/2 application that accesses DB2.[4] It does not discuss the implementation of such an environment. For detailed coverage of the setup and implementation, see *Distributed Relational Database: Using OS/2 DRDA Client Support with DB2*. In such an application development environment, the host DB2 person would need to understand the implications of having the package bound from DB2/2. In fact, the host DB2 person may have to

[4] Please note that references in this section to DB2/2 are equally applicable to OS/2 Database Manager.

understand much more than VTAM* communications, distributed performance, and other issues, but these are not covered here.

5.3.1 DB2/2 to DB2 Remote Access

To access DB2 servers, DB2/2 requires DDCS/2, which has the necessary DRDA application requester protocol support that enables DB2/2 to access DB2. DDCS/2* functions are similar to the DB2 DDF.

Figure 57 shows a typical OS/2 application program that accesses a DB2 server. The figure shows how package information is derived from the program, DB2/2, and DDCS/2.

To establish the connection to a database server, the program uses the START USING DATABASE <database alias>. This is the equivalent of the CONNECT TO <location> statement for DB2. The DDCS/2 database connection services (DCS) directory maps the database alias to the location ID of the DB2 server.

DB2/2 does not *fully* support the idea of collections. The collection_id is fixed when the program is precompiled. At precompilation time, the userid is used as the collection_id. A fully qualified package name in the form of collection_id.package_id is hardcoded into the program when the program is precompiled. At the remote DB2 server, the package is allocated using the location_id, collection_id, package_id, and consistency token.

DB2/2 does not support the concept of a plan.

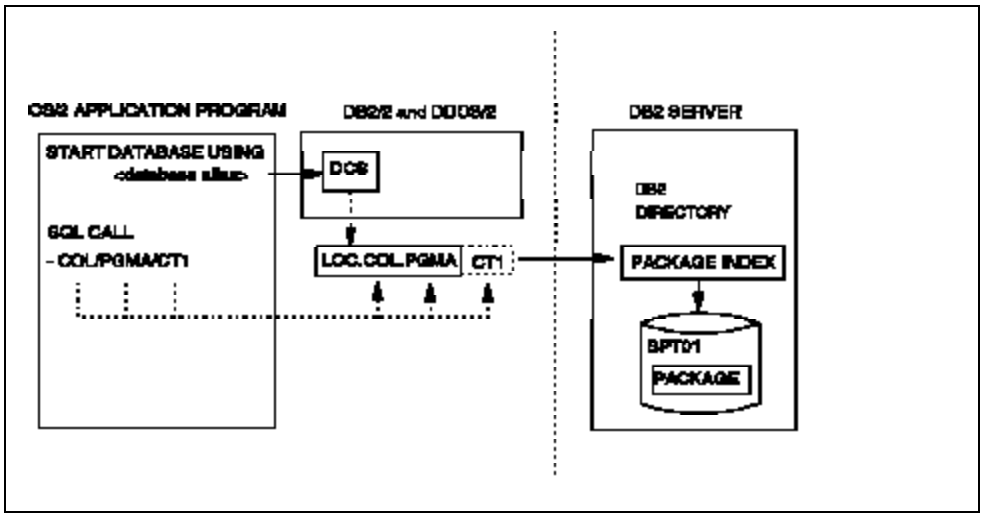

Figure 57. OS/2 Application Program Accessing DB2 Server

5.3.2 DB2/2 Program Preparation and Bind Process

Figure 58 shows the program preparation process for an OS/2 application with SQL access. The SQLPREP command compiles and link-edits the program and binds the package. There is no additional bind step as in the case of DB2.

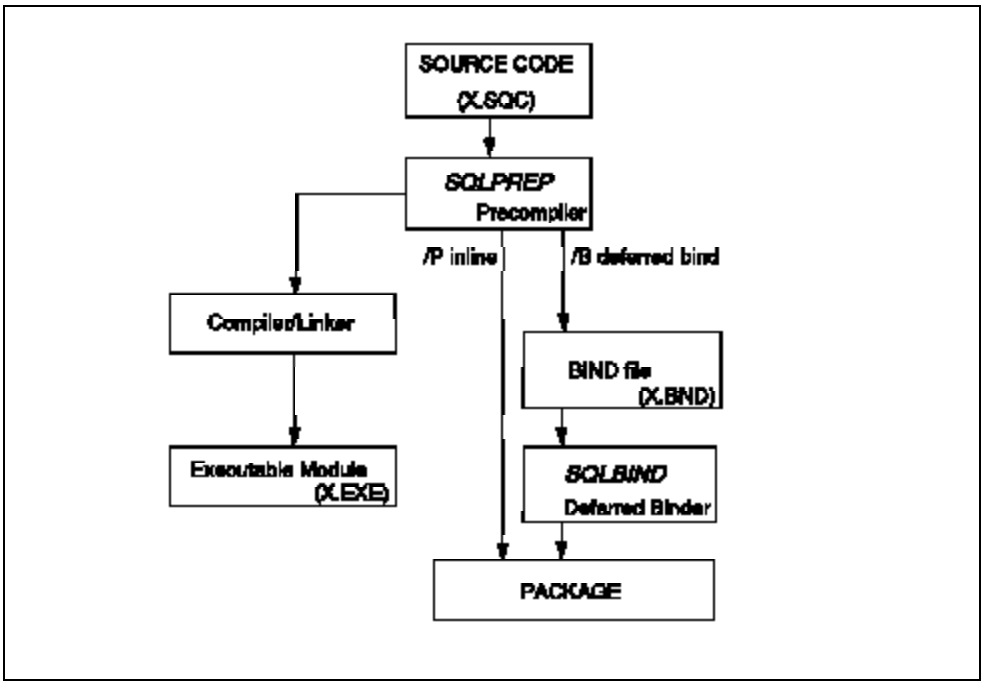

Figure 58. DB2/2 Program Preparation

In Figure 58, the points to note are:

- SQLPREP with the inline bind option (/P) compiles, link-edits, and binds the package in one step. The bind file is the DB2 equivalent of the DBRM.
- SQLPREP with the deferred bind option (/B) produces a bind file only and not the package. The SQLBIND command can be used later to bind the bind file to produce the package.
- Unlike DB2, SQLPREP does not have an independent SQL preprocessor. It relies on DB2/2 to verify the SQL and hence requires access to the database even if the deferred bind option (/B) is used. If the program accesses DB2, the SQLPREP accesses DB2 regardless of inline or deferred bind. In the case of deferred bind, SQLPREP preprocesses the SQL by requesting DB2 to create a package and then performs a ROLLBACK. Thus SQLPREP executed against DB2 with deferred bind still requires all package privileges, such as BINDADD and CREATE IN, at the DB2 server.

The binds that are necessary to support DB2/2 distributed applications that access DB2 servers remotely as well as the DB2/2 locally are discussed below.

Consider a program that accesses DB2/2 locally and two DB2 servers at BOSTON and NEW YORK. Figure 59 on page 125 shows the structure of the program and the binds required. SQLPREP is executed against the local DB2/2 server with the deferred bind option (/B) so that a bind file is produced (PGMA.BND). The SQLBIND command is used to bind the bind file to each location, including the local DB2/2.

Because the SQLPREP was executed against DB2/2, the above assumes that all SQL statements in the program are acceptable to DB2/2. If the program contains SQL statements that are not recognized by DB2/2, such as the OPTIMIZE FOR clause (which is DB2 specific), the SQLPREP will fail.

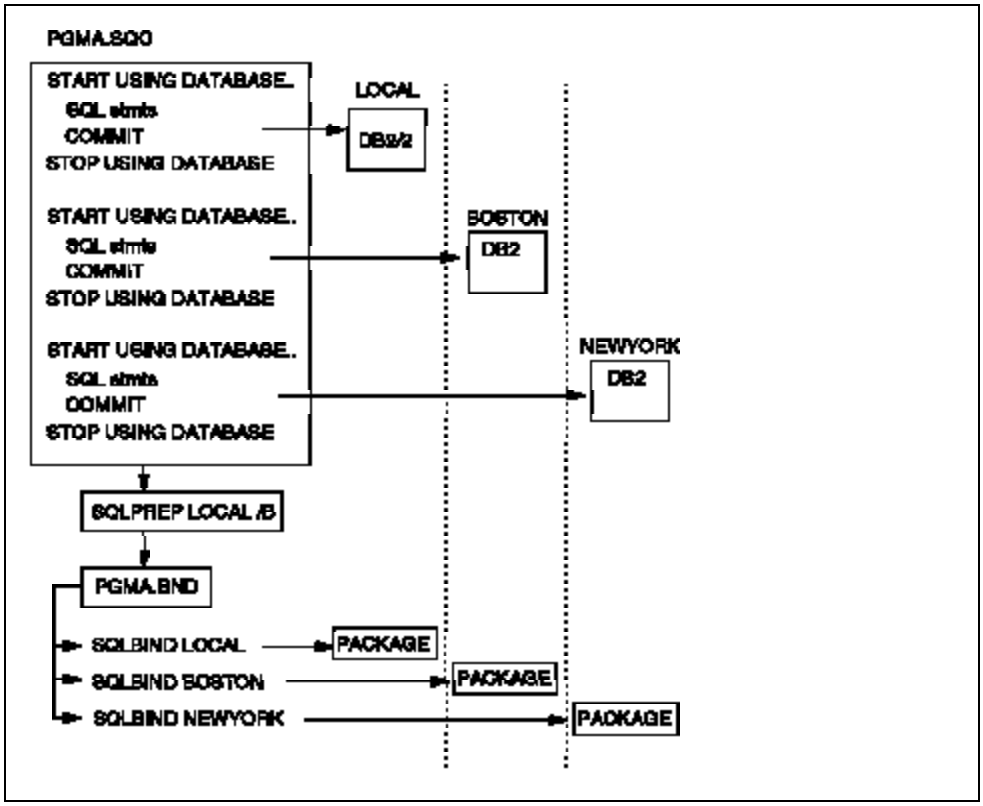

Figure 59. BINDs for DB2/2 Distributed Application

Note: The word *userid* in the DDCS/2 environment has the same meaning as *authid* in the DB2 environment.

When you use DDCS/2 to create a package at a DB2 database (by running SQLPREP or SQLBIND against a DB2 database), the following occurs:

1. The collection_id is determined by the userid that is logged on when the .BND file is created (if you do SQLPREP with the /B option) or by the userid that is logged on when the package is created “directly” (if you do SQLPREP without the /B option, that is, without creating a .BND file).

2. The creator and qualifier of the package are determined by the userid that is logged on when the package is created (by either SQLBIND or SQLPREP).

Thus, if userid AA does an SQLPREP against the database with the /B option, and userid BB does the SQLBIND, the following will result:

- Collection_id = AA
- Creator = BB
- Qualifier = BB.

If, however, userid AA does an SQLPREP against the DB2 database without the /B option (that is, does not create a .BND file), the collection_id, creator, and qualifier will be the same (AA in this example).

5.3.3 DB2/2 Bind Options

The DB2/2 bind process does not support all of the options provided by DB2. The DB2 administrator should be aware of the following SQLPREP and SQLBIND options:

/F option Specifies the date and time format to be used. The valid options for accessing DB2 are USA, EUR, ISO, and JIS. Local format (/F=LOC) is not valid for accessing DB2. The default is ISO.

/G option Requests that package execution privilege be granted to an additional userid (or PUBLIC). The specified userid will be granted the execute privilege without the GRANT option. Can only be specified in the SQLBIND command.

/I option Determines the isolation level for the DB2 package. DB2/2 supports RR, CS, and uncommited read (UR). DB2 supports only RR and CS. If UR is specified, it is converted to CS.

/K option Determines what type of block fetching will be done for ambiguous or unambiguous cursors. This is similar to the DB2 CURRENTDATA option. For a discussion of ambiguous cursors, see “CURRENTDATA” on page 119.

For a complete discussion of all SQLPREP options and SQLBIND commands, please refer to the *IBM Extended Services for OS/2 Guide to Database Manager*.

Note that SQLPREP does not support DB2's equivalent of the SQLERROR option to handle SQL superset statements (SAA SQL statements plus any product-specific SQL statements). A distributed application may access both DB2/2 and DB2. If the SQL is SAA, there would not be a problem with SQLPREP. If the program needs to use a DB2 SQL superset statement (for example, SELECT statement with the OPTIMIZE FOR clause), preprocessing would be different.

Table 9 on page 127 shows how the DB2 package attributes are derived from the DB2/2 bind environment.

The DB2 administrator may want to change the attributes marked with an asterisk by doing a rebind on the package at the host DB2.

Table 9. DB2 Package Attributes	
ATTRIBUTES	**VALUES**
Acquire	Use
Bind Auth	OS/2 userid for SQLPREP or SQLBIND
Copy Auth	OS/2 userid for SQLPREP or SQLBIND
Creator	OS/2 userid for SQLPREP or SQLBIND
Collection_id	OS/2 userid for SQLPREP
Defer Prep	Blanks
Enable/Disable*	All connections
Execute Auth	OS/2 userid for SQLPREP or SQLBIND
Explain*	No
Isolation	Cursor stability or repeatable read
Operative	Yes
Owner*	OS/2 userid for SQLPREP or SQLBIND
Qualifier*	OS/2 userid for SQLPREP or SQLBIND
Release	Commit
SQLERROR	Nopackage
Valid	Yes
Validate*	Run
Version	Blanks
Remote	Yes
PDS name	SQL03033 <luname>

The DB2/2 environment is different from SQL/DS and DB2 in terms of support for DB2 attributes. Table 9 shows, for example, that the userid for SQLPREP or SQLBIND becomes the collection_id and the qualifier attributes of the resultant DB2 package.

Recall that the collection_id is hardcoded by SQLPREP into the application. At the time of writing, the only solution is to use functional IDs to perform SQLPREP or SQLBIND. For example, if the required collection_id is CARDEAL, a userid by the same name CARDEAL would be used to prepare and bind the program.

Similarly, the qualifier for tables may be resolved by using functional IDs. If this is the case, the functional ID would be the collection_id and the high level qualifier (owner) of the tables at the DB2 server.

Alternatively, the qualifier can be changed by rebinding the package at the host DB2 and specifying a new qualifier.

Depending on user requirements, a rebind at the DB2 server may also be required to:

- Find out the DB2 access path for the package
- Prevent use of package by unauthorized connections

- Change run-time validation to bind-time validation for performance
- Change ownership and execute authority of the package for security reasons
- Promote a package from one environment to another (for example, from test to production) using the bind COPY option.

5.3.4 Authorizations for DB2/2 Requesters

To bind the package at the DB2 server, the following authorizations and privileges must be granted to the userid performing the SQLPREP or SQLBIND:

- Privileges to execute the package's static SQL statements
- BINDADD
- CREATE IN COLLECTION <collection_id>. The collection_id is the userid used to run SQLPREP against the program.

If the userids performing SQLPREP and SQLBIND are different (when deferred bind is used), the same privileges must be given to both userids.

If the privileges are not granted, the following error message is usually seen:

```
SQL0552N  "<userid>" does not have the privilege to perform
        operation "BIND".
```

When there is a bind privilege error, DB2/2 usually returns SQL0552N. This message simply says that the userid does not have the privilege to perform a bind. To make the error messages reflect whether the error is due to BINDADD or CREATE IN privilege, some changes have to be made on the host to the OS/2 SQL code mapping file called DCS1DSN.MAP, which is in SQLLIB directory. Edit the file and change the line that appears as:

```
-567, -552, (c2c,c1c)
```

to read:

```
-567, -552, (c2c,c4c)
```

If you want to know more about mapping files, please refer to the *IBM Extended Services for OS/2 Guide to SAA Distributed Database Connection Services/2*.

The userid performing the SQLPREP (inline bind) or SQLBIND (deferred bind) automatically receives the following privileges at the DB2 server:

- BIND, REBIND, DROP
- EXECUTE
- COPY.

To execute a DB2 package, the EXECUTE privilege on the package must be granted to the userid logged on to DB2. DB2 V2.3 does not allow secondary authids to be used for execute package authority checking for non-DB2 application requesters. If secondary authids are to be used, APAR PN28249 is required.

5.3.5 Handling DB2-Specific SQL

When an OS/2 program accesses DB2/2 as well as DB2, it may use DB2-specific SQL, which is not acceptable to DB2/2. The SQLPREP will fail if it is issued against DB2/2, which does not have the equivalent of DB2's SQLERROR option to handle such a situation (see “SQLERROR” on page 115 for DB2's SQLERROR option).

Figure 60 shows how it is possible to compensate for the lack of an SQLERROR option in the DB2/2 bind. In the example, the program needs to access the local DB2/2 and the remote DB2 server. When accessing the DB2 server, the program uses the OPTIMIZE FOR clause in the SQL statement. The SQLPREP does not allow this DB2-specific SQL clause if the target database is DB2/2. It will be successful if the SQLPREP is targeted at DB2 instead. Hence, the solution is to divide the program into two programs. The first program consists of DB2/2 SQL statements and will undergo an SQLPREP against the local DB2/2. The second program, which contains the DB2 SQL superset statement, will undergo an SQLPREP against the DB2 server.

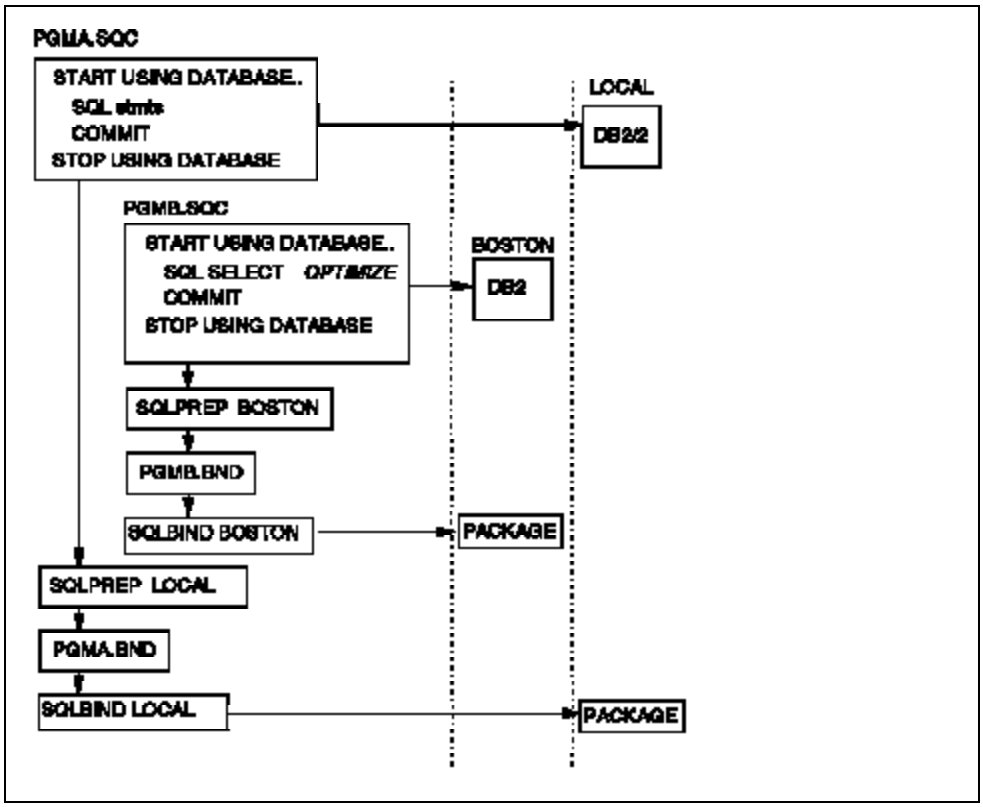

Figure 60. OS/2 Application Program Using DB2-Specific SQL

DB2/2 does not support DB2's VERSION precompiler option. Its own versioning support is shown in Figure 61 on page 130. The default package name is the program name. By using the /P option, you can change the package name to a name that reflects the version.

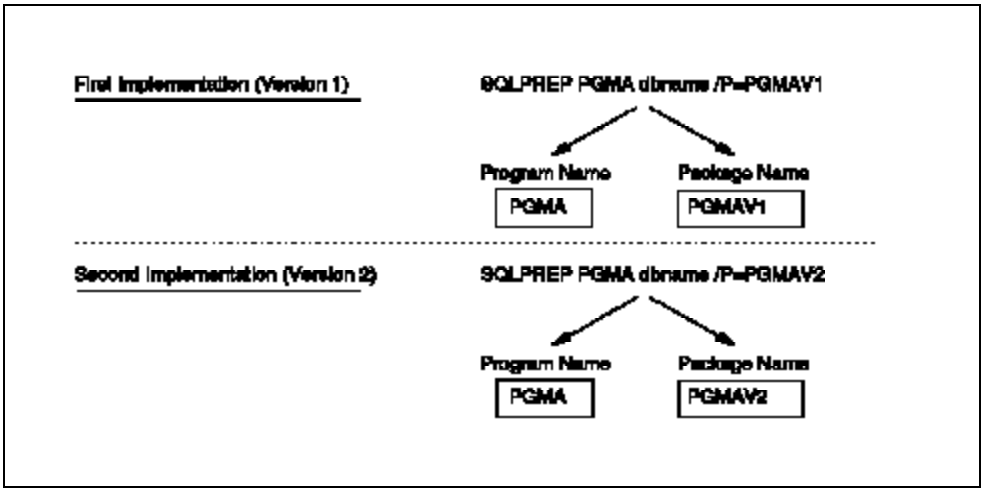

Figure 61. DB2/2 Application Version Support

5.3.6 DB2/2 to DB2 Considerations

Collection_id fixed

The collection_id is fixed at SQLPREP time and is hardcoded into the program. It is also the userid performing the SQLPREP. In a development environment that accesses a test DB2, for example, this would mean a collection for every userid. This arrangement may differ from that of the host DB2 programmers who may share collections. At large test environments or in a production environment, functional IDs may have to be set up on the LAN so that the desired collection_ids are used.

Owner_id not passed at execution time

In a DB2 to DB2 environment, DB2 uses the owner_id, passed in the DRDA flow, to check package execute authority at package execution time. This is the plan owner_id. By granting EXECUTE privilege on the package to the owner_id at the server site, the administrator at the server need not grant the privilege to every requester ID that executes the package.

However, DDCS/2 V1.0 does not pass the plan owner_id to the DB2 server at execution time. It is different from a DB2 requester in that it does not have a plan, nor does it allow the user to specify the plan owner_id or equivalent ID at execution time. Hence, at the DB2 server, EXECUTE privilege has to be granted to every DDCS/2 userid. APAR 28249 alleviates the problem by making DB2 check secondary authids for EXECUTE package authority, and thus only a single EXECUTE privilege grant needs to be given to the secondary authid.

The DB2 administrator must be aware of the difference in treating DB2 and non-DB2 requesters in terms of granting EXECUTE privilege for packages.

Qualifier

SQLPREP uses the user_id for any unqualified tables. This is desirable only if the accessed host tables belong to the same user_id as well. If not, functional_ids may have to be set up to get the desired qualifiers. Alternatively, a rebind on the DB2 server specifying the needed qualifier can be done.

DDCS/2 and NULLID

DDCS/2 supplies bind files that have been SQLPREPed with NULLID as the collection_id. These bind files have to be bound at the DB2 server using the SQLJBIND command. For the binds to work, the necessary privileges have to be granted to the user_id accessing DB2:

- GRANT BINDADD TO <SQLJBIND userid>
- GRANT CREATE IN COLLECTION NULLID to <SQLJBIND userid>.

5.4 SQL/DS to DB2

This section discusses the package-related issues for SQL/DS distributed applications that access DB2. It covers the SQL/DS program preparation and bind processes and how they affect the DB2 package created at the DB2 server.

For information regarding DRDA implementation and application design and coding, please refer to *Distributed Relational Database Remote Unit of Work Implementation DB2-DB2 and DB2-SQL/DS, Volumes 1 and 2*.

5.4.1 SQL/DS to DB2 Remote Access

Figure 62 on page 132 shows a distributed SQL/DS application that accesses a remote DB2 server through DRDA.

The SQL/DS program uses the SQL CONNECT TO command to establish an SQL connection with a remote server. When the SQL/DS program executes the SQL statements, there is no choice as to which package to use. Plans, package list, and SET CURRENT PACKAGESET are not supported by SQL/DS. A fully qualified package name in the form of collection_id.package_id is hardcoded into the program when the program is precompiled. At the remote DB2 server, the package is allocated using the location_id, package_id, collection_id, and consistency token.

It is worthwhile to point out that SQL/DS itself does not support the concept of collections. An SQL/DS package has a two-part name:

```
Creator-Authid.Package_name
```

SQL/DS packages are logically grouped by creator authids rather than by collections. SQL/DS 3.3 supports only the ability to specify the collection_id when binding a package at a DB2 server.

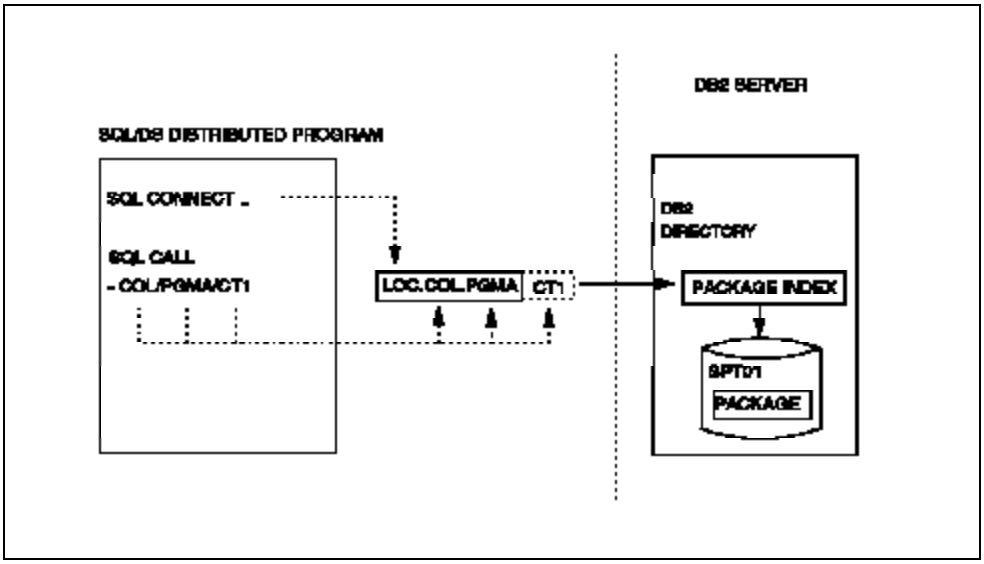

Figure 62. SQL/DS Accessing a Remote DB2 Server

5.4.2 SQL/DS Program Preprocessing and Bind

The SQLPREP preprocessor in SQL/DS precompiles the program source and binds the package in one step. This is different from DB2's precompiler stage and bind stage. Hence, in SQL/DS, a DBRM is not produced. The output from an SQLPREP is the modified source code and the package bound at the remote server.

Figure 63 shows the SQLPREP process.

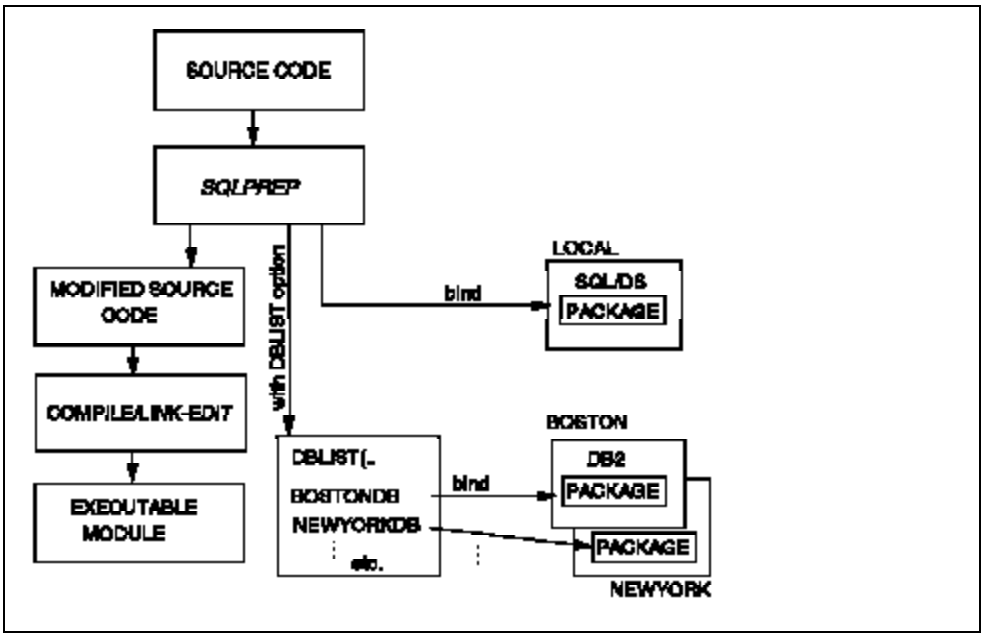

Figure 63. SQLPREP Process

SQLPREP can be used to bind the package against several database servers at the same time. The DBLIST option specifies the databases where the package is to be bound. This specification facilitates the bind process because the DBRM is not produced for subsequent binds. When DBLIST is specified, SQL/DS actually does an SQLPREP against each database specified, one by one using the same options. This operation may not be desirable if different options are required for different databases in an unlike environment.

If DBLIST is not specified, the target of the bind would be the local SQL/DS database machine.

When the server is DB2, the resultant package created can be replicated in any other DB2 sites by using the DB2 BIND command with COPY option. If the server is SQL/DS, the package can be replicated in other SQL/DS sites by using SQL/DS UNLOAD PACKAGE and RELOAD PACKAGE.

The following is an example of the SQLPREP command:

```
SQLPREP COBOL
  PREPPARM(PREPNAME=<collection_id>.<package_id>,
          OWNER(<owner_id>),
          QUALIFIER(<qualifier>),
          ISOLATION(<isolation level>),
          BLOCK/NOBLOCK,
          EXPLAIN(<YES/NO>),
          CTOKEN(<YES/NO),
          ERROR,
          ......
          )
          DBLIST(<dbname1>,<dbname2>,..)
```

5.4.3 SQLPREP Parameters

The following SQLPREP parameters are relevant when binding to a DB2 server:

Collection_id: Specifies the collection_id of the package created at DB2. Supported only for DB2 servers including SQL/DS. Default is the SQLPREP CMS ID or userid.

Package_id: Specifies the package_id if different from the program name. Default is the program name.

Owner: Specifies the owner of the package on the DB2. Default is the CMS ID or userid. This is supported only for DB2 servers.

Qualifier: Specifies the qualifier for any unqualified tables referenced in the program. Default is the CMS ID or userid. This is supported only for DB2 servers.

ISOLATION: Specifies the isolation level.

BLOCK/NOBLOCK: Specifies whether block fetch operations should be performed.

EXPLAIN: EXPLAIN is supported when binding on the DB2 server.

CTOKEN: Specifies whether the consistency token is to be used. CTOKEN(YES) would use a time stamp for the token. CTOKEN(NO) would leave the token as blanks (hex '40'). This would be similar to using blanks for the DB2 precompiler option, LEVEL.

The SQL/DS preprocessor offers options to handle non-SQL/DS SQL statements and missing objects and authorizations. These options are necessary for distributed applications where the SQL may be DB2-specific and not all the objects may be at a particular server. These options are similar to DB2's precompiler SQL option and SQLERROR bind option. Table 10 shows the relationship between the SQL/DS and DB2 options. For more information, please refer to “SQLERROR” on page 115.

Table 10. SQLPREP Options for SQL, Objects, and Authorizations

SQL/DS Option	Description (When Used)	DB2 Equivalent
NOCHECK	SQL/DS servers only All objects and authids exist	SQL(DB2) and SQLERROR(NOPACKAGE)
CHECK	Validate SQL only	No equivalent
ERROR	Non-SQL/DS servers	SQL(ALL) and SQLERROR(CONTINUE)
EXIST	All objects and authids exist	VALIDATE(BIND) and SQLERROR(NOPACKAGE)
NOEXISTS	Missing objects and authids	VALIDATE(RUN) or VALIDATE(BIND) and SQLERROR(CONTINUE)

5.4.4 Authorizations for SQL/DS Requesters

The following privileges must be granted at the DB2 server to the owner_id of the package. The owner_id of the package is specified using the SQLPREP option OWNER. If the SQLPREP option OWNER is not used, the owner_id is the userid (or CMS ID) used to issue the SQLPREP command.

The privileges are:

- Privileges to execute the package's static SQL statements
- BINDADD privilege
- CREATE IN COLLECTION <collection_id>. The collection_id is specified in the SQLPREP option PREPNAME. If the collection_id is not specified, then it will be the VM* logon ID.

At the SQL/DS requester site, authid translation may occur for outgoing traffic by means of the NAMES file of the requester.

At the DB2 server end, authid translation may occur if specified in the SYSIBM.SYSUSERNAMES table. For SQLPREP, authids that might be translated are:

- Authid performing the SQLPREP (the CMS ID)

- The collection_id specified in SQLPREP
- The owner_id specified in SQLPREP.

The qualifier specified in SQLPREP is *not* translated.

EXECUTE privilege needs to be granted to any requester who wants to execute the package. For non-DB2 application requesters, DB2 does not allow secondary authids to be used for execute package authority checking unless APAR PN28249 or PTF UN30436 is applied. This applies to non-DB2 application requesters only.

For a discussion of authid translation and security in an unlike environment, refer to *Distributed Relational Database Remote Unit of Work Implementation DB2-DB2 and DB2-SQL/DS Vol. I*.

5.5 OS/400 to DB2

OS/400 supports DRDA both as a server and a requester. This section describes how OS/400 accesses DB2 and how the package can be created. We recommend that you also refer to Chapter 9 of *Structured Query Language/400*: A Guide for Implementing OS/400 Version 2 Release 2.0*.

Similar to other databases, a distributed OS/400 program issues a CONNECT TO statement to establish connection to a remote server like DB2. If CONNECT TO is not specified within a program, the database specified in the RDB parameter of the compile will be used. The default value for this parameter is *NONE, which means that no DRDB processing is possible and only the local database can be used.

5.5.1 OS/400 Program Preparation

Figure 64 on page 136 shows the program preparation for an OS/400 program that accesses DB2. Only one command is needed to precompile and compile an OS/400 SQL program, and the command can also create a package on a specified application server. The command is CRTSQLxxx, where xxx represents the language compiler to invoke (CBL for COBOL, FTN for FORTRAN, RPG for RPG, C for C Language, and PLI for PL/I). The command is executed specifying which remote database is to create the package. The extracted SQL is shipped to the DB2 server, and a package is created. The collection_id and package_id are taken from the CRTSQLxxx option SQLPKG, which is discussed below. The default package_id is taken from the CRTSQLxxx option, SQLPKG. The default package_id is the same as that of the collection or library of the program on the OS/400.

Note: In OS/400, the word *collection* is used to group objects for reference purposes. A fully qualified object name is rdbname.collection.objectname. The object could be a program, table, or package. Hence, the word *collection* in OS/400 not only qualifies packages, but all objects as well. The collection is also known as the library.

The BIND options relevant to the DB2 environment are:

PGM(aaa) Specifies the program to be prepared.

COMMIT(*NONE/*CHG/*CS/*ALL) *CS translates to DB2 CS, and *ALL translates to DB2 RR.

RDB(rdbname) Specifies the remote database name where the SQL package is to be created. It must be defined in the OS/400 relational database (RDB) directory. This can be done using the WRKRDBDIRE command.

USER(userid) Specifies a valid userid that will actually perform the package bind at the remote site. Package creation privileges must be granted to this userid at the DB2 server.

PASSWORD(password) Specifies the password for the userid as indicated in the USER parameter.

Naming Convention The *SYS value must be used for this parameter in a DB2 environment because it sets the table naming convention to COLLECTION.TABLE rather than to the OS/400 default value of *SYS, which represents COLLECTION/TABLE.

DFTRDBCOL(collid) Specifies the default qualifier or collection for tables at the DB2 server.

SQLPKG(collid/pkgname) Specifies the collection_id and package_id at the DB2 server if different from the default. The default is (*PGMLIB/*PGM). The default as used on the OS/400 may not fit DB2 name lengths. For example, a package_id on the OS/400 can be up to 10 characters, while for DB2 it is 8 characters. These values are hardcoded into the program and are used during execution time to locate the package on the DB2 server.

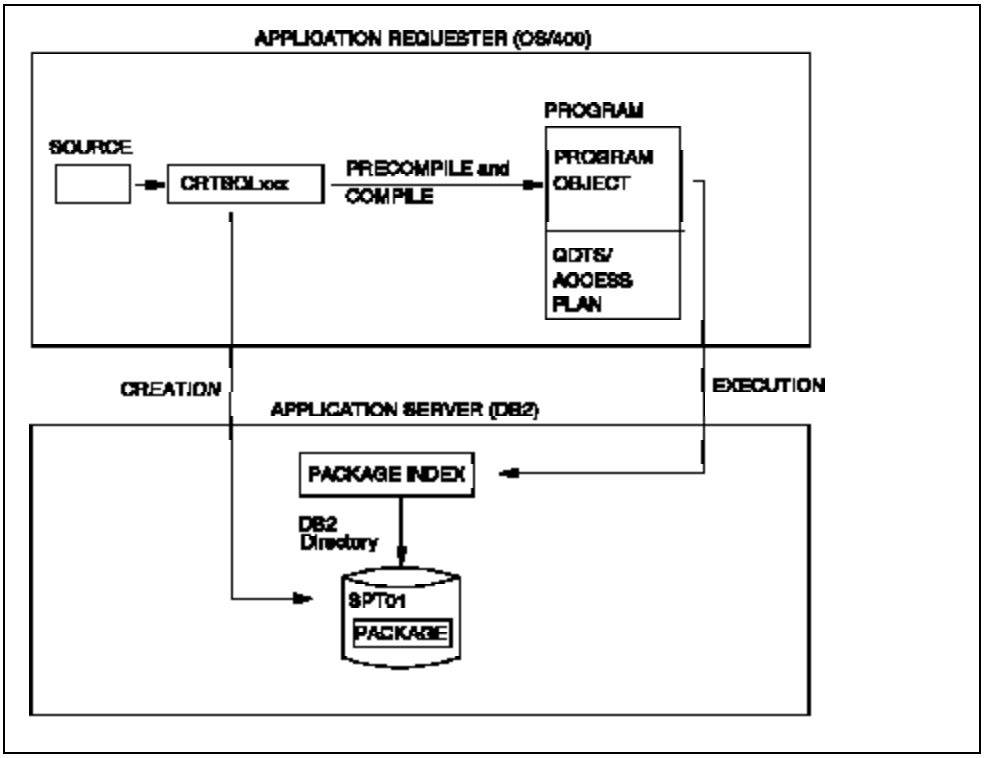

Figure 64. OS/400 Program Preparation Using CRTSQLxxx

If errors occur during the execution of the CRTSQLxxx command, the package is not created, and the command ends normally. Changing the default generation levels may help. Also, the package creation listing will always have messages even if the default option *NOSRC was specified. The listing should be checked for warning or error messages.

Sometimes there may be a requirement to create a package after the program has been compiled. This may be useful if the naming conventions and change procedures at DB2 are different from those on the AS/400, or if the AS/400 program needs to access more than one remote database. The command to bind the package after the program has been created is:

```
CRTSQLPKG PGM(aaa)
  RDB(remoteRDB)
  USER(userid)
  PASSWORD(password)
  DFTRDBCOL(collid)
```

The parameters are the same as those described for the CRTSQLxxx command.

In a networking environment where there are a number of AS/400s as well as a DB2 system that may act as an application server, a package can be created on the local AS/400 system for later distribution to other AS/400s. A package is not needed on the local AS/400 because the compiled program contains all required SQL definitions.

To handle DB2-specific SQL that OS/400 does not recognize, the GENLVL parameter of the CRTSQLxxx command should be set to 31 to allow the creation of the program. This would bypass the level-30 severity errors caused by unacceptable SQL syntax generated by the SQL precompiler. Setting the GENLVL parameter to 31 is equivalent to the DB2 precompiler option SQL(ALL) and bind option SQLERROR(CONTINUE).

Both the Interactive SQL (ISQL) and Query Manager/400 (QM/400) components of the SQL/400 program product require a package on a DB2 application server. The package is automatically created the first time it is required in DB2 if the ISQL or QM/400 user has BINDADD and CREATE IN COLLECTION QSQL400 authority. Subsequent users of ISQL and QM/400 will use the same package. The package name will be QSQLabcd, where the values of a, b, c, and d are as shown in Table 11.

Table 11. Package Name for ISQL and QM/400

Position	Option	Value
a	Date format	0=ISO/JIS 1=USA 2=EUR
b	Time format	0=JIS 1=USA 2=EUR/ISO
c	COMMIT level and decimal delimiter	0=CS/period 1=CS/comma 2=RR/period 3=RR/comma
d	String delimiter and default characters	0=apost/SBCS 1=apost/DBCS 2=quotes/SBCS 3=quotes/DBCS

5.5.2 Authorizations

The following privileges are required to bind a package at the DB2 server:

- Privileges to execute the package's SQL statements
- BINDADD
- CREATE IN COLLECTION <collection_id>. <collection_id> is the collection specified in the CRTSQLxxx option SQLPKG.

The privileges must be granted to the package owner that will be either the userid specified on the *USER parameter of the CRTSQLXXX command or the current user profile name of the OS/400 programmer by default.

Although OS/400 does not support outgoing or incoming userid translations, DB2 does through SYSIBM.SYSUSERNAMES and signon exits. Hence, the privileges listed above are for *translated* userids. Note, however, that the collection_id specified in the CRTSQLxxx option SQLPKG and the qualifier for tables as specified in CRTSQLXXX option DFTRDBCOL are not translated.

The privilege to execute a package must be granted to the userid that will be the OS/400 user profile (USRPRF). To allow secondary authids to be used for execute package authority checking, APAR PN28249 must be applied.

Chapter 6. Implementing Packages

There is no single solution on how best to implement packages. The approach an installation takes will depend to a large extent on its development procedures and the options of the new package bind feature that it will implement.

The terms *migration to packages* and *conversion to packages* are used in this chapter. In this context, migration refers to the modification of the application development and production procedures to support packages; conversion refers to the modification of existing application plans to use packages.

This chapter discusses the factors that should be considered in planning for and migrating to the use of packages. The topics covered include:

- Migration planning
- Modification of existing plans to use packages
- Getting started with packages
- Package administration.

6.1 Migration Planning

This section discusses the factors that should be considered when establishing guidelines for the use of packages. These factors include the scope of collections to be used in the various test and production environments and the use of versioning, new authorization privileges, and new bind options.

Based on the guidelines established, the modifications required for existing procedures can be determined and estimates for completing them prepared.

6.1.1 Migrating to DB2 V2.3

Existing applications do not need to be precompiled, compiled, or relinked, nor the plans rebound when they are migrated to DB2 V2.3. Therefore, it is recommended that the migration to DB2 V2.3 be completed for all applications and that all environments be stabilized before any migration from plans to packages is attempted.

If DB2 V2.3 is already installed and stabilized, planning for package implementation can begin immediately. If DB2 V2.3 has not yet been installed, the planning and preparation for implementing packages can be done in parallel with the installation and testing activities.

Ideally, all test and production environments should be converted to DB2 V2.3 before migration to packages begins. Where this is not possible and different releases of DB2 will be used for a considerable period, steps need

to be taken to ensure that problems do not occur when applications are promoted. The assumption in this case is that the test environments are using DB2 V2.3, while the production environments are using DB2 V2.2. The actions needed will depend on the method of promotion.

Where promotion through each of the test levels to production involves precompile, compile, and link of the load modules, there should be no problems, provided that functions unique to DB2 V2.3 are not used.

Problems may occur where promotion is done by copying the load modules and the DBRMs if DB2 in the new environment is at an earlier release level. These problems can be avoided by applying the Small Programming Enhancement (SPE) UL90583 to DB2 V2.2. This allows DBRMs precompiled on DB2 V2.3 to be bound on DB2 V2.2.

Further problems may occur where there is distributed processing using DDF between DB2 subsystems that are at different release levels. Table 12 shows the results of having mixed DB2 releases at the requester and server sites executing programs precompiled under different DB2 releases.

Table 12. DBRM Release Dependencies for DDF Processing

DB2 Release Level			Processing Viability
DBRM	Requester	Server	
V2.3	V2.3	V2.2	Works with maintenance applied
V2.3	V2.2	V2.3	Does not work
V2.2	V2.3	V2.2	Works with maintenance applied
V2.2	V2.2	V2.3	Works with maintenance applied

The table shows that the simplest solution for supporting DDF-based applications involving subsystems with mixed DB2 releases is to use DBRMs created from precompiler runs using DB2 V2.2 until all environments have been migrated to DB2 V2.3.

6.1.2 Understanding Packages

Before any decisions regarding the use of packages are made, it is essential that those responsible for implementing packages understand package concepts and their use. Education courses and experience in the use of packages, for example, in a pilot project, would provide much of this needed understanding.

A suitable pilot project could be based on the common SQL routines used in the installation. Conversion of these routines to packages would provide some of the experience required, test the current preparation and promotion procedures, and provide immediate benefits by removing the need to rebind many plans when one of the modules is changed.

6.1.3 Collections

Perhaps the most important decision to be made in implementing packages is the scope of the collections that will be used in the test and production environments. Scope in this context refers to the selection criteria used in allocating packages to collections and could be, for example, a plan, application, business area, or environment.

The advantages and disadvantages of using these different scopes are summarized below.

Collection Scope Comments

Package This entry is included for completeness only. The administrative and performance implications would generally make this approach untenable.

Plan Advantages

- Gives one-to-one conversion for existing applications
- Provides accounting reports at plan level as at present
- Provides greater security control than the application and business area scopes
- Provides good performance as only a single PKLIST entry is needed for the collection.

Disadvantages

- Greater administrative effort required:
  - More authorities to be GRANTed and REVOKEd
  - More BIND/REBIND activity required to set up and maintain the collections as many packages would be bound into more than one collection.

Application Advantages

- Less administrative effort than above but may still be high
- Equal performance for package retrieval
- Monitor reporting possible at existing plan level or at application level using a plan with a single PKLIST entry.

Disadvantages

- Less security as user has greater access to data than when collections are based on plans

Business Area Advantages

- Even less administrative effort
- Equal performance for package retrieval
- Monitor reporting possible at existing plan level or at business area level using a plan with a single PKLIST entry.

Disadvantages

- Even less security as user has even greater access to data

Environment Advantages

- Least administrative effort
- Equal performance for package retrieval

  While the performance in package retrieval may be equal, collections of greater scope may make optimizing performance in other areas easier. For example, a scope of application may allow easier tuning of the CICS DB2 connection than a scope of plan.
- Monitor reporting possible at existing plan level, application level, or at business area level using a plan with a single PKLIST entry.

Disadvantages

- Least control. Security may have to be provided through external controls.

  It is possible to restrict the packages that can be executed from a collection by limiting the execute authority on individual packages to specific authorization IDs. However, the administrative effort required to set up and maintain this approach would be very high.
- Least granularity

  Having a single collection containing all packages for an environment appears at first glance to have merit. However, compromises will usually be required at package bind time for such collections. For example, a package that is used in a number of transactions may require different locking options for greatest efficiency. This is not possible if the package is bound only once in the collection. There may also need to be trade-offs between the bind RELEASE options and the EDM pool size.

For each of the scopes described above, the number of packages that would be bound into a collection could vary greatly across installations. For example, a scope of application in one installation may result in collections that contain packages numbered in the tens or low hundreds. The same scope in another installation may result in collections that contain a thousand packages or more.

There is no single answer to the question on how the scope of collections should be established. As a general guideline in migrating to packages, the recommendation is that extremes in scope be avoided and that the packages in a collection be numbered in the tens or low hundreds. Installations should review this guideline based on their experiences in the use of packages in their own environments.

6.1.4 Use of VERSIONing

Versioning using the new precompile parameter is another option to be considered. Where the VERSION parameter is to be used, decisions need to be made on:

- The format of the value assigned to this parameter
- Whether the value can be generated and entered at precompile time automatically or has to be entered manually

  Refer to Figure 40 on page 92 for details of a CLIST that automatically provides a new version value at precompile time.

- The number of versions of a package that are retained in each environment and the method used to control these instances.

This control is needed not just to ensure that the correct version of a package is executed but to avoid growth in some of the DB2 catalog and directory tables. This growth is most likely to occur when a program is being developed, as many recompiles may be necessary. Growth in the number of entries in the catalog and directory will not occur if the same value in the version parameter is used each time. This precludes the use of AUTO as the version value until the program is working.

Another method of avoiding growth in the catalog and directory during program development would be to FREE all existing versions before binding the package after program preparation.

6.1.5 Package List Entries

The impact of the number and types of entries in the package list on the processing required to locate a package is discussed in Section 2.2.2, “Package Search Overview” on page 33. Of particular concern is the cost of the package search where a number of wild-card package_ids, that is, of the form COLL1.*, are in the package list and the PACKAGESET register is not set. Package lists consisting of a single entry such as this normally perform better than lists containing multiple entries.

The performance achieved is influenced by many factors, such as the size of the EDM pool, the RELEASE value specified at bind time, and the access pattern to packages in the collection. However, as the number of entries in the package list increases, so do the instances where multiple accesses to the DB2 Directory (SPT01) may be required to locate the package at execution time.

The following package list entry types should be used with caution:

***.PGMA** All collections containing PGMA

. All packages in all collections

.*. All packages in all collections at all locations.

Where entries of this type are included in the package list, a check is always made at execution time on the plan owner's authority to execute the package. This check is made each time a package is allocated to the plan.

If the package remains allocated to the plan for a considerable period, the overhead on this check will be small in comparison to the total processing. However, if the package is allocated and deallocated frequently, the overhead of this check could be significant. See also Section 7.1.1, “Package List Organization” on page 165.

6.1.6 Use of New SQL Statements

Use of the SET CURRENT PACKAGESET statement in supporting mirror tables is discussed in Section 3.2, “Mirror Tables” on page 45.

The other major use of this statement is to improve package search performance where the package list contains a number of wild-card package_id entries. Each wild-card package_id id entry represents a possible extra access to the directory in locating a package. The use of the PACKAGESET special register allows these extra accesses to be avoided by specifying the collection_id to be used in locating the package.

6.1.7 When Plan and Package BIND Options Differ

Options that in previous releases applied at the plan level at BIND time can now be specified at the package level. Generally, the options specified when binding the package take precedence over the options specified when the plan is bound.

Locking

The isolation level and duration of locks are controlled by the ISOLATION and RELEASE options. If these options are not specified when a package is bound, the default values on the local server are those used to bind the plan under which the package is executed. If values for the options are not specified when the plan is bound, the default values are ISOLATION(RR) and RELEASE(COMMIT). If the package is bound with a different set of options from those used to bind the plan, the package options override the plan options for locks acquired during the execution of the package.

Not only can a package execute under a plan with different isolation and release options from those of the plan, but two or more packages with different sets of options in effect can acquire locks on the same resource during the same execution of a plan (see Figure 65 on page 145). Where this occurs, a single lock will be held on the resource, with isolation level and release options determined by the following rules:

- If any of the packages requires ISOLATION(RR), the lock will be held with this option. Otherwise the option will be ISOLATION(CS).
- If any of the packages requires RELEASE(DEALLOCATE), the lock will be held with this option. Otherwise the option will be RELEASE(COMMIT).

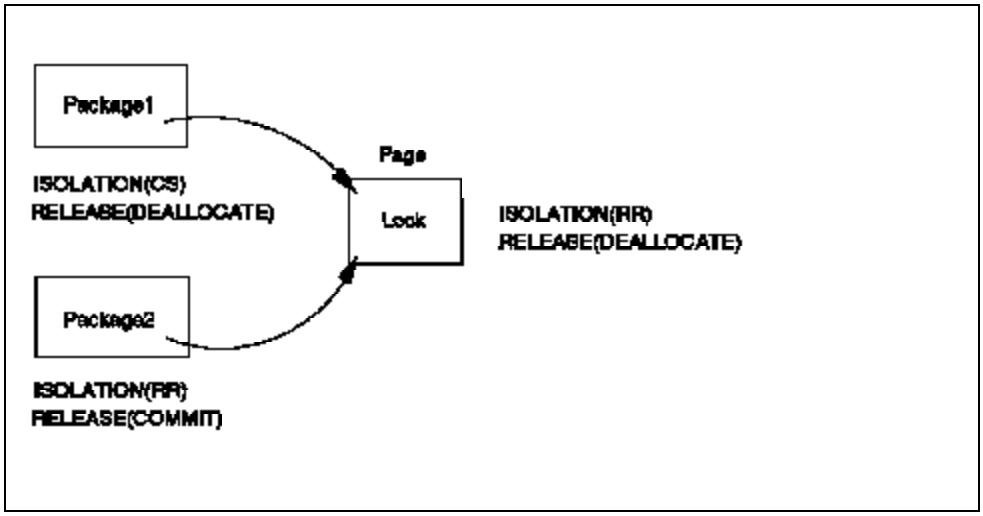

Figure 65. Lock Conflict on the Same Resource. A single lock is held on the page with ISOLATION(RR) and RELEASE(DEALLOCATE).

If one of the packages running under the same plan commits, locks, including those held by the other packages of the plan, are released according to the RELEASE option of the package that owns the lock (see Figure 66).

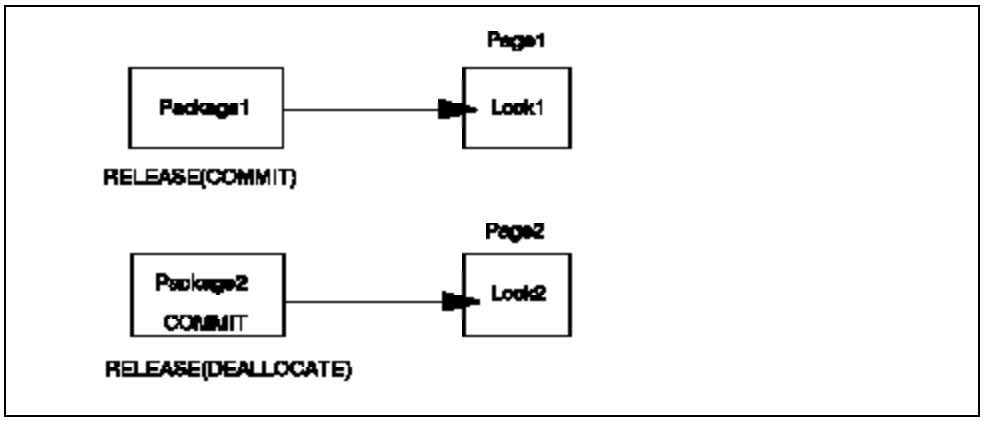

Figure 66. RELEASE Rules. When Package2 commits, Lock1 is released, and Lock2 is maintained.

The Effect of WITH HOLD for a Cursor

For a cursor defined as WITH HOLD, the cursor position is retained past a commit point by maintaining the locks needed to hold the position, even if they were acquired with ISOLATION(CS) or RELEASE(COMMIT). The locks needed to retain position for the cursors are:

- **Page locks**

  All necessary locks are held past the commit point with X and U locks demoted to S locks as all changes have been committed. The release of these locks following the commit point is governed by the options in

effect when they were acquired. Locks acquired under CS are released when all cursors on the page are moved or closed. Locks acquired under RR are released at the next commit.

- **Table, tablespace, and DBD locks**

  All necessary locks are held past the commit point and will be released subsequently according to the RELEASE option under which they were acquired; that is, at the next commit point when COMMIT was specified or at application deallocation where DEALLOCATE was specified.

6.1.8 CICS Application Design and Packages

The advantages of packages in allowing a large number of DBRMs to be associated with a single plan without the overheads previously associated with rebinding are particularly relevant to CICS. Often in CICS design, many program modules are invoked under a single transaction identifier. In early releases of DB2, this resulted in the transaction requiring a large plan that had to be rebound each time any of the program modules changed. Not only were these binds slow and resource intensive but they had the added disadvantage that the entire transaction was unavailable during the rebind process.

To address this problem, dynamic plan selection (DPS) was introduced in DB2 2.1. DPS allows these CICS transactions to be designed around a number of smaller plans that can be dynamically selected at execution time by means of an existing routine. Replacing the large plan with a number of smaller plans reduces the maintenance and availability problems mentioned earlier. However, the use of DPS is restricted to nonprotected threads and requires an implicit or explicit CICS SYNCPOINT to allow switching between plans.

The use of packages removes not only the need for DPS but also the need for SYNCPOINTs and the use of nonprotected threads. In addition, the ability to handle a large number of packages in a plan containing a single package list entry for a collection provides an environment where protected threads can be used and thread reuse optimized.

For example, a transaction that uses DPS currently could be converted to use packages as follows:

- Bind all of the DBRMs contained in the plans associated with the transaction into a single collection.
- Bind a new plan with a PKLIST containing a single wild-card entry for this collection.
- Modify the resource control table (RCT) entry for this transaction to use the new plan. Protected threads could now be used for this transaction to optimize thread reuse.

Frequently used packages could be bound with RELEASE(DEALLOCATE), with low usage packages bound with RELEASE(COMMIT). This would result in the high-usage packages being retained in the plan's package directory until plan deallocation, while the low usage packages would be removed from the directory, and the space in the EDM pool released.

This approach has two disadvantages. The first is that the use of RELEASE(DEALLOCATE) may require intervention to force the deallocation of high-use packages allocated to long-running threads to allow a package to be rebound. The time to rebind the package will be very much less than the time required to rebind the plan previously.

The second disadvantage in such an approach relates to accounting data. DB2 V2.3 does not report accounting data by package in the SMF 100 and 101 records. Therefore, the use of packages described above would require a change in accounting procedures where the accounting information is currently collected by plan name.

The alternative would be to set up plans that correspond to those currently being used in DPS. This would probably restrict the level of thread reuse possible.

6.1.9 New Administrative Roles

The new administrative authorities provided in DB2 V2.3 include SYSCTRL, PACKADM, and BINDAGENT. Examples of the use of these authorities are given below.

- SYSCTRL

  The SYSCTRL administrative authority is designed to separate administrative functions from ownership and access to data. For packages, this authority can be used to promote collections and packages through the test levels to production.

- BINDAGENT

  The BINDAGENT system privilege again provides separation of the administrative functions from ownership and access to data. This privilege is granted by an owner. When granted by the owner of an application, for example, BINDAGENT includes all package and plan privileges except execute. This authority can be used in any environment but is probably best suited to the production environment in maintaining plans and packages on behalf of the application owners. BINDAGENT can also be used for promotion between levels where the owner at the higher level, and grantor of the authority, has COPY privileges on the packages at the lower level.

- PACKADM

  The PACKADM administrative authority is intended to simplify the management of collections and packages. This authority can be used at all test and production levels.

  For example, an administrator in the development environment could have PACKADM administrative authority over all collections.

  Where a new application is to be developed, a new collection would be created by SYSADM granting PACKADM on the collection to the administrator (UTADMIN):

```
GRANT PACKADM ON COLLECTION NEWAPPL TO UTADMIN;
```

The administrator would grant CREATE IN authority on this collection to the programmers:

```
GRANT CREATE IN COLLECTION NEWAPPL TO PROG1, PROG2, .. :
```

Programmers would need to be granted BINDADD authority from SYSADM to allow them to bind new packages into the collection:

```
GRANT BINDADD TO PROG1, PROG2, .. :
```

When the application is developed and promoted to production, PACKADM could revoke the CREATE IN privileges:

```
REVOKE CREATE IN COLLECTION NEWAPPL FROM PROG1, PROG2, .. :
```

The administrator would retain PACKADM on the collection and, where maintenance of the application was required, grant collection or package privileges to the programmers involved.

Where greater control is required, BINDADD authority could be granted to PACKADM instead of to the programmers. This would allow PACKADM to do the initial bind of the plan and packages and then grant the programmers execute privilege on the plan and all package privileges on the packages involved.

6.1.10 Review of Existing Procedures and Practices

A review of existing procedures and practices should be made to identify those that need to be changed to support package implementation. The extent of the changes to existing procedures will depend on the options chosen for the use of packages. The following list briefly describes the areas where changes may be required:

- Support will have to be included in the program preparation procedures for the new BIND and REBIND PACKAGE commands. The new copy function within package bind allows a package to be rebound into a new collection without the DBRM so there is no longer a need to retain DBRMs. The modification to the program preparation procedures could provide for retention of the DBRM in a temporary data set until the package has been bound.

  It is recommended that DBRM libraries be retained during the conversion of existing plans to packages to allow faster fallback should it be necessary.

- Changes will also be required to the program preparation procedures where the VERSION option is to be used. Ideally, the changes should provide a process for generating the version value that will guarantee uniqueness where required or include the value from an existing version that is to be replaced.

- The procedures for promoting applications between environments will need to account for package bind or rebind and the method used for versioning. If the new VERSION option is used, and we want a different version_id for each environment, promotion will require precompile, compile, and link-edit for each environment. This will generate new

DBRMs to be used in binding the package. Otherwise BIND PACKAGE with the COPY option can be used if the DBRMs are kept.

- Procedures used to identify the plans that should be rebound when a program module is recompiled will need to be modified to identify the collections into which a package must be rebound when a program module is recompiled.
- Procedures used to identify and rebind the plans affected by significant changes to DB2 tables or their indexes will need to be enhanced to perform the same functions for packages.
- Procedures used to provide EXPLAIN information will now need to support packages.

6.1.11 Fallback from Packages

The migration strategy outlined at the beginning of the chapter was intended to ensure that DB2 had been fully tested and was stable before the migration to packages. However, fallback to plans within the same release or to the previous release that does not support packages may be required.

The following actions are recommended to reduce the time involved in fallback should it become necessary:

- Retain the DBRMs generated during precompile. As noted earlier, DBRMs do not need to be retained when packages are used because the COPY option of the BIND command allows a rebind of the package without the DBRM. If they are not retained, all programs would need to be recompiled to obtain the DBRMs required for binding the plans.

  DBRMs generated using the DB2 V2.3 precompiler can be used in DB2 V2.2 if SPE UL90583 has been applied. Retaining the DBRM libraries that were used in DB2 V2.2 prior to migration to DB2 V2.3 may not be satisfactory for fallback because of program changes.

  The release dependencies discussed in Section 6.1.1, “Migrating to DB2 V2.3” on page 139 should be considered in fallback planning where distributed processing using DDF is involved.

- Retain copies of other libraries that may be required for fallback. This would include:

  - BIND commands for the plans
  - Procedures used in previous release
  - JCL for running all production jobs
  - DB2 libraries
  - For CICS, the RCT and DPS exit routines.

- Applications developed since the migration to DB2 V2.3 that use functions such as SET CURRENT PACKAGESET or rely on collections or versions could not be used in the fallback environment without program modification. Where such applications exist, fallback is no longer an option.

6.1.12 EDM Pool Sizing

The size of the EDM pool should be reviewed as part of the migration planning. A number of factors influence the space required by packages compared to that required by DBRMs in this pool. These are:

- Package size

  Space in the EDM pool for packages is allocated in whole pages; that is, the smallest possible package requires one page, whereas a package that is 4.1K requires two pages in the EDM pool.

  This is greater than the space required for the equivalent DBRMs bound into a plan because the DBRMs shared pages in the EDM pool. SQL statements for calculating the size of packages are given in Chapter 7 of the *DB2 Administration Guide*, Volume III. You can also refer to Section 7.2, “Impact on EDM Pool” on page 169 to see how to calculate the space necessary in the EDM pool.

- SKPT reuse

  The skeleton copy of a package will only be present once in the EDM pool regardless of the number of plans that are using the package concurrently. Where DBRMs are used, there will be one skeleton cursor for each plan that is using it concurrently. Therefore, there will be a saving in the EDM pool where DBRMs that are used in many plans are replaced by packages.

- RELEASE options

  DB2 V2.3 provides greater flexibility in controlling when cursors are freed in the EDM pool by allowing the RELEASE option to be specified at the package level. Binding low-usage packages with RELEASE(COMMIT) in plans that were previously bound with RELEASE(DEALLOCATE) could result in reduced space being required in the EDM pool.

The effects of the values specified at both the plan and package levels are shown in Table 13.

Table 13 (Page 1 of 2). Effects of RELEASE Options on EDM Pool Storage

Plan Specification	Package Specification	Impact on EDM Pool
COMMIT	COMMIT or not specified	EDM pool storage for the package is released when the application issues a commit. If all packages default to the plan value, only the plan header will remain in the EDM pool following the commit.[1]
COMMIT	DEALLOCATE	EDM pool storage for the package is not released until the plan is deallocated.
DEALLOCATE	COMMIT	EDM pool storage for the package is released at commit time.[1]

Table 13 (Page 2 of 2). Effects of RELEASE Options on EDM Pool Storage		
Plan Specification	**Package Specification**	**Impact on EDM Pool**
DEALLOCATE	DEALLOCATE or not specified	EDM pool storage for the package is not released until the plan is deallocated.
Note: [1]Where a cursor is defined as WITH HOLD, the EDM pool storage for the package is not freed when the application issues a commit, irrespective of the value specified for the RELEASE option.		

6.2 Modifying Existing Plans to Use Packages

Once the standards are in place and the required changes made to the existing development and migration tools, conversion of plans from DBRMs to packages can begin. A number of approaches can be taken. It is suggested that a pilot application be selected that will exercise all or many of the changes that have been made to the application development environment to support packages.

Programs for the selected application would be precompiled, compiled, and linked at the unit test or equivalent environment. The associated packages would be bound into the collections, which, in turn, would be bound into plans. Once the application has been proved in this environment, it would be promoted through the various test environments to production. Any problems in the preparation, testing, or promotion of programs would be highlighted during the pilot and corrected.

This process would prove that the changes to the procedures and practices are effective and allow general conversion of plans to begin.

6.2.1 Approaches in Converting DBRMs to Packages

The conversion of DBRMs to packages can be achieved through a number of steps that address a group of DBRMs, plans, or applications or through a global change that addresses environments. Of the two, the former is preferred as it allows easier fallback to DBRMs should the conversion fail or adversely affect performance. In addition, although the approach requires a number of iterations to complete the conversion, the packages created in each run can be bound into a single collection to achieve the same result as the global change.

The stepwise conversion could be based on:

- Shared DBRMs

  An analysis of the SYSIBM.SYSDBRM table will provide details of the DBRMs that are used in more than one plan. Converting these DBRMs to packages would provide immediate benefits through the reduced need to rebind plans where one of these common modules needs to be modified.

These DBRMs could probably be further divided into those that are used within an application or application area and those that are used across applications. Converting this latter group would provide potentially the greater benefit in reducing the need for plan rebinds.

Common modules can be handled in two ways when converting the DBRMs to packages. The first way is to include the packages in all collections. If the number of collections is small, the administrative effort required to rebind the package into all of these collections following a modification to one of the modules is small. If the number of collections is large, the effort is correspondingly larger.

The second approach is to bind these modules into a separate collection and include this collection in the package list for all plans. To offset any possible performance degradation, these modules could be modified to set the CURRENT PACKAGESET register to point to the required collection and to reset the value before exiting.

- Plans

  Using plans as the basis for converting from DBRMs to packages may be required where the existing accounting procedures are based on plan reporting. Collections could be established on the basis of one per plan or one per a number of plans. Conversion on this basis would allow easy fallback to DBRMs should it become necessary.

- Transactions

  Using transactions as a basis for converting from DBRMs to packages should be considered for CICS transactions that use DPS. This would remove the restrictions of DPS on the use of protected threads and allow thread reuse for the transaction to be optimized. The SYNCPOINTs could be retained until the ability to fall back to DBRMs is no longer required.

- Applications

  Conversion of an application would probably involve a number of plans with a common owner. In this instance, the DBRMs from all plans in the application could be included in a single collection. One plan containing this collection in the package list could be used for the application. Alternatively, each plan could have this collection in its package list if accounting by plan is required.

- Environment

  The aim of the conversion ultimately should be to have a small number of collections bound into any one plan in a production environment. As an example, consider a CICS production subsystem that is used to support a number of applications, say 10. If these applications were converted so that each had its own collection, one plan containing 1 collection could be set up per application or one plan containing 10 collections could be set up to serve the environment. The latter implementation is better from a CICS/DB2 perspective as it allows greater flexibility in allocating transactions to protected threads and promoting thread reuse.

  However, performance would be adversely affected by the package list processing, as an average of five accesses to SPT01 may be required to

locate a package that is not already in the EDM pool. This could be offset by modifying the application modules to set the CURRENT PACKAGESET register.

Another solution in this case would be to have the DBRMs from all applications bound into a single collection. This would provide better performance in package search while providing greater flexibility in defining the RCT to promote thread reuse. The caveats on this approach are discussed in Section 6.1.3, “Collections” on page 141.

Where reporting at plan level is required, each plan could be bound with a single entry in the package list for this collection.

6.2.2 Converting CICS Applications

As discussed previously, the use of packages solves the problems that DPS addressed. Conversion of the plans for transactions that use DPS to the use of packages allows greater flexibility in setting up the plans needed to support the CICS applications. The choices in the number of plans to be used could include:

- One plan for the application

  This approach gives greatest flexibility in defining the RCT for the application as the transactions involved can be grouped to better utilize protected threads and optimize thread reuse. The steps in converting to this environment are:

  1. Bind all DBRMs for the transactions in the application into packages using a single collection such as COLAPPL1.
  2. Bind a new plan, PLNAPPL1, with a package list consisting of a single entry, COLAPPL1*.

     ```
     BIND PLAN (PLNAPPL1) ..... PKLIST (COLAPPL1.*) ..
     ```

  3. In the RCT, replace the DPS exit program name with the name of this new plan. For example, replace:

     ```
     DSNCRCT TYPE=ENTRY,THRDM=0,PLNPGME=DSNCUEXT,TXID=TRAN1,TWAIT=P
     ```

     with

     ```
     DSNCRCT TYPE=ENTRY,THRDM=0,PLAN=PLNAPPL1,TXID=TRAN1,TWAIT=POOL
     ```

- One plan per transaction

  The steps in using this approach would include:

  1. Bind the DBRMs for groups of transactions or all transactions into packages. The collections to be used could be based on the transactions being converted, such a TRAN1, or on the application, as above. The latter approach is preferable because creating and maintaining collections on a transaction basis would require greater administrative effort particularly if there are many common routines.

2. Bind a new plan for each transaction with a package list referring to a single wild-card package list entry that would depend on the approach taken: 'TRAN1.*' if the collections were based on transactions, or COLAPPL1' if a single collection was set up for all transactions:

```
BIND PLAN (TRAN1) .... PKLIST (TRAN1.*) ...
```

or

```
BIND PLAN (TRAN1) .... PKLIST (COLAPPL1.*) ...
```

3. Modify the RCT to replace the DPS exit with the plan names associated with the transactions and reassemble.

This approach is preferable when using DB2 V2.3 if accounting data by plan name is required. Using one plan for all transactions would provide accounting data by transaction ID but not by the plan names previously associated with DPS.

A similar approach can be taken for converting all CICS applications whether DPS is used or not.

Using DPS with Packages

Users who want to continue to use DPS when plans have been converted to packages should be aware of two recent APARs that affect all DB2 CICS Attachment Facility users when DPS is being used. The two APARs are PN20663 and PN14955.

PN20663 is a documentation change to *IBM DATABASE 2 Version 2 Administration Guide*, Volume III. The change describes conditions under which a thread will not become reusable at intermediate SYNCPOINTs, that is, SYNCPOINTs taken before end of task.

The documentation change that affects DPS users is that threads will not become reusable at intermediate SYNCPOINTs if any of the special registers that maintain their value across a unit of recovery are not at the initial state value. These special registers include CURRENT SQLID, CURRENT SERVER, and CURRENT PACKAGESET.

For DPS users, if the value of the CURRENT PACKAGESET register, for example, is changed during processing and not reset to the initial state before the SYNCPOINT is taken, the following will occur:

- The thread will not be released by the CICS Attachment Facility.
- Thread reuse will not occur as the task will continue to use this thread with the same plan allocated.
- DPS will not occur because the same thread and plan will be used; that is, the DPS exit will not be taken on the first SQL statement issued following the SYNCPOINT.

Therefore care should be taken to ensure that, where a special register is changed while processing with DPS, the register is reset to its initial value before the SYNCPOINT is taken.

6.2.3 SCLM and Packages

Users of Software Configuration and Library Manager (SCLM), especially where it is used in conjunction with Cross System Product (CSP), should be aware of a new redbook, *AD/Cycle Library Integration: SCLM V3.4 and CSP/370AD V4.1*.

Chapter 10, "SCLM and DB2 Packages," in that book describes a sample solution for the use of packages in an SCLM/CSP 4.1 environment. The sample relates to a specific development environment based on the following assumptions:

- Only one plan will be required for all applications in a particular CSP 4.1 target environment.
- Each logical SCLM level will have exclusive use of a DB2 database.
- The same DB2 subsystem will be used for both development and production environments.
- Collections will be used to support multiple versions of packages. The VERSION option of the precompiler will not be used.

Installations that use SCLM and are interested in migrating to packages may find Chapter 10 in the new Redbook useful.

6.3 Getting Started with Packages

This section discusses the steps that could be taken in developing a strategy for using packages. It is intended for those customers who have not developed a plan for migrating to packages but would like to gain experience in their use so they can better define the role that packages will play in future application development.

The activities involved are discussed briefly below.

1. Education

   The starting point has to be courses in package bind and the new authorizations and options available that apply to it. As a minimum, staff from database administration, development, and operations should attend those courses so that the impact of using packages in these areas can be better assessed.

2. Develop guidelines for using packages

   The guidelines developed at this point should consider all areas that the use of packages is likely to impact. The factors that should be considered include:

   - The scope of collections that will be used in the test and production environments and the naming standards to be used.
   - Whether the new precompile VERSION option will be used. If so, the values that will be assigned, the method for generating the value, the number of versions that will be supported in the different

environments, and the manner of enforcing these limits should all be documented.

- The method that will be used for promoting between environments.
- The authorizations that will be used to control package bind and use in the different environments and, in particular, the use of the new SYSCTRL, BINDAGENT, and PACKADM authorities.

3. Review existing procedures

   When the guidelines have been developed, the existing procedures should be reviewed to identify those that will need to be changed to support packages. The procedures that need to be changed normally include those used in program preparation, promotion, plan binding and rebinding, and EXPLAIN processing.

4. Choose a pilot project

   Ideally, a pilot project should exercise all the areas of change where new procedures need to be validated, and it should be relatively short in duration. For the pilot, either a small new application, conversion of an existing application, or conversion of the common modules to use packages may be suitable.

   Where the pilot is based on conversion of existing plans to use packages, the programs involved would have to go through the program preparation process as if they were new programs. Simply rebinding the DBRMs as packages would not provide the testing required for the new procedures. Fallback planning also should be included where conversion of existing applications or plans is chosen as the pilot.

5. Modify existing procedures

   The procedures identified above as needing changes to support packages should be modified as required.

6. Educate staff involved in the project

   Education of the staff that will participate in the project can be run in parallel with the modification of procedures. A course should be developed that covers not only package bind and the new options and authorities that relate to it but also the guidelines and procedures that are being developed to support its use in the installation. This course should form the basis of the education that will be given to all development staff following successful implementation of the pilot. Modified versions of the course could be prepared for other functions such as operations.

7. Develop the pilot

   The pilot chosen should now be implemented at the unit test level and promoted through the other test levels to production. Any inadequacies in the guidelines, procedures, or education of staff that are revealed during the pilot should be noted for later review.

8. Review pilot results

   A formal review of the results of the pilot should be held shortly after its completion. All staff involved in the pilot should attend. The results

and conclusions of the pilot can then be documented with special attention paid to any problems that were experienced.

9. Amend guidelines, procedures, and education as required

   Amendments to the guidelines, procedures, and education material should be amended where necessary in light of the experiences of the pilot.

10. Education of all staff

    Education should now be given to all staff whose function will be affected by the use of packages.

6.4 Package Administration

This section discusses the additional administrative requirements involved in using packages. Details are also provided on any IBM or third-party tools that were announced or available at the time of writing that address these requirements.

The topics discussed include:

- Administrative requirements
- Enhancements to existing procedures
- Availability of tools.

6.4.1 Administrative Requirements

There is great flexibility in the manner in which packages can be implemented. The approach taken can influence the administrative effort required to manage both the new DB2 objects and the new administrative authorities and privileges related to them. This section discusses the possible impact of using packages on the administrative functions. The discussion is limited to the new activities that may be necessary.

Packages and Collections

Effective control of packages and collections through their related authorities and privileges is essential to the successful use of packages. The use of packages increases the number of DB2 objects that must have privileges granted for such activities as bind, rebind, copy, and execute.

The way in which collections are used to group packages also impacts the administrative effort. For example, having collections at the application level may facilitate promotion of applications and package maintenance, but this may be offset by an increased administrative effort required to manage and control the collections. Conversely, having one collection per test or production environment reduces the effort required to control collections but may increase it for other activities, such as application promotion and package maintenance.

The increased options available for both package and plan bind may also increase the administrative requirements. For example, the ability to specify parameters such as OWNER, QUALIFIER, ISOLATION, VALIDATE, and RELEASE at package bind time provides greater flexibility in the design and operation of applications. The administrative effort required to maintain these parameters in the various test and production environments and change them for promotion between the environments will be greater than for the equivalent activities for plans in previous releases.

Versions

Controlling the number and currency of package versions is another new administrative activity in DB2 V2.3. This control of the number of versions of a package that are to be retained is necessary to avoid uncontrolled growth in the catalog and directory tablespaces.

If versioning is used, the format of the value specified for the VERSION parameter at precompile time is important. The use of AUTO, either alone or in combination with other values, ensures that all version_ids are unique and so does not limit the number of versions that can exist. The use of AUTO would also assist in identifying the older versions of a package because the value generated is a time stamp. Freeing individual package versions requires that the version_id be fully specified; in case of AUTO, a 26-character value must be entered accurately. Managing the number of versions in the different environments requires greater administrative effort if tools are not available to perform the task.

6.4.2 Authorizations

The additional authorizations needed to support the use of packages vary from the authorizations programmers need to develop and maintain applications using packages in the test environment to those owners and binders required to promote and run the applications in other test and production environments. The choices made for implementing packages in these test and production environments determine to a large extent the administrative effort required.

Package Maintenance

As discussed previously, the number of packages associated with an application are far greater than the number of plans associated with the application. Where significant changes to the DB2 environment occur for an application, a rebind may be needed to reflect those changes in the access paths being used. Such changes could include addition of new indexes or modifications of existing indexes or significantly altered DB2 statistics following table maintenance and the running of the RUNSTATS utility.

Where packages are not used, the number of plans involved is usually small, and it is relatively simple to identify them. For example, a simple query against the SYSIBM.SYSPLANDEP catalog table would identify the plans that are dependent on the tables whose statistics or environment has changed. Although the resulting rebind of these plans may be resource intensive and result in a catalog contention, the administrative effort is minor.

Where packages are used, the number of control structures to be identified and rebound is much greater. In addition, where a plan in previous releases had only one owner and one value for parameters such as VALIDATE and RELEASE, the values for these parameters can be different for different packages.

The alternatives are either to identify and rebind the packages that are affected individually or to rebind all of the packages. As the number of packages involved may run into the hundreds or even thousands for a large application, identifying and binding the affected packages would require significantly greater administrative effort. Reference to the SYSIBM.SYSPACKDEP table will provide a list of the packages that may be affected by the changes, and the SYSIBM.SYSPACKAGE table will provide the parameters to be used on the rebind.

Rebind of all the packages would require much greater machine resource and may result in unexpected and unwanted changes to the access paths in some packages that are unrelated to the changes that prompted the rebind.

6.4.3 Additional Administrative Requirements

Section 6.1.10, “Review of Existing Procedures and Practices” on page 148 discusses the modifications that will be needed to existing procedures to support packages. However, there is also a requirement for a whole new range of tools to assist in the management of packages. This section discusses a number of tools that would assist in the administration of packages:

- Converting plans from DBRMs to packages

  There will be a need for some time for tools that assist in the conversion of plans from DBRMs to packages. The function provided should allow selective conversion based on, for example, plan names, application names, or environment. The assumption here is that the DBRM name will include an indication of the application to which it belongs.

  The tool could automatically generate the commands to both bind the packages from user-specified selection criteria into one or more collections and to bind the plan containing these collections in the package list.

- Version control

  The problems associated with version control are discussed previously. What is required to assist in versioning control is a tool that:

  - Displays all versions of a package within a collection or across all collections
  - Allows freeing of a package from this list based on screen selection
  - Allows a limit to be set on the number of versions that can exist in a collection with an option to specify selection criteria for the package to be freed when the addition of a new version exceeds the limit. The limit should be variable by collection or environment.

- Identifying version_id

  There are problems at present in identifying the version_id of the package that is selected to run with a load module where several versions for the package_id exist within the same collection. Although DB2PM provides package support with DB2 V3, the package qualifiers that will be reported include location name, collection_id, and package_id only. A tool that could identify the package that would be invoked for a particular load library member would be useful in analyzing package-not-found conditions (-805 SQL codes).

- Assist promotion between environments

  As a minimum, this tool would generate the commands to bind both the packages and plans when an application is promoted. The commands generated could be based on defaults for the bind parameters and would be directed to a data set to allow editing of the commands before execution.

- Package maintenance

  The problems associated with package maintenance are discussed in Section “Packages and Collections” on page 157. A tool that reports on the packages that are affected by a change to DB2 tables or indexes with an option to automatically bind all or selected packages from this list would be a useful administrative tool.

- Package move

  This tool would enable a package to be moved from one collection to another and would involve a BIND with COPY option followed by a FREE PACKAGE command. A further refinement would be the ability to do the copy without generating a new access path. This could avoid possible performance problems where the access path for a package is changed when it is copied to another collection for fallback purposes and a fallback occurs.

- EXPLAIN compare

  A tool that compares the access paths for a package that has been bound into different collections, as in the fallback situation above, and highlights the differences would help avoid unexpected performance problems.

- Redundant packages

  A tool that would list redundant packages, that is, packages that have identical package_ids, version_ids, and consistency tokens and that are bound into two or more collections, could assist in reducing the number of packages held in the catalog and directory.

  There may be reasons for having redundant packages. For example, common modules may be stored in a number of collections for performance reasons, rather than in a separate collection that is in the package list for every plan.

The above list is not intended to be an exhaustive list of requirements; rather, it indicates the types of tools that would assist in administering packages.

6.4.4 Products That Assist in Package Administration

This section contains brief descriptions of products that are available from vendors to assist in package management. The purpose is to advise on the products that had been announced or were available at the time this book was written. Details are included only for the functions that address package management. The details were provided by the vendors and were current at the time.

This section does not compare the different products; those interested in any of the products should check with the vendor to obtain the latest product information.

It is the user's responsibility to assess the applicability of any of these products to his or her own environment.

PLATINUM Package/It** Version 1.1

PLATINUM Package/It helps automate the conversion from DBRMs to packages and the management of DB2 packages and collections.

PLATINUM Package/It is designed to analyze existing plans to help determine the benefits of converting to packages and then automate all or some of the conversion process. It can then provide ongoing maintenance of package versions and automated deletion of unwanted versions of packages.

The product has four major components: Pre-conversion Services, Conversion Services, Version Management, and Load Module Services.

1. Pre-conversion Services

   This component is designed to help users determine the most efficient use of packages in their DB2 subsystems through an analysis of DBRM usage. Reports on plans that share DBRMs and on DBRMs that have multiple versions can be generated.

   This component also supports definition of collections to which the DBRMs identified in the above analysis can be assigned based on the above analysis. The BIND parameters for the collection can also be assigned at this time.

2. Conversion Services

   This component provides four levels of automated conversion from DBRMs to packages:

   - Global Package Conversion converts all DBRMs within the specified plans into packages.
   - Shared Package Conversion converts only DBRMs that are shared by the specified plans into packages.
   - Selective Package Conversion converts only those DBRMs that match user-specified selection criteria.
   - Simple Conversion is a simplified and quick way of converting DBRMs into packages. No customization is possible with this function, which takes default options and automatically converts DBRMs into packages.

Simple Unconversion allows fallback to the old plan (without packages) if required.

All BIND parameters can be specified and executed in either batch or online mode.

3. Package Management Services

   This component provides three methods of maintaining packages and package versions:

   - The online Version Deletion facility allows users to display all versions for specified collections and package_ids and to delete selected versions immediately or have the commands written to a data set for later execution.
   - The Version Retention facility allows users to specify the rules for retaining and deleting versions. The retention definition is defined as a number of versions to be retained or as the period for which versions should be retained. These definitions can be assigned to a grouped set of collections or packages to control version deletion during scheduled maintenance runs.
   - The Package Usage Analysis facility allows users to analyze how their site is making use of packages and alerts them about certain misuses. The Package Evolution, Package Redundancy, and Package Orphan analysis reports can be run in online or batch mode.

4. Load Module Services

   This component generates the required BIND commands for a specific set of load modules. The Load Module Mapping report shows which packages are being used within the specified load modules. This information is critical for users who need to know which packages are being placed or replaced in production when moving load modules into production environments.

PLATINUM Plan Analyzer** Version 1.5

PLATINUM Plan Analyzer (PPA) is a plan and package analysis, management, and monitoring tool. It has three main components: Explain Services, Reporting Services, and Management Services.

1. Explain Services

   This component provides the following facilities, which expand on DB2 EXPLAIN processing:

   - Enhanced EXPLAIN output provides information on access path choices including the summary, relative cost, object tree, catalog statistics, object dependency, access path, SQL statement, and predicate analysis. Current type EXPLAIN determines the current access paths. Future type EXPLAIN predicts the impact of a BIND without actually executing it.
   - SQL statement and physical object design facility gives recommendations on how to resolve design problems.
   - Grouping facilities for plans, packages, DBRMs, and SQL statements help in explaining the history of an application.

- Enhanced EXPLAIN outputs can be saved through EXPLAIN Strategy Services for later version comparison.
- EXPLAIN on multiple subsystems allows you to tune applications on the test system without migrating the plan or package to the production system.
- PPA's Catalog Update facility enables you to manipulate catalog statistics for any EXPLAIN processing. The catalog updates can be committed or rolled back.

2. Reporting Services

   This component provides reporting facilities that include information on plans, DBRMs, packages, objects, SQL statements, authorizations, and dependencies. The Identify Problem function allows for specific searches in the historical database. The reports support DB2 remote locations as well.

3. Management Services

   This component enables you to BIND, REBIND, and FREE plans or packages in online or batch mode.

Candle Corporation !DB/EXPLAIN** Version 230

Version 230 of !DB/EXPLAIN provides support for DB2 V2.3 and allows the benefits of such enhancements as packages, collections, and distributed processing to be exploited.

!DB/EXPLAIN provides easy access to 55 fields in the four new DB2 catalog tables that support packages and displays this data on user-tailored screens.

Some of the specific functions that support packages include:

- Conversion of DBRMs to packages
- Display of all DB2 data related to packages and collections
- BIND, REBIND, FREE a package. Global changes to BIND parameters can be made for more than one package.
- Display of EXPLAIN output in an easy-to-understand format for packages. Remote EXPLAIN is also provided on any connected DB2 subsystem.
- “What if” analysis for packages to assess the effects of changes to catalog statistics on access path selection
- Display and comparison of package history statistics including access path, cost, and package attributes
- Display of package list, packages, collections, and connections associated with a plan
- Display of tables associated with a package
- Display of collections and locations for all packages

- Comparison of the EXPLAIN output for different packages and display of the differences in access paths
- Display of all environments in which a package can be executed
- Display of all environments in which a plan can be executed

Chapter 7. Performance Implications

The impact on performance that packages may have is a great concern, especially when you are migrating DBRMs to packages. This chapter discusses the performance implications of package list search, the impact of packages on the storage used in the EDM pool, and performance monitoring.

7.1 Package Search

When implementing packages, the way in which you organize your packages, collections, and plans can have an impact on the performance of package search. This section discusses package list organization, the number of packages to be grouped in one collection or plan, and the difference between packages and DBRMs directly bound into a plan.

7.1.1 Package List Organization

The performance of a plan can be negatively affected by poor package list organization. Thus, it is critical that development and system administration personnel become familiar with the various ways of specifying packages in a package list and understand the package search algorithm.

When constructing a package list, the following factors may affect the performance of the plan:

- The number of package entries included in the package list
- The order of the package entries specified
- The use of CURRENT PACKAGESET register
- The type of package entry.

The order and number of package entries in the package list are especially important if the program does not set the CURRENT PACKAGESET special register. When the CURRENT PACKAGESET register is blank, DB2 searches for the appropriate package through the package list starting from the first entry as described in detail in the Section 2.2, “Package List Search” on page 32.

The number of package list entries should be minimized, because this number determines the number of searches. There will be some performance impact when there are more than 60 package list entries. Thus, it is better to group packages into the minimum number of collections and specify the package_id with an asterisk in the package list. For example, compare the following two plans and package lists:

```
PLAN1  :  PKLIST(COL1.*)
PLAN2  :  PKLIST(COL1.PROG1, COL1.PROG2, ... COL1.PROG80)
```

For package COL1.PROG1, there is no difference in package search performance between the two plans because DB2 can get the correct

package with the first entry search in either package list. However, for package COL1.PROG80, in PLAN2, DB2 needs to go through all other entries just to find out that they do not match, whereas, in PLAN1, DB2 only needs to search the first entry to find COL1.PROG80.

You can probably see that when there are a number of package entries, the order of the package entries matters most. The order of package entries generally should be based on usage frequency. As the search is linear, the frequently used package should be at the front of the list. If you have both a specific package and a collection_id.* in your package list, both of which are used frequently, you might want to put the specific package upfront. Specifying an asterisk for the package_id does not alter performance for a specific entry if DB2 finds the appropriate package within the collection for which it was searching. For example, if DB2 is searching for the package COL1.PROG1, there is no performance difference between the two plans with the following package list structure:

```
PLAN3  :  PKLIST(COL1.PROG1, COL2.*)
PLAN4  :  PKLIST(COL1.*, COL2.*)
```

However, when DB2 cannot find the required package, DB2 can more easily recognize that the package is missing if the entry for the package is a fully qualified entry. For example, if DB2 is searching for PROG2, which actually is bound in collection COL2, the package list in PLAN3 is better in performance than that in PLAN4 because DB2 can tell that the first entry, COL1.PROG1, does not match by just looking at the package list.

The above discussion leads us to advise you to exercise caution when specifying a number of collection_id.* entries in the package list. Compare the following two package lists:

```
PLAN5  :  PKLIST(COL1.PROG1, COL2.PROG2, COL3.PROG3, COL4.PROG4,
                 COL5.PROG5, COL6.PROG6, COL7.PROG7, COL8.PROG8)

PLAN6  :  PKLIST(COL1.*, COL2.*, COL3.*, COL4.*, COL5.*, COL6.*,
                                          COL7.*, COL8.*)
```

With PLAN5, finding an entry would require a memory search through the list, but with PLAN6, there could be potentially many index searches of SPT01 to find or not to find the package in a collection. Setting the CURRENT PACKAGESET register would resolve the problem PLAN6 potentially has.

When the CURRENT PACKAGESET register is set, the number and order of the package entries are not important, because DB2 directly determines whether the package is included in the package list by deriving the collection_id value from the CURRENT PACKAGESET register and the package_id value from the executing program. Refer to the PLAN6 above, for example. There is no difference between

```
SET CURRENT PACKAGESET = 'COL1'
       and executing a package in COL1
```

and

```
SET CURRENT PACKAGESET = 'COL8'
       and executing a package in COL8.
```

If the CURRENT PACKAGESET register had not been set, searching a package in COL8 would definitely be more time consuming compared to searching a package in COL1. So, if there is a number of collection_id.* type of entries in the package list, consider using the SET CURRENT PACKAGESET statement in the program.

The type of package entry specification also has an impact on performance. There are two types of package entries in general: a package qualified with a specific three-part name (for example, location1.collection1.package1), and a package with an asterisk for any part of the three-part name (for example, location1.*.*). We have already discussed how specifying an asterisk in the package_id could affect search performance compared to the specific entry. We now describe other considerations.

DB2 checks whether the plan owner has the EXECUTE privilege on packages included in the package list. Using an asterisk in the location or collection qualifier of the package entry will force the plan to go through authorization checking at run time, which is not desirable from a performance point of view. The cases that force authid checking at execution time are as follows:

- An asterisk is specified for the collection_id field (for example, PKLIST(*.PACKA))
- An asterisk or a remote location_name is specified for the location_name field (for example, PKLIST(*.COL1.PACKA) or PKLIST(remote-location.COL1.PACKA)). EXECUTE privileges for remote packages cannot be validated at bind plan time.
- An asterisk is specified for the package_id field, and the plan owner has not been granted EXECUTE privileges for the entire collection before binding the plan. VALIDATE(BIND) will result in an error in this case. An asterisk in the place of package_id invokes authid checking at bind plan time whether you bind with VALIDATE(BIND) or VALIDATE(RUN). With VALIDATE(RUN), if the authorization check fails at bind time, it will be postponed until execution time. For example, if your BIND PLAN subcommand is:

  ```
  BIND PLAN(PLAN1) PKLIST(COL1.*) OWNER(OWNER1) VALIDATE(RUN) .....
  ```

  to avoid authid checking at run time, the owner of this plan, OWNER1, must have EXECUTE privilege on the entire collection COL1.* before binding the plan, such as by the following statement:

  ```
  GRANT EXECUTE ON PACKAGE COL1.* TO OWNER1
  ```

  For the plan owner to have the EXECUTE privilege on the existing packages in the specified collection before binding a plan may be sufficient to allow users to execute the application; however, it is not sufficient to have authorization checking validated at bind time.

Needless to say, even the fully qualified package list entry goes through authid checking at run time if its EXECUTE privilege was not granted to the plan owner before the plan bind. Authorization checking should be avoided

at execution time whenever possible, but there is no column in the DB2 catalog to indicate whether or not authorization checking will be done at execution time, so users should be aware of the above cases.

Summarizing what we have discussed so far:

- Minimize the number of entries in the package list.
  - Be especially careful not to list too many entires with an asterisk for the package_id without setting the CURRENT PACKAGESET register.
- Order the package list entry according to the frequency of package usage.
- List fully qualified entries before entries with an asterisk.
- Note whether or not the program sets the CURRENT PACKAGESET register.
- Avoid using an asterisk for location_name and collection_id in the package list entry.
- Grant the required EXECUTE privilege of packages to the plan owner before binding the plan.

In general, bear in mind that a very long package list or many collection_id.*s in the package list can have an adverse influence on performance.

7.1.2 Number of Packages in a Collection and Plan

The number of packages to be included in a collection does not influence performance because DB2 uses an index to locate the package. If you have 1000 packages and bind them into one collection, performance is not a problem as long as you bind the plan structuring the package list as PKLIST(COL.*). Listing all 1000 specific package list entries as PKLIST(COL.PACK1, COL.PACK2, COL.PACK3, ...) will cause a performance problem. So, it is the order and number of package list entries that affect the performance most, not the number of packages included in one collection. However, remember that the more packages you include in a collection, the less granular the security control can be.

The use of packages may lead to using one big plan with many packages, possibly even one plan for the entire system, because modification of a program no longer requires binding a plan. Performance is not affected by the number of packages included in one plan. However, you must take into consideration security issues and the use of EDM pool storage when planning such a big plan. A plan is a unit to control execution authorization for an application. In other words, having only one plan for the entire DB2 subsystem means that all the users can execute all the packages, which is probably not desirable in many cases. Such a big plan may also seriously impact EDM pool storage if the packages are not bound with RELEASE(COMMIT). Also note that in DB2 V2.3, accounting information can only be reported based on plans and not packages. (See Section 7.3.2, “Performance” on page 171 for more information.) Keep the above considerations in mind when binding a big plan with many packages.

7.1.3 Packages versus DBRMs

Allocating packages may require slightly more system overhead compared to DBRMs directly bound to a plan. This is why it is important to consider package list organization when migrating DBRMs to packages. Performance may be adversely affected if you make a long list of package entries.

7.2 Impact on EDM Pool

In the EDM pool, packages work similarly to plans, as we have seen in Section 2.1, “EDM Pool Processing” on page 25. Using packages does not change the mechanism of creating for each user an application copy from the skeleton copy. Compared to DBRMs directly bound into a plan, EDM pool storage will be saved if some packages are bound with RELEASE(COMMIT) in a plan that were previously bound with RELEASE(DEALLOCATE). Storage will also be saved if a DBRM is used concurrently by different plans, because with packages, only one SKPT is allocated in the EDM pool regardless of whether the package is allocated to many plans. However, in other environments, using packages may require more storage space. Plans and packages use storage space in the EDM pool in increments of 4K (so even if you need only 2K, you actually use 4K). One DBRM requires at least 4K of storage when using packages, whereas before packages, the sum of all the DBRMs bound into a plan used storage space in 4K blocks. For example, if two DBRMs, each 1K in size, were bound directly into a plan, they would require only one 4K block of EDM pool storage to hold both DBRMs, whereas if they were bound as two packages and both are allocated to a plan, they would require two 4K blocks. Note that this is an example of the worst case.

After the migration from plans to packages, it is generally recommended that the size of the EDM pool be maintained initially, and the storage size adjusted afterward using performance monitoring tools such as DB2PM.

You can get information on the package size from the DB2 catalog using the following SQL statements:

```
SELECT COLLID,NAME,VERSION,PKSIZE,AVGSIZE
FROM SYSIBM.SYSPACKAGE
ORDER BY COLLID,NAME   ;

SELECT COLLID,NAME,MAX(SECTNO)
FROM SYSIBM.SYSPACKSTMT
GROUP BY COLLID,NAME
ORDER BY COLLID,NAME   ;
```

With this information, you can figure out the space necessary in the EDM pool for the specific package in the following way:

```
celling(4K,((AVGSIZE * MAX(SECTNO)) + PKSIZE)) * NUMBER OF PACKAGES
```

NUMBER OF PACKAGES is the sum of the number of users concurrently allocating the package, plus one for SKPT. And remember that packages take space in the EDM pool in increments of 4K. For example, if the

AVGSIZE is 668 bytes, MAX(SECTNO) is 2, PKSIZE is 1096 bytes, and three users are concurrently using this package, the space this package takes in the EDM pool can be estimated as 16K:

```
(668 * 2 + 1096) * (3 + 1) = 2432 * 4
                             |
                             4K  * 4 = 16K
```

The bind package RELEASE option can have an impact on EDM pool usage as you can see in Table 13 on page 150. Because there is no ACQUIRE option for packages, the resources used in the statements in a package are acquired at the time they are first used just as if ACQUIRE(USE) had been specified. The RELEASE option determines when the copy of the package made in the EDM pool is freed. (However, if any cursor is held at the time a commit is issued, the copy is not freed.) In general, RELEASE(COMMIT) would be a better choice unless you want to hold the locks across a commit point, or the package is used frequently within a plan. Recall that a plan can use a mixture of packages that have different RELEASE options.

7.3 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.

7.3.1 Access Path

BIND PACKAGE supports the EXPLAIN option just as BIND PLAN did to monitor the access path selected by DB2. DB2 inserts the information into the package_owner.PLAN_TABLE. Two columns are added to the PLAN_TABLE to support the package function: VERSION and COLLID. Figure 67 shows the sample EXPLAIN output of the program we used in the scenarios in Chapter 3, “Using Collections” on page 43 and Chapter 4, “Using the VERSION Keyword” on page 69.

```
QUERY|QBLOCK |PROG   |      |ACCESS|      |      |
  NO|   NO |NAME   | ..... |  TYPE| ..... |VERSION|COLLID
----- ------- ------- ------- ------ ------- ------- ------
 184 |   1   |PARTINQ|      |   I  |      |  VER2 | COL1
 214 |   1   |PARTINQ|      |   I  |      |  VER2 | COL1


                                        |     |
                                       New columns
```

Figure 67. PLAN_TABLE

7.3.2 Performance

Accounting information for packages is not available in DB2 V2.3 but will be in DB2 V3. Along with the DB2 V3 enhancement for package accounting support, performance analysis tools are enhanced to support packages as well.

DB2PM V3

DB2PM V3 along with DB2 V3 will have package accounting support. Two DB2PM identifiers, MAINPACK and PACKAGE, have been introduced for this purpose:

- MAINPACK can be used to distinguish plans according to the packages they contain. The representative package is either the first or last package or DBRM in a plan. This identifier is useful when the name of a plan does not provide satisfactory identification, for example, reporting DBATs initiated by remote requesters that all have the same DISTSERV plan name at the server site.
- PACKAGE is used to identify a package regardless of the plan to which it belongs. When usage is reported on a per package basis, it is not possible to attribute activity to specific plans or other DB2PM identifiers.

Highlights of the additional new functions are:

- Accounting reports and traces will have a repeating group of fields for each package or DBRM that was referenced during the duration of the accounting record (see Figure 68 on page 172).
- INCLUDE/EXCLUDE will be expanded to support specific processing for packages or DBRMs.
- ORDER will be expanded to support the ordering of accounting reports by PACKAGE (see Figure 69 on page 173) and MAINPACK (see Figure 71 on page 174).
- The SQL activity trace and report can be summarized by program, and the program can be either a DBRM or a package.
- Record trace provides all package-related data captured by the DB2 instrumentation facility.
- Exception processing (batch and online) will support the new package fields and allow qualification of exceptions by package name.
- Online monitor thread displays will contain the new package fields for the current package.

Figure 68 on page 172 shows a sample of an accounting short trace for data requested by an OS/2 requester from a DB2 server. As can be seen from this trace, two packages are involved, and the plan name at the server site is DISTSERV.

```
1
0 =============================================================
0              ACCOUNTING
0                TRACE
0                  LAYOUT(SHORT)
1  LOCATION: D31B7                            DB2 PERFORMANCE MONITOR (V3)                          PAGE:
1-1
   SUBSYSTEM: D31G                             ACCOUNTING TRACE - SHORT                       DB2 VERSION: V3
 ACTUAL FROM: 09/08/93 04:11:42.31                                                         REQUESTED FROM: ALL
04:11:00.00
   PAGE DATE: 09/08/93                                                                             TO: DATES
04:15:00.00

 PRIMAUTH CORRNAME CONNECT  ACCT TIMESTAMP COMMITS  OPENS UPDATES INSERTS EL. TIME(CL1) EL. TIME(CL2) GETPAGES SYN.READ LOCK SUS
 PLANNAME CORRNMBR THR.TYPE TERM. CONDITION SELECTS FETCHES DELETES PREPARE TCB TIME(CL1) TCB TIME(CL2) BUF.UPDT TOT.PREF LOCKOUTS
 -------- -------- -------- -------------- ------- ------- ------- ------- ------------- ------------- -------- -------- --------

 USRT001  004C0001 SERVER  04:11:42.312608      2      1      0      0      5.974271      0.120076      12      10       1
 DISTSERV 'BLANK'  DBAT    NORM DEALLOC         0     65      0      0      0.027459      0.024048       0       1       0

    ---------------------------------------------------------------------------------------------
    |PROGRAM NAME       TYPE     SQLSTMT  CL7 ELAP.TIME   CL7 TCB TIME  CL8 SUSP.TIME  CL8 SUSP|
    |SSQLFFO           PACKAGE       68      0.119931      0.023910      0.090771       8|
    ---------------------------------------------------------------------------------------------

    ---------------------------------------------------------------------------
    |REQUESTER       METH   TRANS  ROLLBCK  COMMITS  SQLRECV  ROWSENT  CONVI|
    |<WSD2194A>      APPL      1       0        2       6       64      1|
    ---------------------------------------------------------------------------

 USRT001  004D0001 SERVER  04:12:11.647066      2      0      0     64      5.872581      0.664971     151      18       1
 DISTSERV 'BLANK'  DBAT    NORM DEALLOC         0      0      0      0      0.157637      0.103878     197       0       0

    ---------------------------------------------------------------------------------------------
    |PROGRAM NAME       TYPE     SQLSTMT  CL7 ELAP.TIME   CL7 TCB TIME  CL8 SUSP.TIME  CL8 SUSP|
    |SSQL              PACKAGE       64      0.664814      0.103727      0.554440      18|
    ---------------------------------------------------------------------------------------------

    ---------------------------------------------------------------------------
    |REQUESTER       METH   TRANS  ROLLBCK  COMMITS  SQLRECV  ROWSENT  CONVI|
    |<WSD2194A>      APPL      1       0        2      64        0      1|
    ---------------------------------------------------------------------------

 USRT001  004E0001 SERVER  04:12:40.832168      2      1     64      0      7.827162      0.189291      17       3       1
 DISTSERV 'BLANK'  DBAT    NORM DEALLOC         0     65      0      0      0.140760      0.077407      64       1       0

    ---------------------------------------------------------------------------------------------
    |PROGRAM NAME       TYPE     SQLSTMT  CL7 ELAP.TIME   CL7 TCB TIME  CL8 SUSP.TIME  CL8 SUSP|
    |SSQL              PACKAGE      132      0.189144      0.077266      0.110461       4|
    ---------------------------------------------------------------------------------------------

    ---------------------------------------------------------------------------
    |REQUESTER       METH   TRANS  ROLLBCK  COMMITS  SQLRECV  ROWSENT  CONVI|
    |<WSD2194A>      APPL      1       0        2     130       64      1|
    ---------------------------------------------------------------------------

 USRT001  004F0001 SERVER  04:13:12.425355      2      1      0      0      8.651578      0.222817     342       3       1
 DISTSERV 'BLANK'  DBAT    NORM DEALLOC         0     65     64      0      0.141431      0.098112     197       1       0

    ---------------------------------------------------------------------------------------------
    |PROGRAM NAME       TYPE     SQLSTMT  CL7 ELAP.TIME   CL7 TCB TIME  CL8 SUSP.TIME  CL8 SUSP|
    |SSQL              PACKAGE      132      0.222658      0.097969      0.123500       4|
    ---------------------------------------------------------------------------------------------

    ---------------------------------------------------------------------------
    |REQUESTER       METH   TRANS  ROLLBCK  COMMITS  SQLRECV  ROWSENT  CONVI|
    |<WSD2194A>      APPL      1       0        2     130       64      1|
    ---------------------------------------------------------------------------


 ACCOUNTING TRACE COMPLETE
```

Figure 68. DB2PM Accounting Short Trace

Figure 69 shows the accounting short report ordered by package. The report indicates the use of resources on a per package/DBRM basis, regardless of the plan under which a particular package is executed.

```
1
0 ============================================================
0              ACCOUNTING
0                REPORT
0                  LAYOUT(SHORT)
0                  ORDER(PACKAGE)
1    LOCATION: D31B7                        DB2 PERFORMANCE MONITOR (V3)                        PAGE:
1-1
     SUBSYSTEM: D31G                          ACCOUNTING REPORT - SHORT                  DB2 VERSION: V3
 INTERVAL FROM: 09/08/93 04:11:42.31                                                  REQUESTED FROM: NOT SPECIFIED
          TO: 09/08/93 04:13:12.42                 ORDER: PACKAGE                          TO: NOT SPECIFIED

                                                                 TYPE          SQLSTMT   CL7 TCB TIME  CL8 SUSP
 PACKAGE                                                          #OCCURS  CL7 ELAP.TIME  CL8 SUSP.TIME
 ------------------------------------------------------------  --------  -------------  -------------  --------

 D31B7.USRT001.SSQL                                              PACKAGE         109.33      0.092988     8.67
                                                                    3      0.358872      0.262800

 D31B7.USRT001.SSQLFFO                                           PACKAGE          68.00      0.023910     8.00
                                                                    1      0.119931      0.090771

 ***  GRAND TOTAL  ***
                                                                PACKAGE          99.00      0.075718     8.50
                                                                    4      0.299137      0.219793

 ACCOUNTING REPORT COMPLETE
```

Figure 69. DB2PM Accounting Short Report Ordered by Package

Figure 70 shows the accounting short report ordered by plan name. Since the plan name is DISTSERV, data for different packages is summarized under this plan.

```
1
0 ============================================================
0              ACCOUNTING
0                REPORT
0                  LAYOUT(SHORT)
0                  ORDER(PLANNAME)
1    LOCATION: D31B7                        DB2 PERFORMANCE MONITOR (V3)                        PAGE:
1-1
     SUBSYSTEM: D31G                          ACCOUNTING REPORT - SHORT                  DB2 VERSION: V3
 INTERVAL FROM: 09/08/93 04:11:42.31                                                  REQUESTED FROM: NOT SPECIFIED
          TO: 09/08/93 04:13:12.42                 ORDER: PLANNAME                         TO: NOT SPECIFIED

                      #OCCURS #ROLLBK SELECTS INSERTS UPDATES DELETES CLASS1 EL.TIME CLASS2 EL.TIME GETPAGES SYN.READ LOCKSUS
 PLANNAME              #DISTRS #COMMIT FETCHES   OPENS  CLOSES PREPARE CLASS1 TCBTIME CLASS2 TCBTIME BUF.UPDT TOT.PREF #LOCKOU
 ------------------------ ------- ------- ------- ------- ------- ------- -------------- -------------- -------- -------- ----

 DISTSERV                    4      0    0.00   16.00   16.00   16.00      7.081398       0.299289   130.50     8.50   1.00
                            4      8   48.75    0.75    0.75    0.00      0.116822       0.075861   114.50     0.75   0

    ------------------------------------------------------------------------------------------------------------------
    |PROGRAM NAME       TYPE      #OCCURS  SQLSTMT  CL7 ELAP.TIME   CL7 TCB TIME  CL8 SUSP.TIME  CL8 SUSP|
    |SSQL               PACKAGE         3   109.33       0.358872       0.092988       0.262800      8.67|
    |SSQLFFO            PACKAGE         1    68.00       0.119931       0.023910       0.090771      8.00|
    ------------------------------------------------------------------------------------------------------------------

    ------------------------------------------------------------------------------
    |REQUESTER        METH  #DDFS    TRANS  #ROLLBK  #COMMIT  SQLRECV  ROWSENT  CONVI|
    |<WSD2194A>       APPL      4     1.00        0        8    82.50    48.00   1.00|
    ------------------------------------------------------------------------------

 ACCOUNTING REPORT COMPLETE
```

Figure 70. DB2PM Accounting Short Report Ordered by Plan Name

Figure 71 shows the accounting report ordered by main package within a plan name. This report facilitates distinguishing between accounting records that have the same plan name but executed different packages. Thus, the break up of the two packages SSQL and SSQLFFO executed under the same plan DISTSERV is presented. This would not be possible if the report were ordered by plan name as illustrated in Figure 70 on page 173.

```
1
0 ============================================================
0              ACCOUNTING
0                REPORT
0                  LAYOUT(SHORT)
0                  ORDER(PLANNAME-MAINPACK)
1    LOCATION: D31B7                      DB2 PERFORMANCE MONITOR (V3)                        PAGE:
1-1
     SUBSYSTEM: D31G                        ACCOUNTING REPORT - SHORT                  DB2 VERSION: V3
 INTERVAL FROM: 09/08/93 04:11:42.31                                                REQUESTED FROM: NOT SPECIFIED
          TO: 09/08/93 04:13:12.42           ORDER: PLANNAME-MAINPACK                          TO: NOT SPECIFIED

 PLANNAME                #OCCURS #ROLLBK SELECTS INSERTS UPDATES DELETES CLASS1 EL.TIME CLASS2 EL.TIME GETPAGES SYN.READ LOCKSUS
  MAINPACK               #DISTRS #COMMIT FETCHES   OPENS  CLOSES PREPARE CLASS1 TCBTIME CLASS2 TCBTIME BUF.UPDT TOT.PREF #LOCKOUT
 ------------------------ ------- ------ ------- ------- ------- ------- -------------- -------------- -------- -------- --------

 DISTSERV                     3      0    0.00   21.33   21.33   21.33      7.450440      0.359026   170.00     8.00    1.00
  SSQL                        3      6   43.33    0.67    0.67    0.00      0.146609      0.093132   152.67     0.67    0

    ---------------------------------------------------------------------------------------------------------
    |PROGRAM NAME       TYPE     #OCCURS  SQLSTMT  CL7 ELAP.TIME   CL7 TCB TIME  CL8 SUSP.TIME  CL8 SUSP|
    |SSQL              PACKAGE         3   109.33      0.358872      0.092988      0.262800     8.67|
    ---------------------------------------------------------------------------------------------------------

    ---------------------------------------------------------------------------
    |REQUESTER        METH  #DDFS    TRANS  #ROLLBK  #COMMIT  SQLRECV  ROWSENT  CONVI|
    |<WSD2194A>       APPL      3     1.00        0        6   108.00    42.67   1.00|
    ---------------------------------------------------------------------------

 DISTSERV                     1      0    0.00    0.00    0.00    0.00      5.974271      0.120076    12.00    10.00    1.00
  SSQLFFO                     1      2   65.00    1.00    1.00    0.00      0.027459      0.024048     0.00     1.00    0

    ---------------------------------------------------------------------------------------------------------
    |PROGRAM NAME       TYPE     #OCCURS  SQLSTMT  CL7 ELAP.TIME   CL7 TCB TIME  CL8 SUSP.TIME  CL8 SUSP|
    |SSQLFFO           PACKAGE         1    68.00      0.119931      0.023910      0.090771     8.00|
    ---------------------------------------------------------------------------------------------------------

    ---------------------------------------------------------------------------
    |REQUESTER        METH  #DDFS    TRANS  #ROLLBK  #COMMIT  SQLRECV  ROWSENT  CONVI|
    |<WSD2194A>       APPL      1     1.00        0        2     6.00    64.00   1.00|
    ---------------------------------------------------------------------------

 ***  TOTAL  ***
 DISTSERV                     4      0    0.00   16.00   16.00   16.00      7.081398      0.299289   130.50     8.50    1.00
                              4      8   48.75    0.75    0.75    0.00      0.116822      0.075861   114.50     0.75    0

    ---------------------------------------------------------------------------------------------------------
    |PROGRAM NAME       TYPE     #OCCURS  SQLSTMT  CL7 ELAP.TIME   CL7 TCB TIME  CL8 SUSP.TIME  CL8 SUSP|
    |ALL PROGRAMS       PACKAGE        4    99.00      0.299137      0.075718      0.219793     8.50|
    ---------------------------------------------------------------------------------------------------------

    ---------------------------------------------------------------------------
    |REQUESTER        METH  #DDFS    TRANS  #ROLLBK  #COMMIT  SQLRECV  ROWSENT  CONVI|
    |<WSD2194A>       APPL      4     1.00        0        8    82.50    48.00   1.00|
    ---------------------------------------------------------------------------

 ACCOUNTING REPORT COMPLETE
```

Figure 71. DB2PM Accounting Short Report Ordered by Main Package within Plan Name

Index

I

L

M

N

O

P

Q

R

S

T

V