IBM

Application System/400™

SC21-9659-1

Programming:
Database Guide

IBM

Application System/400™

SC21-9659-1

Programming: Database Guide

Second Edition (September 1989)

This major revision makes obsolete SC21-9659-0. Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition. See "About This Guide" for a summary of major changes to this edition.

This edition applies to Release 2, Modification Level 0, of the IBM Operating System/400 Licensed Program (Program 5728-SS1), and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions or technical newsletters.

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

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 licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

The numbers at the bottom right of illustrations are publishing control numbers and are not part of the technical content of this manual.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to your IBM-approved remarketer.

This publication could contain technical inaccuracies or typographical errors. A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department 245, Rochester, Minnesota, U.S.A. 55901. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About This Guide

This guide contains information about the AS/400 database management system and describes how to set up and use a database on the AS/400 system.

This guide does not cover in detail all the database-related capabilities on the AS/400 system. The *Programming: Structured Query Language/400 Programmer's Guide*, SC21-9609, and *Programming: Structured Query Language/400 Reference*, SC21-9608, describe how to use Structured Query Language/400 in more detail. The *Programming: Data Description Specifications Reference*, SC21-9620 describes how to specify OS/400 data description specifications (DDS). The *Programming: Control Language Reference*, SBOF-0481 describes how to specify OS/400 control language (CL) commands. The *Utilities: Interactive Data Definition Utility User's Guide*, SC21-9657, describes the OS/400 interactive data definition utility (IDDU) and data dictionary in more detail. The *Programming: Backup and Recovery Guide*, SC21-8079, describes database recovery considerations in more detail. This guide may refer to products that are announced but not yet available.

Who Should Use This Guide

This guide is intended for the system administrator or programmer who creates and manages files and databases on the system. In addition, the guide is intended for programmers who use the database in their programs.

What You Should Know

Before using this guide, you should be familiar with the introductory material for using the system. You should also understand how to write a high-level language program for the AS/400 system.

If you plan to use Structured Query Language/400 to create and process your database files, see the appropriate guides in "Related Printed Information" on page vi.

If you plan to use OS/400 IDDU to define data on the system, see the appropriate guides in "Related Printed Information" on page vi.

If you plan to use the database recovery functions on the AS/400 system, you should be familiar with the topics discussed in the *Programming: Backup and Recovery Guide*, SC21-8079.

How This Guide Is Organized

The parts in this guide are organized as follows:

- Introducing the AS/400 database

  The chapters in this part describe the concepts of database systems and how the database operates on the AS/400 system. This part also describes the basic steps in setting up, processing, and managing database files on the AS/400 system.

- Setting up database files

  The chapters in this part of the guide describe in detail how to define and create database files on the system.

- Processing database files in programs

  The chapters in this part of the guide describe in detail how to process database files in programs. Included is a description of opening database files, reading, updating, adding, and deleting records in the file, and closing database files.

- Managing database files

  This part of the guide describes additional functions and considerations when using the database on the AS/400 system. Included is a description of managing database file members, using database cross-reference information, and planning considerations for database recovery.

- Appendixes

  The appendixes discuss the minimum and maximum file sizes on the AS/400 system, the double-byte character set (DBCS) considerations, and the database lock considerations.

- Glossary

  The glossary defines words used in this guide. Use the glossary if you find an unfamiliar word.

- Index

  The index lists database processing tasks used in this guide. Use the index when you want to find instructions for a specific task.

How This Guide Has Changed

The following major changes were made since the previous edition of this guide:

- OS/400 DDS

  Changes have been made to the DDS coding form

- Duplicate key arrangement

  A new DDS keyword, FCFO (first-changed-first-out), has been added

- Hexadecimal data type is supported

- Communications Service Trace function is supported

- Input/Output considerations

  Determining if duplicate keys exist is described

- Appendix D, "Office Output File Considerations"

  This appendix has been removed from this manual. Output file information for Office command is now provided in *Programming: Office Services Concepts and Programmer's Guide*, SC21-9758.

Changes since the previous edition of the manual are indicated by a vertical line to the left of the change.

Related Online Information

The following online information is available on the AS/400 system. After pressing the Help key on any menu, you can press the Help key a second time to see an explanation of how the online information works, including the index search function. You can press either the Help key or F1 for help.

Help for Displays

You can press the Help key on any display to see information about the display. There are two types of help available:

Field
Extended

Field help explains the field on which the cursor is positioned when you press the Help key. For example, it describes the choices available for a prompt. If a system message appears at the bottom of the display, position the cursor on the message and press the Help key to see information about the cause of the message and the appropriate action to take.

Extended help explains the purpose of the display. Extended help appears if you press the Help key when the cursor is outside the areas for which field help is available.

To exit the online information, press F3 (Exit). You return to the display on which you pressed the Help key.

Index Search

Index search allows you to specify words or phrases that identify the information that you want to see. To use index search, press the Help key, then press F11 (Search index). You can also use index search by entering the Start Index Search (STRIDXSCH) command on any command line or by selecting option 2 on the User Support and Education menu.

Help for Control Language Commands

To see prompts for parameters for a control language command, type the command, then press F4 (Prompt) instead of the Enter key. To see extended help for the command, type the command on any command line and press the Help key.

Online Education

AS/400 online education provides training on a wide variety of topics. To use the online education, press F13 (User support) on any system menu to show the User Support menu. Then select the option to use online education.

Question-and-Answer Function

The question-and-answer (Q & A) function provides answers to questions you may have about using the AS/400 system. To use the Q & A function, press F13 (User support) on any system menu to show the User Support menu. Then select the option to use the question-and-answer function. You can also use the question-and-answer function by entering the Start Question and Answer (STRQST) command on any command line.

Related Printed Information

If you need more information, refer to the following guides:

- *Query: User's Guide*, SC21-9614. This guide has information about using AS/400 Query.
- *Communications: Distributed Data Management User's Guide*, SC21-9600. This guide has information about using OS/400 distributed data management (DDM).
- *Programming: Backup and Recovery Guide*, SC21-8079. This guide has information about using the backup and recovery functions of the AS/400 system.
- *Programming: Control Language Programmer's Guide*, SC21-8077. This guide has information about using the command language on the AS/400 system.
- *Programming: Control Language Reference*, SBOF-0481. This set of manuals has detailed information about all system commands, including database-related commands.

  This set of manuals is divided into five volumes. Volume 1 provides an overview of the control language commands and describes the syntax coding rules needed to code them. Volumes 2 through 5 describe every control language command, including commands for the system control program, languages, and utilities. The order numbers for these volumes are:

  Volume 1, SC21-9775
  Volume 2, SC21-9776
  Volume 3, SC21-9777
  Volume 4, SC21-9778
  Volume 5, SC21-9779

- *Programming: Data Description Specifications Reference*, SC21-9620. This manual has detailed information about using OS/400 data description specifications (DDS).
- *Programming: Data Management Guide*, SC21-9658. This guide has information about using other data management tools on the AS/400 system. Included are topics on the Copy File (CPYF) command and the override commands.
- *Programming: Security Concepts and Planning*, SC21-8083. This guide has information about using the integrated security features of the AS/400 system to secure your database file information.

- *Programming: Structured Query Language/400 Reference*, SC21-9608. This manual has detailed information about Structured Query Language/400 statements.
- *Programming: Structured Query Language/400 Programmer's Guide*, SC21-9609. This guide has information about using SQL/400.
- *Utilities: Interactive Data Definition Utility User's Guide*, SC21-9657. This guide has information about using OS/400 interactive data definition utility (IDDU).

Contents

Part 1. Introducing the AS/400 Database

Part 2. Setting Up Database Files

Part 3. Processing Database Files in Programs

Part 4. Managing Database Files

Figures

Part 1. Introducing the AS/400 Database

The chapters in this part provide an overview of the AS/400 database, database concepts, and database structure. This includes definitions of the terms frequently used when working with the AS/400 database and those concepts which make up the building blocks, or structure of the database on your AS/400 system. The database management system is defined and the advantages of this type of system versus a traditional database system are discussed.

The operation of database on your AS/400 system is also described in this part. This includes a basic overview on how to set up database files and how to use database files in a program. Other topics that are covered in this part include managing database files and functions that allow you to recover your database in the event of a system failure.

Other tasks covered in this part are setting up database files, including describing the database files to the system and then, actually creating the database files. Also, the basic steps for processing a database file in a program are covered.

All of the topics in Part 1 are dealt with in more detail later in this manual.

Chapter 1. The AS/400 Database

This chapter introduces the AS/400[1] database. It discusses why you might want to use a database management system instead of a system with no database manager. The chapter also discusses how database works on the AS/400 system.

Database Concepts

The term database is often associated with complex operating environments and large data processing staffs. In fact, a **database** is simply a place to store your data. In this sense, the files on your current system can be viewed as a database. What sets a system with a **database management system** apart from one without a database management system is that a database management system helps manage the data in the database for you, rather than you doing all of the work.

For example, if you need to read data for customer number 100 and that customer has information in both a master file record and an address file record, you would normally do two read operations in your program (one read for each file). However, if you were using a database manager, your program could do one read operation for customer number 100, and the database manager could get the data from the two files and present it to your program as if it were coming from a single file. Your program does not need to know how many files, and from which files, the data is being read.

Another example of how a database management system can help you concerns adding new fields to a file. Most programs on systems without a database are tied very closely to the way data is physically stored on the system. That is, if you have defined a file as containing fields *A*, *B*, and *C*, each program that uses that file is coded with the knowledge of that record layout. If you decide later to add field *D*, all your programs that use that file must be changed and recompiled.

A database management system, on the other hand, can hide the physical record layout of a file. In the example above, the database system could present data to your program as if the record layout was still *A*, *B*, *C* (even though the new physical record layout was *A*, *B*, *C*, *D*). Your programs would not have to change or be recompiled (unless, of course, the programs needed to use the new field).

Database management systems often have the following characteristics:

- Data and program independence
- Data sharing across applications
- Data secured at the file, record, and field levels
- Data integrity and recovery
- Data joined from more than one file
- Data selected and arranged based on processing requirements

In addition, database management systems are often associated with a data definition function. This function allows you to define data one time to the system, and that definition is maintained by the system.

[1] AS/400 is a trademark of the International Business Machines Corporation.

Data and Program Independence

The AS/400 database management system separates the program's logical view of the data from the way the data is physically stored on the system. The ability to logically view data independent of its physical structure allows you to make changes to the physical data without, necessarily, having to change or recompile your programs. For example, you can add fields to a file, remove fields from a file, or change the length of fields in a file without changing anything in programs that do not use those fields.

Data Sharing

The AS/400 database management system allows many users and programs to share the data in the database files at the same time. The AS/400 system maintains the integrity of the data regardless of the number of programs sharing the data or how those programs use the data. In addition, the system ensures that changes to database records are reflected in the database. Therefore, programs see the latest information. New programs and applications can be added to the system and have immediate access to the shared data, without affecting existing programs.

Because database management systems allow greater data sharing, you often can reduce the amount of data that you must define and store. Traditionally, each application has its own files. For example, both the inventory application and the accounts payable application might have their own version of the inventory file. The inventory application file might contain an inventory item number field, an item description field, and a warehouse location field. The accounts payable file might have an inventory item number field, an item description field, and a supplier number field.

The AS/400 database management system shares data across applications, reducing the need to duplicate data. In the previous example, one file containing the inventory item number field and item description field could be created and shared by both the accounts payable and inventory applications.

In addition, the database uses record locks to ensure data integrity and the ability to safely share data in the database. That is, the system automatically protects a record from being updated by more than one user at the same time. The system can also ensure that incomplete transactions are not seen by other users until a transaction is complete.

Data Security

Because the AS/400 database management system allows you to share data easily across many applications and users, you need a way to prevent unauthorized use of the data. The AS/400 system supplies a comprehensive security function to allow you to secure your data at many levels. For example, you have defined FILEA as containing three fields: customer name, customer address, and amount due. You can:

- Give authority for the whole file to specified users. For example, Cheryl and Lois can access FILEA, but Rick cannot.
- Give authority for certain types of file operations to specified users. For example, Dave can read, but not update any records in FILEA.

- Give authority for certain fields to be read by specified users. For example, Mark's view of the file includes the customer name and customer address fields, but not the amount due field.
- Give authority for certain fields to be changed by specified users. For example, Robin's view of the file can include all the fields, but Robin cannot change the amount due field.
- Give authority for certain records to specified users. For example, records with a certain value can only be viewed by the payroll department.

The ability to secure data at the file, record, and field levels can be combined in different ways for different users, according to your business requirements.

Data Integrity

The system allows you to journal (record) all changes that occur to the database. The journal can then be used for recovering your data in case the database file must be reset to a previous point.

The AS/400 database management system can recover data to the transaction level. That is, if a transaction ends abnormally, the AS/400 system can automatically reverse all changes the failed transaction made to the files. This ensures the data is at a known transaction boundary, and incomplete updates are not left in the database.

The AS/400 system can verify the validity of data for fields defined in the database. For example, you might define a field as having a valid range of 1000 to 2000. A display file can use this information to verify the correct information was typed by the operator. If the operator types anything outside of the range, the system rejects the data. Your program does not need to check for this range of data values; the system does the checking for you when the data is entered from a display. Only records whose data meets your validity checks are passed to your program.

The AS/400 system allows you to save and restore your database files, individually and by groups. You can also save and restore all the data on your system.

Joining Data from More Than One File

The AS/400 database management system allows you to use two or more physically separate files as if they were one file. The database management system does all the necessary read operations to get the data from the separate files and joins the data to make it appear as if it were coming from a single file. The AS/400 database management system does the work, not your program.

For example, assume you have an accounts receivable master file and an accounts receivable payment file on your current system. The master file has a record for each of your customers. The master file record format includes the customer number, customer name, and customer address fields. The payment file contains a record for each payment received. Each payment record includes the customer number, payment date, invoice number, and payment amount fields. There is a relationship between the master file data and the payment file data. That is, the record in the master file with customer number 15 can have one or more corresponding payment records in the payment file. Using a traditional file system to get the customer's master record information along with the customer's latest payment record, your program would have to:

1. Read the record in the master file.
2. Read the corresponding record in the payment file.

A database management system, once you have defined the relationship between the two files, could present the data to the program as if all the data resided in one file. That is, your program would do one read to retrieve the data from the two files.

Selecting and Arranging Data

The AS/400 database management system can improve productivity by doing some of the functions you would normally have to do in a program. This can reduce your coding effort. For example, the database management system can select records based on selection values you specify. Your program only sees the records you want it to see. In addition the database management system can present data in the sequence and groups you specify, without the need to sort or duplicate the data. You can also use the built-in database system functions to determine, among other things, the standard deviation, variance, square root, and sum of a series of values in the database. The database management system can do this work for you, rather than you coding it in a program.

Data Definition

The AS/400 system supplies a common, consistent way of describing data. When you create a database file, you describe the fields that are contained in that file. All programs that use that file can then use the common definition of the fields in that file. This can help reduce coding errors, makes programs easier to maintain, and helps ensure the consistency of field names and field characteristics.

Summary

Given these characteristics, some of the benefits of using a database management system include:

- Programs are easier and faster to write and maintain.
- You have better control over the data in your system.
- Data integrity and security are improved.
- Files and applications can easily grow and change.

The Integrated AS/400 Database Management System

The AS/400 database management system supports a rich set of database functions. However, what sets this system apart from most other systems is that it is **integrated** into the operating system and licensed internal code of the AS/400 system. That is, the database management system is a fundamental part of the AS/400 system. Any program that wants to store or use data on the AS/400 system uses the integrated database management system. The integrated database management system supplies not only a rich set of functions, but also a level of productivity, ease of use, and consistency that many other systems with separate database management systems cannot achieve. Some of the advantages of using the AS/400 integrated database management system are:

- No installation required
- Single source of data
- Data available to all programs
- Faster performance
- Improved consistency

No Installation Required

There is no requirement to install or maintain a separate database management system. When you install the AS/400 system, you install the integrated database management system.

Single Source of Data

Traditional systems that do not have an integrated database system usually have multiple ways of storing and retrieving data. The programmer frequently must use one language to access file data (a file interface), and another language to access database data (a database interface). In addition, the data stored by the file system often cannot be directly accessed by the database management system, and the data stored by the database management system cannot be directly accessed by the file system.

The AS/400 database serves as the single source of data on the system, and the AS/400 database manager serves as the single manager for that data. Because there is only one way to store and retrieve data on the system, programs can access *all* the data on the system (provided, of course, the user running the program has the proper authority to use the data). The single database manager supports several ways to access and manage the data:

- File interface (for example, read and write high-level language statements)
- Database interface - Structured Query Language/400 (SQL/400[2])
- Operating System/400[2] interactive data definition utility (IDDU)
- OS/400[2] control language
- System menus

In addition, products such as AS/400 Query, AS/400 Office, AS/400 PC Support, and the AS/400 Application Development Tools access data stored in the database.

Regardless of the method you choose, you can access all the data on the AS/400 system. For example, you can create a database using the database interface, then write a program using the file interface to process the data in that database. Or, you could create a file using the traditional file interface, then process that data using the database interface.

In addition, because a file interface is available for the database management system, you can use many programs originally written for traditional file systems, with little or no change. These traditional programs, when run on the AS/400 system, immediately use the integrated database management system. Over time, you can change your traditional file programs to take full advantage of the capabilities of the AS/400 database management system.

Data Available to All Programs

All programs that store or retrieve data on the AS/400 system use its integrated database management system. Therefore, you can use the database on the AS/400 system to store data from one program, then use that data in another, completely different application. For example, you can run AS/400 Query to select records from a production database file and write the results to a new database file. Then, you can run the AS/400 Business Graphics Utility to produce a chart of the data in that new file. You could then run a high-level language program to change the data in

[2] SQL/400, Operating System/400, and OS/400 are trademarks of the International Business Machines Corporation.

the database file and, finally, run AS/400 Office to merge the changed data into documents.

Performance

An integrated database management system normally provides superior performance when compared to the performance of database management systems added on later. The AS/400 system was designed with database in mind, so the system runs its database management system efficiently.

Consistency

The database management system uses other AS/400 integrated functions. For example, the database manager uses the integrated security functions on the AS/400 system. Therefore, once you learn how to use the security function on the system, you will use it to secure data in the database, just like you use it to secure anything else on the system.

Another example of the benefits of consistency is how the the database management system communicates with you. The database system uses the common message handling function on the AS/400 system. Therefore, messages sent by the database management system to the programmer or operator are in the same, consistent format as messages sent by other functions of the system.

Chapter 2. The AS/400 Database Structure

This chapter discusses the basic concepts of the AS/400 database structure.

Fields and Field Definitions

A **field**, or column, is a named group of values that helps describe an attribute of some thing. For example, the field *Address* is a group of characters that, taken together, describe one attribute of a customer. Other fields that help describe a customer could be: *Custname*, *Age*, *Sex*, and *Amtdue*. A field is the smallest unit of addressable data in the database. A **field definition** is a description of the field attributes (such as its length or whether the field is numeric or character).

Records and Record Formats

A **record**, or row, is a named group of fields that, taken together, describe some thing. For example, the fields *Custname*, *Custaddr*, *Age*, *Sex*, and *Amtdue* when put together describe a customer. These fields, then, make up a record. A **record format** is simply a description of all the fields in the record and their arrangement in the record. The record format is given a name; some high-level languages can require the use of the record format name in the program.

Database Files and File Descriptions

A **file** is a named group of records. For example, the customer master file is the group of all customer master records. A **file description** describes the file including the record format and access path used by the file. (Access paths are described later in this chapter.)

Note: In this guide the terms file and database file are used interchangeably. In addition, in this guide the term database refers to all the database files on the system.

The two types of database files are: physical files and logical files. Both physical files and logical files can be used to access data in the database. In fact, in most cases, a programmer does not need to know whether data is accessed through a physical or logical file.

Physical Files

Physical files, or tables, contain the actual data stored on the system. They are similar to traditional files. Each physical file has only one fixed-length record format.

Note: IDDU can be used to make it appear to AS/400 Query, AS/400 PC Support, and the AS/400 Application Development Tools data file utility (DFU) that a physical file contains more than one record format.

A physical file can have a keyed sequence access path to present the data in a sequence other than the order in which the records were added.

Logical Files

Logical files do not actually contain data, but describe how records contained in one or more physical files are to be presented to your program. Some of the things you can do with a logical file are:

- Logically change the attributes of fields from physical files (for example, field length and field order).
- Provide additional logical sequences of records.
- Secure one or more fields in physical files from being read or changed.
- Derive new fields from physical file fields.
- Secure specific physical file records from being read.
- Make two or more files appear as a single file.

The four categories of logical files are:

- A **simple logical file**, which uses data from one physical file. A simple logical file is the most commonly used category of logical file. It is used to select fields or records from the physical file it is based on. It is also used to arrange the physical file data through a keyed sequence access path. You can read, update, add, and delete records through a simple logical file.
- A **join logical file**, which combines (in one record format) fields from two or more physical files. A join logical file is read-only; you cannot change, add, or delete records through a join logical file.
- A **multiple format logical file**, which uses fields from two or more physical files. A multiple format logical file has more than one record format. That is, a record format for each physical file used must be described in the logical file. The records from each of the physical files is presented in a hierarchical fashion. You can read, update, add, or delete data through a multiple format logical file.
- A **view**, which is created using SQL/400. A view is a logical file without a keyed access path defined for it. One of the most powerful aspects of a view is that it can be created over physical files and other views. You can read, update, add, or delete data through a view. Views are described and created only through SQL/400 statements; therefore, they are discussed in more detail in the *SQL/400 Programmer's Guide.*

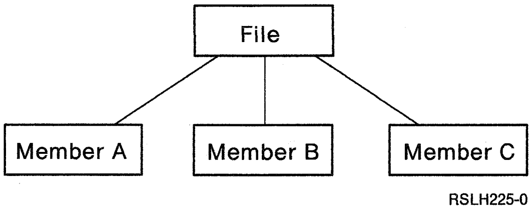
Data and Source Files

Database files are also categorized as either data files or source files. A **data file** contains the actual data, or a logical view of it, that your programs use. A **source file** contains source specifications the system uses to create an object. For example, to create a program you enter the source statements into a source file, and then run the appropriate command to use the source statements to create a program. Source files can be processed in the same way as data files. For more information about using source files, see Chapter 17.

Members

Data records in a database file are grouped into **members**. All the records in a file can be in one member or they can be grouped in different members.

If you want to process data in a file, the file must have at least one member. The following illustrates the concept of members:



Each member has its associated data and its own access path for that data. The access path specifications and the record formats described in the file are only specified once in the DDS to create the file, and are used for each member's access paths and record formats.

Normally, database data files have only one member, the one added by default when the file is created. Because, by default, most files are created with one member and most database commands and operations use it, you do not normally have to use member names when working with files.

Depending on how you use the data in the file, you might want to divide the file into smaller groups of records. If you decide to make smaller groups, you can manage them more easily by assigning a member name to each group. The system can then read only records from that member when you specify the member name.

For example, you define an accounts receivable file. You decide to keep one year's data in that file, but you usually process just one month's data at a time. In this case, you can create a physical file with 12 members, one named for each month. The JANUARY member only contains data for January, the FEBRUARY member only contains data for February, and so on. You can then process each month's data separately, by processing a member at a time. At year end, you can process several (or all) of the members together.

Members are especially useful when storing source statements in a database file. Members allow you to more easily manage the source. For example, if you had one file that contained the source statements for all your COBOL/400[1] programs, then you could divide that file into members, with each member containing one program's source statements. So, the PGMA member would contain the source statements for PGMA, the PGMB member would contain the source statements for PGMB, and so on. You could manage each program's source separately through members, without creating multiple files.

[1] COBOL/400 is a trademark of the International Business Machines Corporation.

Access Paths

When you add data to a physical file, normally new records are physically stored at the end of the file. That is, records are stored in the order that they arrive in the file. However, you may want to read records in an order different from the arrival sequence of the data. For example, you may want to read data in sequence by the *Custname* field.

An **access path** describes the order in which records are to be read. The two types of access paths are: arrival sequence access paths and keyed sequence access paths.

Arrival Sequence Access Path

An **arrival sequence access path** is the order of records as they are stored in the file. Processing files using the arrival sequence access path is similar to processing consecutive, sequential, or direct files on traditional systems.

Keyed Sequence Access Path

A **keyed sequence access path** is the order of records established by the contents of one or more fields in the record. These fields are called key fields. A **key field** is a field whose contents are used to arrange the sequence of the records in the file. Using a keyed sequence access path, the system can present data to a program arranged by the key field of the file. For example, you could retrieve data arranged by the key field *Custname*. Processing files using a keyed sequence access path is similar to processing indexed or keyed files on traditional systems.

A keyed sequence access path is changed whenever records are added to or deleted from a file or whenever a record is changed and the contents of a key field changes. Thus, the access path to the data is automatically maintained by the system.

Chapter 3. Working with the AS/400 Database

This chapter describes how you work with database files on the system. Topics include how you set up database files, use database files in a program, and manage database files. This chapter is only an overview. Detailed information about these topics are covered later in this guide.

Setting Up a Database File

The two steps to setting up a database file on the system are:

1. Describe the database file to the system.
2. Create the database file.

Describing the Database File

You can choose whether or not to describe the fields in your database file to the system. If you choose not to describe the fields that make up the record format for the file, then you simply tell the system the length of a record in your physical file (through a parameter on the create file commands). It is then up to you to describe the record format in each of the programs that use that file. A file that must be described by each program that uses it is called a program-described file. This approach is similar to most traditional file systems, but it does not take full advantage of the capabilities of the AS/400 database management system.

The preferred approach is to describe the file, in detail, to the system one time. This lets you use more of the functions of the database management system. For example, once the file is described and created, you do not have to code the record format in every program that uses that file. Instead, when you write your program you can request the compiler to automatically extract the field information from the system and include it in your program. A file containing a description of its fields is called an externally described file because the description of the file is maintained as part of the file, external to your programs. Externally described data saves you coding effort, makes it easier to maintain your programs, helps ensure consistent use of field names in your programs, and can provide better documentation.

Externally described and program-described files can also be described in a data dictionary. A **data dictionary** on the AS/400 system is a set of database files that contain file, record format, and field definitions. A file that is defined in the data dictionary is called a dictionary-described file. IDDU allows you to manage the data dictionary. For more information about the data dictionary and IDDU, see the *IDDU User's Guide*.

A database file description can contain:

- A description of the record format. The record format describes the fields in the file. Some of the things you can describe for each field include:
  The field name
  The size of the field
  The field data type (for example, packed decimal or character)
  Text description for each field
  Validity checks for each field (applies when the field is used in a display file)
  Editing values for each field (applies when the field is used in a display or printer file)
- A description of the access path. The access path describes the order in which records are to be presented to programs. Some of the things you can describe for the access path include:
  If the access path is arrival or keyed sequence
  The key field(s)
  If unique or duplicate keys are supported in the file

Creating the Database File

After you have described the database file to the system, you can create the file. When you create a file, you can specify attributes for that file. Some of the attributes you can specify include:

- A text description of the file
- The size of the file
- The amount of time a program should wait if a record in the file is locked by another user or program
- The maximum number of members allowed in the file

You can specify many additional database file attributes. For more information about file attributes, see "Specifying Database File and Member Attributes" on page 4-24.

An Example of Using DDS to Set Up Database Files

As a guide to setting up your first physical and logical files using Data Description Specifications (DDS), you might follow these steps:

1. Create a source file (see "Creating a Source File" on page 17-1).
2. Enter the DDS for a physical file into the source file (see "Describing Database Files Using DDS" on page 4-6 and "Working with Source Files" on page 17-4).
3. Create the physical file (see "Creating Database Files" on page 4-23).
4. Enter the DDS for a logical file into the source file (see "Describing Database Files Using DDS" on page 4-6 and "Working with Source Files" on page 17-4).
5. Create the logical file (see "Creating Database Files" on page 4-23).

Processing a Database File in a Program

To use a database file in a program, the basic steps are:

1. Determine the run-time attributes of the file.
2. Open the database file.
3. Process the data in the file.
4. Close the database file.

Before you open and use a file in a program, you should consider how you plan to use the file. For example, if you attempt to read a record that is locked by someone else, your program normally waits 60 seconds for the record to be released. You may want to wait a shorter or longer period of time. You can control this run-time file attribute, along with other file attributes. It is important to understand these run-time file attributes to take full advantage of the database management system. A description of the run-time file attributes are discussed in Chapter 8.

A program must open a file before working with the data in that file. An open request connects a file to a program. If you do not specifically request that the file be opened, some languages automatically do it for you when you first refer to the file. For more information about opening a file, see Chapter 9.

After the file is open, the program can proceed to read, update, add, and delete records in the file. For more information about file processing operations, see Chapter 10.

The program indicates that it is finished processing the file by closing the file. A close request disconnects the file from the program. If the program does not specifically request that the file be closed, the language can automatically close the file. For more details about closing database files, see Chapter 11.

Managing Database Files

The AS/400 system has a number of commands to help you manage your database system. These include commands to:

- Create and change database files and members
- Display database cross-reference information
- Start and stop database recovery functions

You can also use the Work with Files (WRKF) command and menus to help manage your database files. The WRKF command has menus that allow you to:

- Copy data from a file
- Delete a file
- Display the records in a file member
- Display the description of the file
- Save the file
- Restore the file
- Change certain file attributes

Member Operations

Because data is kept in file members, the system supplies you with a number of functions to manage files and members:

- Add members to a database file
- Rename members
- Remove members from a database file
- Change processing attributes of database file members
- Prepare data in physical file members
- Clear data in physical file members
- Reorganize data in physical file members
- Display data in physical file members

These commands are discussed in more detail in Chapter 13.

Database Cross-Reference Information

The AS/400 system contains information to assist you in cross-referencing files and program use. This information can be very important to programmers in determining such things as:

- What files are used in specified programs
- What files depend on other files for data or access paths
- The attributes of specified files
- The fields defined in a specified file or record format

The programmer can use this information to help determine the effect that a change to a field or file might have on other programs in the system.

Some of the system commands can write the results of the command to a database file. Then, you can run a program to analyze the results.

These commands and functions are discussed in more detail in Chapter 15. If a file is described using IDDU, it can provide cross-reference information. For more information about the cross-reference information available through IDDU, see the *IDDU User's Guide*.

Database Recovery

It is always a good data processing practice to be prepared to recover your data should the system fail for any reason. The AS/400 system supplies a number of integrated functions to help you recover your database files quickly. These functions include:

- A journal to record changes to your database files that can be used to help recover your data and access paths.
- A function, known as commitment control, to help ensure that, should a job end abnormally, any changes made to the database that are incomplete are rolled-back to a known, clean state (known as a transaction boundary).
- A comprehensive set of commands to save and restore your data.

For more detail about the AS/400 database recovery functions, see Chapter 16 of this guide and the *Backup and Recovery Guide*.

Remote File Access

Programs on the AS/400 system can access files on remote systems through OS/400 DDM. **Distributed data management** (DDM) allows remote files to appear to the programmer and end user as local database files. Programs using DDM do *not* need special or additional programming statements. The fact that the program is accessing data on a remote system, as opposed to the local system, is transparent to the program.

DDM files have the following characteristics:

- The AS/400 system supports read, add, update, and delete operations to remote database files. Programs and commands using DDM files run as though the remote database files were local database files.
- The DDM file, in effect, represents a remote database file. The DDM file refers to the communications path and the remote file, and links the remote system to the local system when the file is opened.

For more information about DDM, see the *DDM User's Guide*.

Part 2. Setting Up Database Files

The chapters in this part describe in detail how to set up any AS/400 database file. This includes describing database files and access paths to the system and the different methods that can be used. The ways that your programs use these file descriptions and the differences between using data that is described in a separate file or in the program itself is also discussed.

This part includes a chapter with guidelines for describing and creating logical files. This includes information on describing logical file record formats and different types of field use using data description specifications (DDS). Also information is included on describing access paths using DDS as well as using access paths that already exist in the system. Information on defining logical file members to separate the data into logical groups is also included in this chapter.

A section on join logical files includes considerations for using join logical files, including examples on how to join physical files and the different ways physical files can be joined. Information on performance, integrity, and a summary of rules for join logical files is also included.

There is a chapter on database security in this part which includes information on security functions such as file security, public authority, restricting the ability to change or delete any data in a file, and using logical files to secure data. The different types of authority that can be granted to a user for a database file and the types of authorities you can grant to physical files is also included.

Chapter 4. General Considerations

This chapter discusses things to consider when you set up any AS/400 database file. Later chapters will discuss unique considerations for setting up physical and logical files.

Describing Database Files

Records in database files can be described in two ways:

- Field level description. The fields in the record are described to the system. Some of the things you can describe for each field include: name, length, data type, validity checks, and text description. Database files that are created with field level descriptions are referred to as externally described files.
- Record level description. Only the length of the record in the file is described to the system. The system does *not* know about fields in the file. These database files are referred to as program-described files.

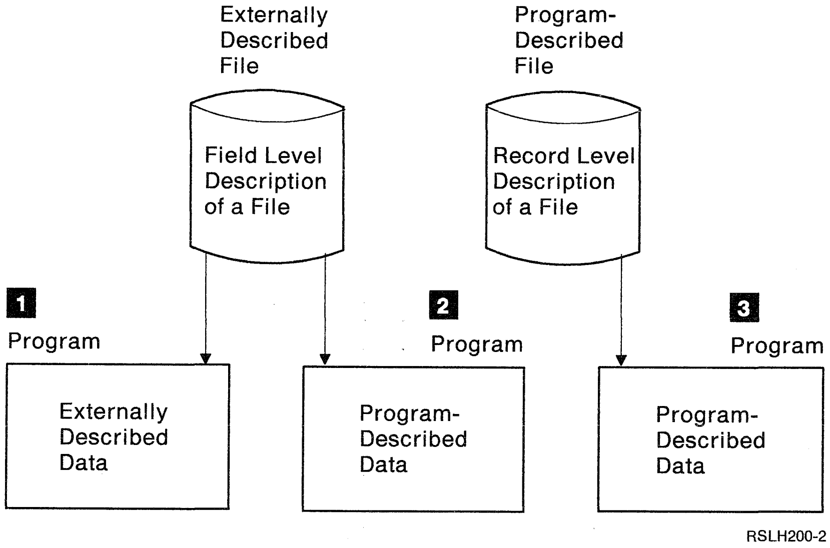
Regardless of whether a file is described to the field or record level, you must describe and create the file before you can compile a program that uses that file. That is, the file must exist on the system before you use it.

Programs can use file descriptions in two ways:

- The program uses the field-level descriptions that are part of the file. Because the field descriptions are external to the program itself, the term, externally described data, is used.
- The program uses fields that are described in the program itself; therefore, the data is called program-described data. Fields in files that are only described to the record level must be described in the program using the file.

Programs can use either externally described or program-described files. However, if you choose to describe a file to the field level, the system can do more for you. For example, when you compile your programs, the system can extract information from an externally described file and automatically include field information in your programs. Therefore, you do not have to code the field information in each program that uses the file.

The following figure shows the typical relationships between files and programs on the AS/400 system:


Externally Described File
Program-Described File
Field Level Description of a File
Record Level Description of a File
1
Program
2
Program
3
Program
Externally Described Data
Program-Described Data
Program-Described Data
RSLH200-2

1 The program uses the field level description of a file that is defined to the system. At compilation time, the language compiler copies the external description of the file into the program.

2 The program uses a file that is described to the field level to the system, but it does not use the actual field descriptions. At compilation time, the language compiler does not copy the external description of the file into the program. The fields in the file are described in the program. In this case, the field attributes (for example, field length) used in the program must be the same as the field attributes in the external description.

3 The program uses a file that is described only to the record level to the system. The fields in the file must be described in the program.

Externally described files can also be described in a program. You might want to use this method for compatibility with previous systems. For example, you want to run programs on the AS/400 system that originally came from a traditional file system. Those programs use program-described data, and the file itself is only described to the record level. At a later time, you describe the file to the field level (externally described file) to use more of the database functions available on the system. Your old programs, containing program-described data, can continue to use the externally described file while new programs use the field-level descriptions that are part of the file. Over time, you can change one or more of your old programs to use the field level descriptions.

Dictionary-Described Data

A program-described file can be dictionary-described. You can describe the record format information using IDDU. Even though the file is program-described, AS/400 Query, AS/400 PC Support, and DFU will use the record format description stored in the data dictionary.

An externally described file can also be dictionary-described. You can use IDDU to describe a file, then create the file using IDDU. The file created is an externally described file. You can also move into the data dictionary the file description stored in an externally described file. The system always ensures that the definitions in the data dictionary and the description stored in the externally described file are identical.

Methods of Describing Data to the System

If you want to describe a file just to the record level, you can use the record length (RCDLEN) parameter on the Create Physical File (CRTPF) and Create Source Physical File (CRTSRCPF) commands.

If you want to describe your file to the field level, several methods can be used to describe data to the database system: IDDU, SQL/400 commands, or DDS.

Note: Because DDS has the most options for defining data for the programmer, this guide will focus on describing database files using DDS.

OS/400 IDDU

Physical files can be described using IDDU. You might use IDDU because it is a menu-driven, interactive method of describing data. You also might be familiar with describing data using IDDU on a System/36. In addition, IDDU allows you to describe multiple-format physical files for use with Query, AS/400 PC Support, and DFU.

When you use IDDU to describe your files, the file definition becomes part of the OS/400 data dictionary.

For more information about IDDU, see the *IDDU User's Guide*.

Structured Query Language/400

The Structured Query Language/400 can be used to describe an AS/400 database file. SQL/400 supports statements to describe the fields in the database file, and to create the file.

You might use SQL/400 because it is the IBM Systems Application Architecture database language for defining and processing database data. When database files are created using SQL/400, the description of the file is automatically added to a data dictionary in the SQL collection. The data dictionary (or catalog) is then automatically maintained by the system.

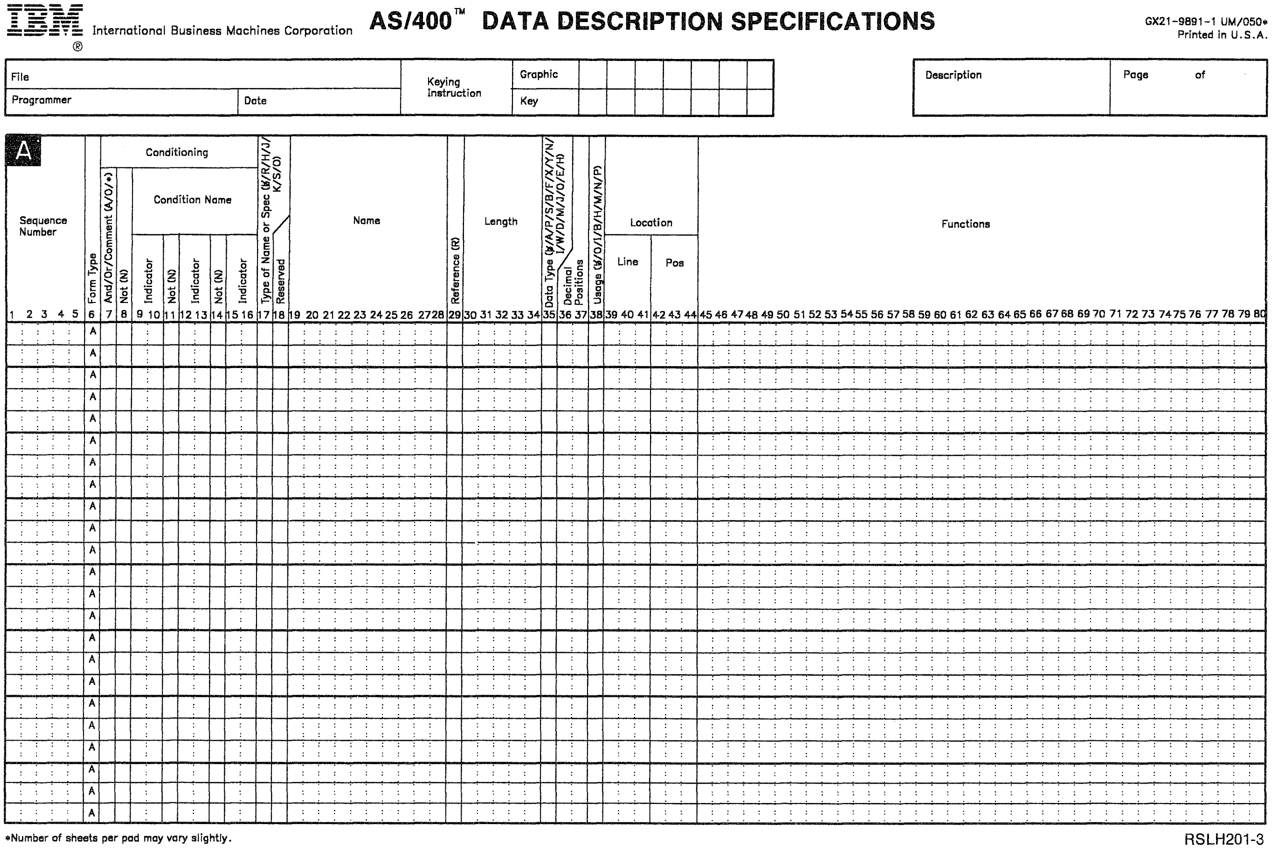
For more information about SQL/400, see the *SQL/400 Programmer's Guide*.

OS/400 DDS

Externally described data files can be described using DDS. Using DDS, you provide descriptions of the field, record, and file level information.

You might use DDS because it provides the most options for the programmer to describe data in the database. For example, only with DDS can you describe key fields in logical files.

The *DDS Coding Form*, shown in Figure 4-1 on page 4-4, provides a common format for describing data externally.



International Business Machines Corporation

AS/400™ DATA DESCRIPTION SPECIFICATIONS

GX21-9891-1 UM/050*
Printed in U.S.A.

File

Programmer

Date

Keying Instruction

Graphic

Key

Description

Page of

A		Conditioning															Location		
Sequence Number	Form Type	And/Or/Comment (A/O/*)	Not (N)	Condition Name					Type of Name or Spec (b̸/R/H/J/K/S/O)	Reserved	Name	Reference (R)	Length	Data Type (b̸/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)	Decimal Positions	Usage (b̸/O/I/B/H/M/N/P)			Functions
				Indicator	Not (N)	Indicator	Not (N)	Indicator									Line	Pos	
1 2 3 4 5	6	7	8	9 10	11	12 13	14	15 16	17	18	19 20 21 22 23 24 25 26 27 28	29	30 31 32 33 34	35	36 37	38	39 40 41	42 43 44	45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		
	A																		

*Number of sheets per pad may vary slightly.

RSLH201-3

Figure 4-1. Sample DDS Form

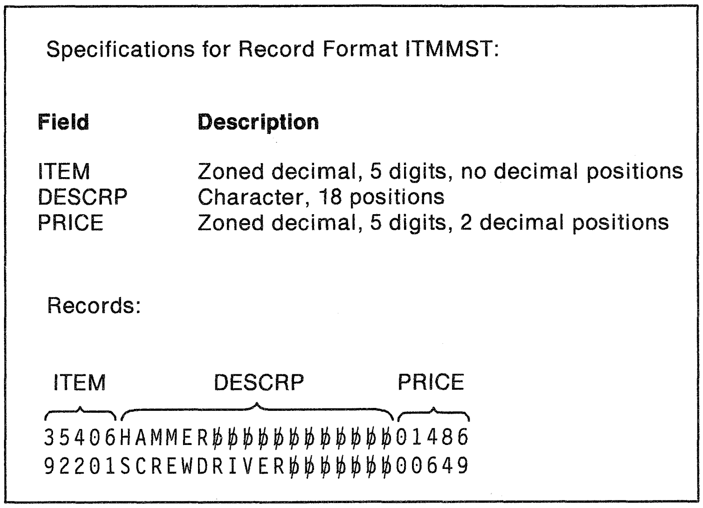
The *DDS Reference* contains a detailed description of DDS functions to describe physical and logical files.

Describing a Database File to the System

When you describe a database file to the system, you describe the two major parts of that file: the record format and the access path.

Describing the Record Format: The record format describes the order of the fields in each record. The record format also describes each field in detail including: length, data type (for example, packed decimal or character), validity checks, text description, and other information.

The following example shows the relationship between the record format and the records in a physical file:



Specifications for Record Format ITMMST:

Field	Description
ITEM	Zoned decimal, 5 digits, no decimal positions
DESCRP	Character, 18 positions
PRICE	Zoned decimal, 5 digits, 2 decimal positions

Records:

```
 ITEM          DESCRP          PRICE
35406HAMMERƀƀƀƀƀƀƀƀƀƀƀƀ01486
92201SCREWDRIVERƀƀƀƀƀƀƀ00649
```

RSLH203-1

A physical file can have only one record format. The record format in a physical file describes the way the data is actually stored. Physical files do not support variable length fields; therefore, all records in a physical file are fixed length and have the same field descriptions.

A logical file contains no data. Logical files are used to arrange data from one or more physical files into different formats and sequences. For example, a logical file could change the order of the fields in the physical file, or present to the program only some of the fields stored in the physical file.

A logical file record format can change the length and data type of fields stored in physical files. The system does the necessary conversion between the physical file field description and the logical file field description. For example, a physical file could describe FLDA as a packed decimal field of 5 digits and a logical file using FLDA might redefine it as a zoned decimal field of 7 digits. In this case, when your program used the logical file to read a record, the system would automatically convert (unpack) FLDA to zoned decimal format.

Describing the Access Path: An access path describes the order in which records are to be retrieved. When you describe an access path, you describe whether it will be a keyed sequence or arrival sequence access path. Access paths are discussed in more detail in "Describing the Access Path for the File" on page 4-16.

Naming Conventions

The file name, record format name, and field name can be as long as 10 characters and must follow all system naming conventions, but you should keep in mind that some high-level languages have more restrictive naming conventions than the system does. For example, RPG/400[1] allows only 6-character names, while the system allows 10-character names. In some cases, you can temporarily change (rename) the system name to one that meets the high-level language restrictions.

[1] RPG/400 is a trademark of the International Business Machines Corporation.

For more information about renaming database fields in programs, see your high-level language guide.

In addition, names must be unique as follows:

- Field names must be unique in a record format.
- Record format names and member names must be unique in a file.
- File names must be unique in a library.

Describing Database Files Using DDS

When you describe a database file using DDS, you can describe information at the file, record format, join, field, key, and select/omit levels:

- File level DDS give the system information about the entire file. For example, you can specify whether all the key field values in the file must be unique.
- Record format level DDS give the system information about a specific record format in the file. For example, when you describe a logical file record format, you can specify the physical file that it is based on.
- Join level DDS give the system information about physical files used in a join logical file. For example, you can specify how to join two physical files.
- Field level DDS give the system information about individual fields in the record format. For example, you can specify the name and attributes of each field.
- Key field level DDS give the system information about the key fields for the file. For example, you can specify which fields in the record format are to be used as key fields.
- Select/omit field level DDS give the system information about which records are to be returned to the program when processing the file. Select/omit specifications apply to logical files only.

Example of Describing a Physical File Using DDS

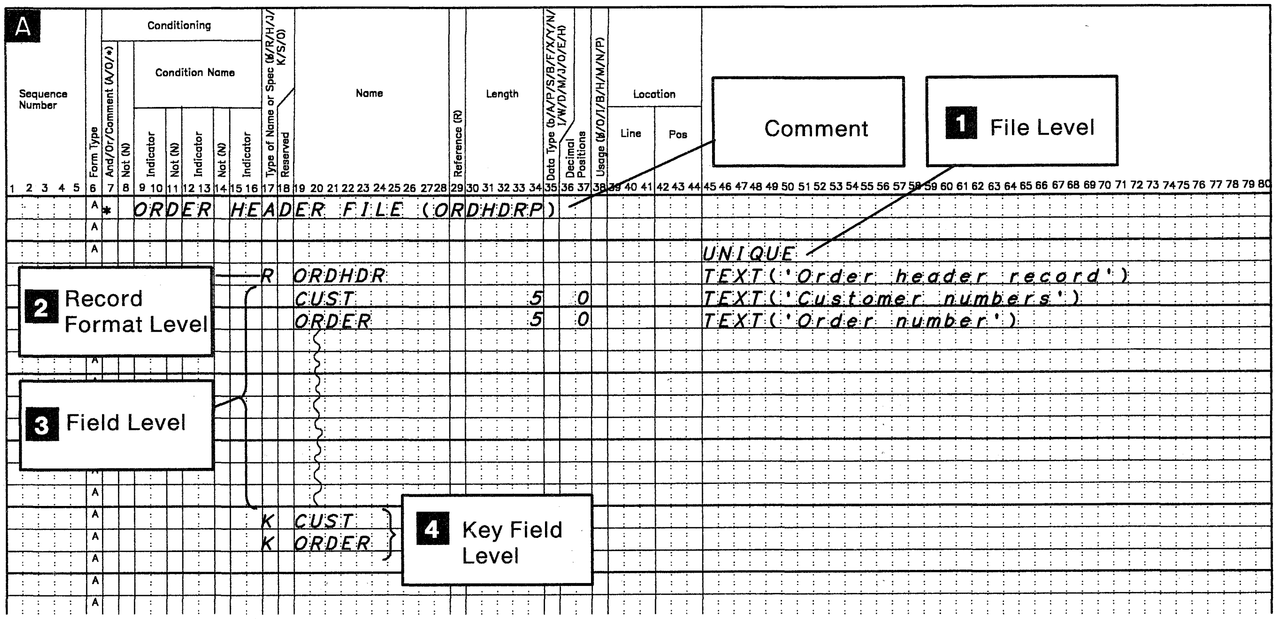
The DDS for a physical file must be in the following order (Figure 4-2 on page 4-7):

1 File level entries (optional). The UNIQUE keyword is used to indicate that the value of the key field in each record in the file must be unique. Duplicate key values are not allowed in this file.

2 Record format level entries. The record format name is specified, along with an optional text description.

3 Field level entries. The field names and field lengths are specified, along with an optional text description for each field.

4 Key field level entries (optional). The field names used as key fields are specified.


A
Comment
1 File Level
2 Record Format Level
3 Field Level
4 Key Field Level
RSLH230-1

```
     A* ORDER HEADER FILE (ORDHDRP)
     A
     A                                      UNIQUE
     A          R ORDHDR                    TEXT('Order header record')
     A            CUST           5  0       TEXT('Customer numbers')
     A            ORDER          5  0       TEXT('Order number')
     A          K CUST
     A          K ORDER
```

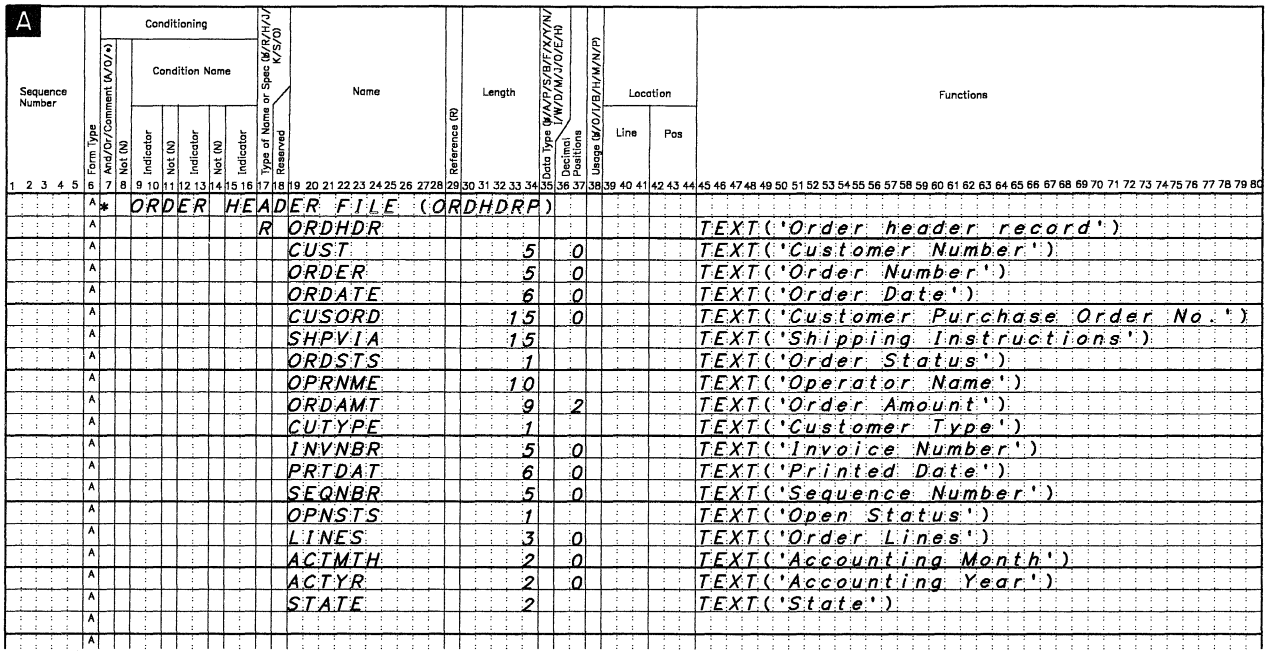
Figure 4-2. DDS for a Physical File (ORDHDRP)

The following example shows a physical file ORDHDRP (an order header file), which has an arrival sequence access path without key fields specified, and the DDS necessary to describe that file.

Record Format (ORDHDR):

Customer Number (CUST)	Order Number (ORDER)	Order Date (ORDATE)	Purchase Order Number (CUSORD)	Shipping Instructions (SHPVIA)	Order Status (ORDSTS)	•••	State (STATE)
Packed Decimal Length 5 No Decimals	Packed Decimal Length 5 No Decimals	Packed Decimal Length 6 No Decimals	Packed Decimal Length 15 No Decimals	Character Length 15	Character Length 1		Character Length 2

RSLH231-6



```
     A* ORDER HEADER FILE (ORDHDRP)
     A          R ORDHDR                    TEXT('Order header record')
     A            CUST           5  0       TEXT('Customer Number')
     A            ORDER          5  0       TEXT('Order Number')
     A            ORDATE         6  0       TEXT('Order Date')
     A            CUSORD        15  0       TEXT('Customer Purchase Order No.')
     A            SHPVIA        15          TEXT('Shipping Instructions')
     A            ORDSTS         1          TEXT('Order Status')
     A            OPRNME        10          TEXT('Operator Name')
     A            ORDAMT         9  2       TEXT('Order Amount')
     A            CUTYPE         1          TEXT('Customer Type')
     A            INVNBR         5  0       TEXT('Invoice Number')
     A            PRTDAT         6  0       TEXT('Printed Date')
     A            SEQNBR         5  0       TEXT('Sequence Number')
     A            OPNSTS         1          TEXT('Open Status')
     A            LINES          3  0       TEXT('Order Lines')
     A            ACTMTH         2  0       TEXT('Accounting Month')
     A            ACTYR          2  0       TEXT('Accounting Year')
     A            STATE          2          TEXT('State')
     A
     A
```

RSLH232-1

The R in position 17 indicates that a record format is being defined. The record format name ORDHDR is specified in positions 19 through 28.

You make no entry in position 17 when you are describing a field; a blank in position 17 along with a name in positions 19 through 28 indicates a field name.

The data type is specified in position 35. The valid data types are:

Entry	**Meaning**
A	Character
P	Packed decimal
S	Zoned decimal
B	Binary
F	Floating point
H	Hexadecimal

Notes:

1. For double-byte character set (DBCS) data types, see Appendix B.
2. The AS/400 system performs arithmetic operations more efficiently for packed decimal than for zoned decimal.

3. Some high-level languages do not support floating-point data.
4. Some special considerations that apply when you are using floating-point fields are:
   - The precision associated with a floating-point field is a function of the number of bits (single or double precision) and the internal representation of the floating-point value. This translates into the number of decimal digits supported in the significant and the maximum values that can be represented in the floating-point field.
   - When a floating-point field is defined with fewer digits than supported by the precision specified, that length is only a presentation length and has no effect on the precision used for internal calculations.
   - Although floating-point numbers are accurate to 7 (single) or 15 (double) decimal digits of precision, you can specify up to 9 or 17 digits. You can use the extra digits to uniquely establish the internal bit pattern in the internal floating-point format so identical results are obtained when a floating-point number in internal format is converted to decimal and back again to internal format.

If the data type (position 35) is not specified, the decimal positions entry is used to determine the data type. If the decimal positions (positions 36 through 37) are blank, the data type is assumed to be character (A); if these positions contain a number 0 through 31, the data type is assumed to be packed decimal (P).

The length of the field is specified in positions 30 through 34, and the number of decimal positions (for numeric fields) is specified in positions 36 and 37. If a packed or zoned decimal field is to be used in a high-level language program, the field length must be limited to the length allowed by the high-level language you are using. The length is not the length of the field in storage but the number of digits or characters specified externally from storage. For example, a 5-digit packed decimal field has a length of 5 specified in DDS, but it uses only 3 bytes of storage.

Example of Describing a Logical File Using DDS

The DDS for a logical file must be in the following order (Figure 4-3 on page 4-10):

1 File level entries (optional). In this example, the UNIQUE keyword indicates that for this file the key value for each record must be unique; no duplicate key values are allowed.

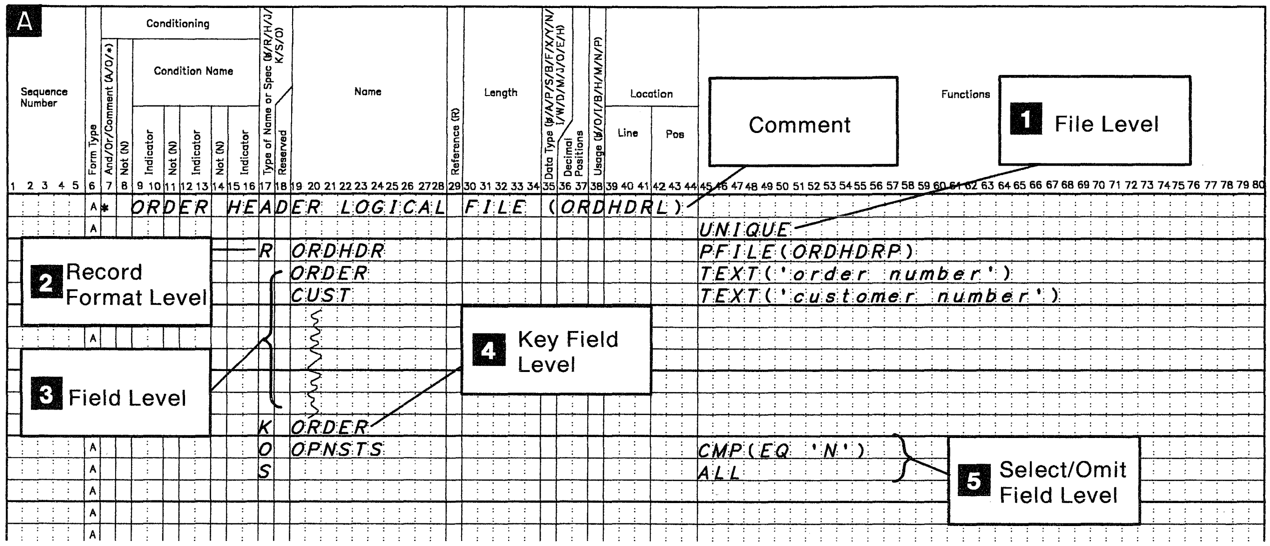
For each record format:

2 Record level entries. In this example, the record format name, the associated physical file, and an optional text description are specified.

3 Field level entries (optional). In this example, each field name used in the record format is specified.

4 Key field level entries (optional). In this example, the *Order* field is used as a key field.

5 Select/omit field level entries (optional). In this example, all records whose *Opnsts* field contains a value of N are omitted from the file's access path. That is, programs reading records from this file will never see a record whose *Opnsts* field contains an N value.



```
     A* ORDER HEADER LOGICAL FILE (ORDHDRL)
     A                                      UNIQUE
     A          R ORDHDR                    PFILE(ORDHDRP)
     A            ORDER                     TEXT('order number')
     A            CUST                      TEXT('customer number')
     A
     A          K ORDER
     A          O OPNSTS                    CMP(EQ 'N')
     A          S                           ALL
```

Figure 4-3. DDS for a Simple Logical File (ORDHDRL)

A logical file must be created after all physical files on which it is based are created. The PFILE keyword in the previous example is used to specify the physical file or files on which the logical file is based.

Record formats in a logical file can be:

- A new record format based on fields from a physical file
- The same record format as in a previously described physical or logical file (see "Sharing Existing Record Format Descriptions" on page 4-14)

Fields in the logical file record format must either appear in the record format of at least one of the physical files or be derived from the fields of the physical files on which the logical file is based.

For more information about describing logical files, see Chapter 6.

Additional Field Definition Functions

You can describe additional information about the fields in the physical and logical file record formats with function keywords (positions 45 through 80 on the *DDS Coding Form*). Some of the things you can specify include:

- Validity checking keywords to verify that the field data meets your standards. For example, you can describe a field to have a valid range of 500 to 900. (This checking is done only when data is typed on a keyboard to the display.)
- Editing keywords to control how a field should be displayed or printed. For example, you can use the EDTCDE(Y) keyword to specify that a date field is to appear as MM/DD/YY. The EDTCDE and EDTWRD keywords can be used to control editing. (This editing is done only when used in a display or printer file.)
- Documentation, heading, and name control keywords to control the description and name of a field. For example, you can use the TEXT keyword to document a description of each field. This text description is included in your compiler list to better document the files used in your program. The TEXT and COLHDG keywords control text and column-heading definitions. The ALIAS keyword can be used to provide a more descriptive name for a field. The alias, or alternative name, is used in a program (if the high-level language supports alias names).

Using Existing Field Descriptions and Field Reference Files

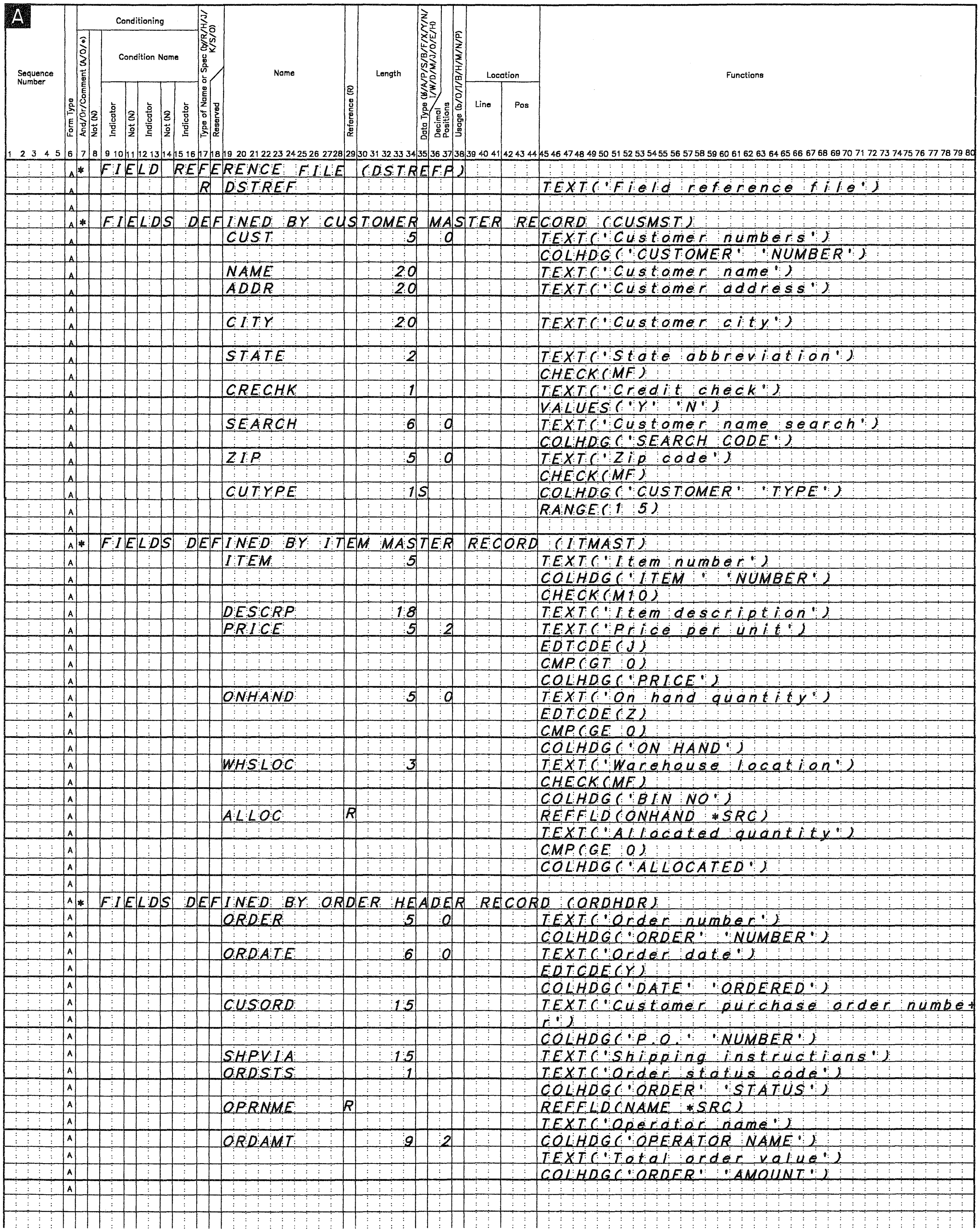
If a field was already described in an existing file, and you want to use that field description in a new file you are setting up, you can request the system to copy that description into your new file description. The DDS keywords REF and REFFLD allow you to refer to a field description in an existing file. This helps reduce the effort of coding DDS statements. It also helps ensure that the field attributes are used consistently in all files that use the field.

In addition, you can create a physical file for the sole purpose of using its field descriptions. That is, the file does not contain data; it is used only as a reference for the field descriptions for other files. This type of file is known as a field reference file. A **field reference file** is a physical file containing no data, just field descriptions.

You can use a field reference file to simplify record format descriptions and to ensure field descriptions are used consistently. You can define all the fields you need for an application or any group of files in a field reference file. You can create a field reference file using DDS and the Create Physical File (CRTPF) command.

After the field reference file is created, you can build physical file record formats from this file without describing the characteristics of each field in each file. When you build physical files, all you need to do is refer to the field reference file (using the REF and REFFLD keywords) and specify any changes. Any changes to the field descriptions and keywords specified in your new file override the descriptions in the field reference file.

In the following example, a field reference file named DSTREFP is created for distribution applications. Figure 4-4 on page 4-12 shows the DDS needed to describe DSTREFP.


A
Sequence Number
Form Type
Conditioning
And/Or/Comment (A/O/*)
Not (N)
Condition Name
Indicator
Not (N)
Indicator
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

```
     A* FIELD REFERENCE FILE (DSTREFP)
     A          R DSTREF                    TEXT('Field reference file')
     A
     A* FIELDS DEFINED BY CUSTOMER MASTER RECORD (CUSMST)
     A            CUST           5  0       TEXT('Customer numbers')
     A                                      COLHDG('CUSTOMER' 'NUMBER')
     A            NAME          20          TEXT('Customer name')
     A            ADDR          20          TEXT('Customer address')
     A
     A            CITY          20          TEXT('Customer city')
     A
     A            STATE          2          TEXT('State abbreviation')
     A                                      CHECK(MF)
     A            CRECHK         1          TEXT('Credit check')
     A                                      VALUES('Y' 'N')
     A            SEARCH         6  0       TEXT('Customer name search')
     A                                      COLHDG('SEARCH CODE')
     A            ZIP            5  0       TEXT('Zip code')
     A                                      CHECK(MF)
     A            CUTYPE         1S         COLHDG('CUSTOMER' 'TYPE')
     A                                      RANGE(1 5)
     A
     A* FIELDS DEFINED BY ITEM MASTER RECORD (ITMAST)
     A            ITEM           5          TEXT('Item number')
     A                                      COLHDG('ITEM ' 'NUMBER')
     A                                      CHECK(M10)
     A            DESCRP        18          TEXT('Item description')
     A            PRICE          5  2       TEXT('Price per unit')
     A                                      EDTCDE(J)
     A                                      CMP(GT 0)
     A                                      COLHDG('PRICE')
     A            ONHAND         5  0       TEXT('On hand quantity')
     A                                      EDTCDE(Z)
     A                                      CMP(GE 0)
     A                                      COLHDG('ON HAND')
     A            WHSLOC         3          TEXT('Warehouse location')
     A                                      CHECK(MF)
     A                                      COLHDG('BIN NO')
     A            ALLOC     R               REFFLD(ONHAND *SRC)
     A                                      TEXT('Allocated quantity')
     A                                      CMP(GE 0)
     A                                      COLHDG('ALLOCATED')
     A
     A* FIELDS DEFINED BY ORDER HEADER RECORD (ORDHDR)
     A            ORDER          5  0       TEXT('Order number')
     A                                      COLHDG('ORDER' 'NUMBER')
     A            ORDATE         6  0       TEXT('Order date')
     A                                      EDTCDE(Y)
     A                                      COLHDG('DATE' 'ORDERED')
     A            CUSORD        15          TEXT('Customer purchase order numbe+
     A                                      r')
     A                                      COLHDG('P.O.' 'NUMBER')
     A            SHPVIA        15          TEXT('Shipping instructions')
     A            ORDSTS         1          TEXT('Order status code')
     A                                      COLHDG('ORDER' 'STATUS')
     A            OPRNME    R               REFFLD(NAME *SRC)
     A                                      TEXT('Operator name')
     A            ORDAMT         9  2       COLHDG('OPERATOR NAME')
     A                                      TEXT('Total order value')
     A                                      COLHDG('ORDER' 'AMOUNT')
     A
```

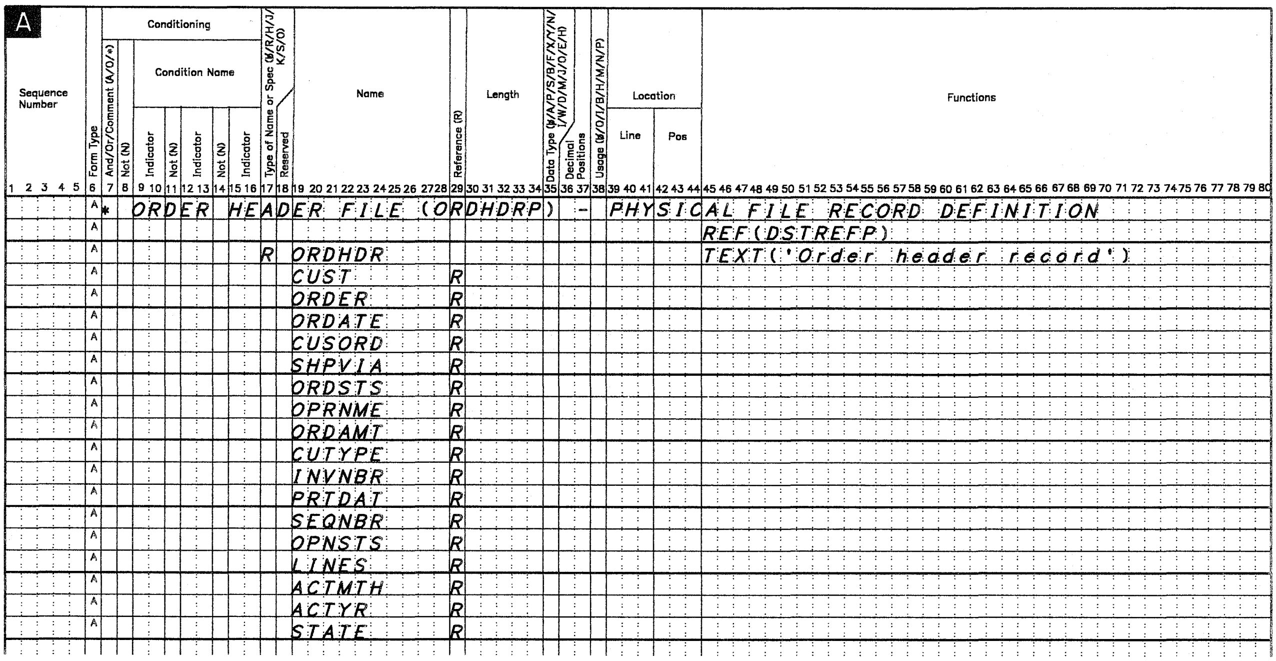
RSLH226-2

Figure 4-4. DDS for a Field Reference File (DSTREFP)

To then create a field reference file, use the Create Physical File (CRTPF) command. Assume that the DDS in Figure 4-4 on page 4-12 was entered into a source file FRSOURCE; the member name is DSTREFP. The parameter MBR(*NONE) tells the system not to add a member to the file (because the field reference file will never contain data and therefore does not need a member).

```
CRTPF FILE(DSTPRODLB/DSTREFP)
      SRCFILE(QGPL/FRSOURCE) MBR(*NONE)
      TEXT('Distribution field reference file')
```

To describe the physical file ORDHDRP by referring to DSTREFP, use the following DDS (Figure 4-5):



```
     A* ORDER HEADER FILE (ORDHDRP) - PHYSICAL FILE RECORD DEFINITION
     A                                      REF(DSTREFP)
     A          R ORDHDR                    TEXT('Order header record')
     A            CUST      R
     A            ORDER     R
     A            ORDATE    R
     A            CUSORD    R
     A            SHPVIA    R
     A            ORDSTS    R
     A            OPRNME    R
     A            ORDAMT    R
     A            CUTYPE    R
     A            INVNBR    R
     A            PRTDAT    R
     A            SEQNBR    R
     A            OPNSTS    R
     A            LINES     R
     A            ACTMTH    R
     A            ACTYR     R
     A            STATE     R
```

RSLH228-2

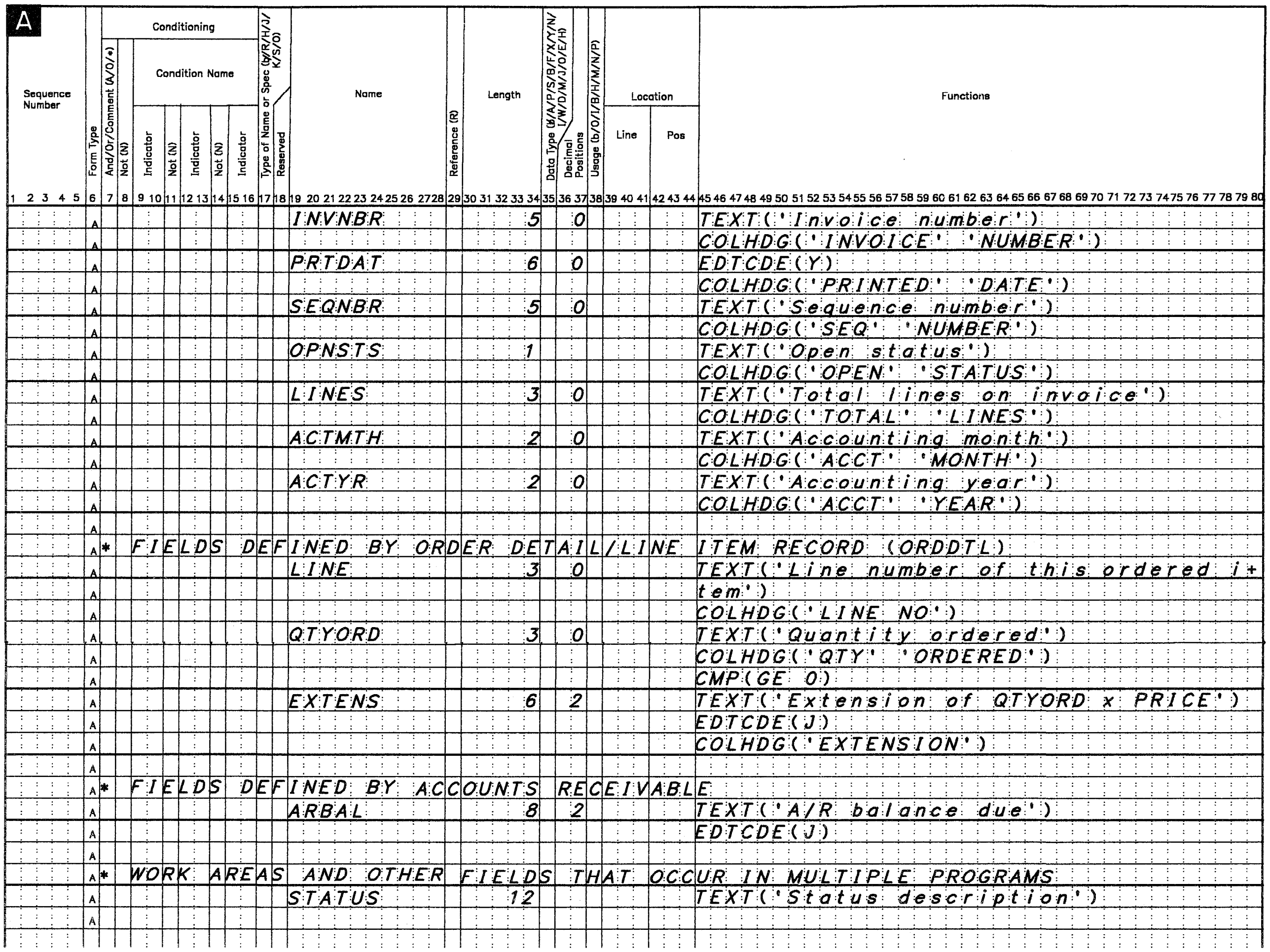
Figure 4-5. DDS for a Physical File (ORDHDRP) Built from a Field Reference File

The REF keyword (positions 45 through 80) with DSTREFP (the field reference file name) specified indicates the file from which field descriptions are to be used. The R in position 29 of each field indicates that the field description is to be taken from the reference file.

When you create the ORDHDRP file, the system uses the DSTREFP file to determine the attributes of the fields included in the ORDHDR record format. To create the ORDHDRP file, use the Create Physical File (CRTPF) command. Assume that the DDS in Figure 4-5 was entered into a source file QDDSSRC; the member name is ORDHDRP.

```
CRTPF FILE(DSTPRODLB/ORDHDRP)
      TEXT('Order Header physical file')
```

Note: The files used in some of the examples in this guide refer to this field reference file.


A
Sequence Number
Form Type
Conditioning
And/Or/Comment (A/O/*)
Not (N)
Condition Name
Indicator
Not (N)
Indicator
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

```
     A            INVNBR         5  0       TEXT('Invoice number')
     A                                      COLHDG('INVOICE' 'NUMBER')
     A            PRTDAT         6  0       EDTCDE(Y)
     A                                      COLHDG('PRINTED' 'DATE')
     A            SEQNBR         5  0       TEXT('Sequence number')
     A                                      COLHDG('SEQ' 'NUMBER')
     A            OPNSTS         1          TEXT('Open status')
     A                                      COLHDG('OPEN' 'STATUS')
     A            LINES          3  0       TEXT('Total lines on invoice')
     A                                      COLHDG('TOTAL' 'LINES')
     A            ACTMTH         2  0       TEXT('Accounting month')
     A                                      COLHDG('ACCT' 'MONTH')
     A            ACTYR          2  0       TEXT('Accounting year')
     A                                      COLHDG('ACCT' 'YEAR')
     A
     A* FIELDS DEFINED BY ORDER DETAIL/LINE ITEM RECORD (ORDDTL)
     A            LINE           3  0       TEXT('Line number of this ordered i+
     A                                      tem')
     A                                      COLHDG('LINE NO')
     A            QTYORD         3  0       TEXT('Quantity ordered')
     A                                      COLHDG('QTY' 'ORDERED')
     A                                      CMP(GE 0)
     A            EXTENS         6  2       TEXT('Extension of QTYORD x PRICE')
     A                                      EDTCDE(J)
     A                                      COLHDG('EXTENSION')
     A
     A* FIELDS DEFINED BY ACCOUNTS RECEIVABLE
     A            ARBAL          8  2       TEXT('A/R balance due')
     A                                      EDTCDE(J)
     A
     A* WORK AREAS AND OTHER FIELDS THAT OCCUR IN MULTIPLE PROGRAMS
     A            STATUS        12          TEXT('Status description')
     A
```

RSLH227-1

Using a Data Dictionary for Field Reference

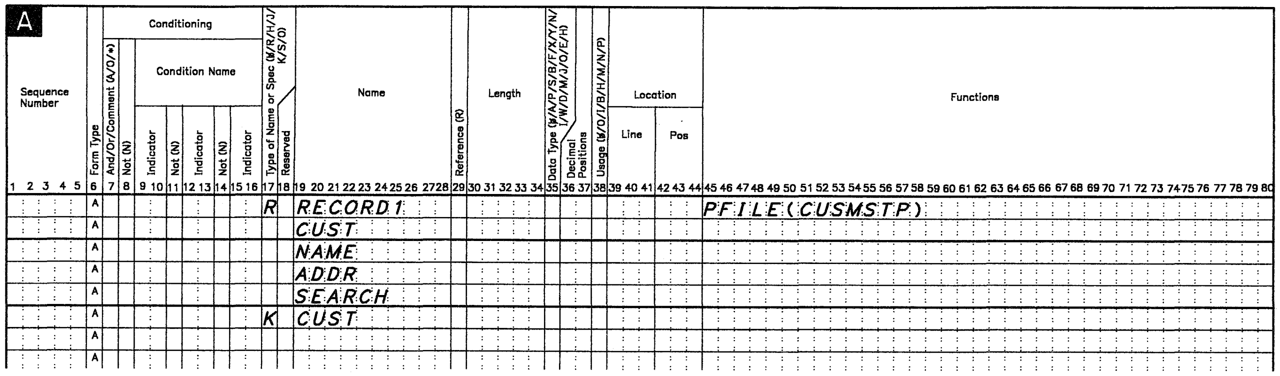
You can use a data dictionary and IDDU as an alternative to using a DDS field reference file. IDDU allows you to define fields in a data dictionary.

Sharing Existing Record Format Descriptions

A record format can be described once in either a physical or logical file (except a join logical file) and can be used by many files. When you describe a new file, you can specify the record format of an existing file is to be used by the new file. This can help reduce the number of DDS statements that you would normally code to describe a record format in a new file and can save auxiliary storage space.

The file originally describing the record format can be deleted without affecting the files sharing the record format. After the last file using the record format is deleted, the system automatically deletes the record format description.

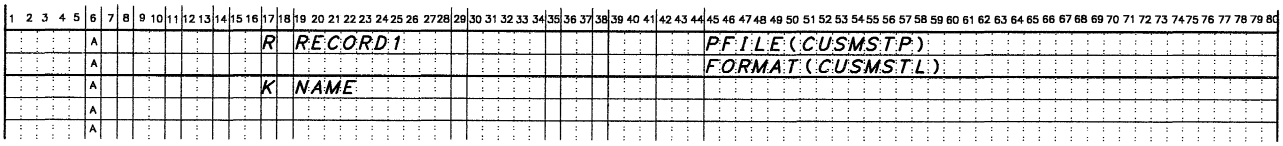
The following shows the DDS for two files. The first file describes a record format, and the second shares the record format of the first:


A
Sequence Number
Conditioning
Condition Name
Form Type
And/Or/Comment (A/O/*)
Not (N)
Indicator
Not (N)
Indicator
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

```
     A          R RECORD1                   PFILE(CUSMSTP)
     A            CUST
     A            NAME
     A            ADDR
     A            SEARCH
     A          K CUST
     A
     A
```

RSLH208-1

Figure 4-6. DDS for a Logical File (CUSMSTL)



```
     A          R RECORD1                   PFILE(CUSMSTP)
     A                                      FORMAT(CUSMSTL)
     A          K NAME
     A
     A
```

RSLH209-0

Figure 4-7. DDS for a Logical File (CUSTMSTL1) Sharing a Record Format

The example shown in Figure 4-6 shows file CUSMSTL, in which the fields *Cust*, *Name*, *Addr*, and *Search* make up the record format. The *Cust* field is specified as a key field.

The DDS in Figure 4-7 shows file CUSTMSTL1, in which the FORMAT keyword names CUSMSTL to supply the record format. The record format name must be RECORD1, the same as the record format name shown in Figure 4-6. Because the files are sharing the same format, both files have fields *Cust*, *Name*, *Addr*, and *Search* in the record format. In file CUSMSTL1, a different key field, *Name* is specified.

The following restrictions apply to shared record formats:

- A physical file cannot share the format of a logical file.
- A join logical file cannot share the format of another file, and another file cannot share the format of a join logical file.
- A view cannot share the format of another file, and another file cannot share the format of a view.

If the original record format is changed by deleting all related files and creating the original file and all the related files again, it is changed for all files that share it. If only the file with the original format is deleted and re-created with a new record format, all files previously sharing that file's format continue to use the original format.

If a logical file is defined but no field descriptions are specified and the FORMAT keyword is not specified, the record format of the first physical file (specified first on the PFILE keyword for the logical file) is automatically shared. The record format name specified in the logical file must be the same as the record format name specified in the physical file.

To find out if a file shares a format with another file, use the RCDFMT parameter on the Display Database Relations (DSPDBR) command.

Describing the Access Path for the File

An access path describes the order in which records are to be retrieved. Records in a physical or logical file can be retrieved using an arrival sequence access path or a keyed sequence access path. For logical files, you can also select and omit records based on the value of one or more fields in each record.

Arrival Sequence Access Path

The arrival sequence access path is based on the order in which the records arrive and are stored in the file. For reading or updating, records can be accessed:

- Sequentially, where each record is taken from the next sequential physical position in the file.
- Directly by relative record number, where the record is identified by its position from the start of the file.

An externally described file has an arrival sequence access path when no key fields are specified for the file.

An arrival sequence access path is valid only for the following:

- Physical files
- Logical files in which each member of the logical file is based on only one physical file member
- Join logical files
- Views

Notes:

1. Arrival sequence is the only processing method that allows a program to use the storage space previously occupied by a deleted record by placing another record in that storage space. For more information about processing deleted records, see "Deleting Database Records" on page 10-10.
2. Through your high-level language, the Display Physical File Member (DSPPFM) command, and the Copy File (CPYF) command, you can be able to process a keyed sequence file in arrival sequence. You can use this function for a physical file, a simple logical file based on one physical file member, or a join logical file.
3. Through your high-level language, you can be able to process a keyed sequence file directly by relative record number. You can use this function for a physical file, a simple logical file based on one physical file member, or a join logical file.
4. An arrival sequence access path does not take up any additional storage and is always saved or restored with the file. (Because the arrival sequence access path is nothing more than the physical order of the data as it was stored, when you save the data you save the arrival sequence access path.)

Keyed Sequence Access Path

A keyed sequence access path is based on the contents of the key fields as defined in DDS. This type of access path is updated whenever records are added or deleted, or when records are updated and the contents of a key field is changed. The keyed sequence access path is valid for both physical and logical files. The sequence of the records in the file is defined in DDS when the file is created and is maintained automatically by the system.

Key fields defined as character fields are arranged based on the sequence defined for EBCDIC characters. Key fields defined as numeric fields are arranged based on their algebraic values, unless the DDS UNSIGNED (unsigned value) or ABSVAL (absolute value) keywords are specified for the field. Key fields defined as DBCS are allowed, but are arranged only as single bytes based on their bit representation.

Arranging Key Fields in Ascending or Descending Sequence: Key fields can be arranged in either ascending or descending sequence. Consider the following records:

Record	Empnbr	Clsnbr	Clsnam	Cpdate
1	56218	412	Welding I	032188
2	41322	412	Welding I	011388
3	64002	412	Welding I	011388
4	23318	412	Welding I	032188
5	41321	412	Welding I	051888
6	62213	412	Welding I	032188

If the *Empnbr* field is the key field, the two possibilities for organizing these records are:

- In ascending sequence, where the order of the records in the access path is:

Record	Empnbr	Clsnbr	Clsnam	Cpdate
4	23318	412	Welding I	032188
5	41321	412	Welding I	051888
2	41322	412	Welding I	011388
1	56218	412	Welding I	032188
6	62213	412	Welding I	032188
3	64002	412	Welding I	011388

- In descending sequence, where the order of the records in the access path is:

Record	Empnbr	Clsnbr	Clsnam	Cpdate
3	64002	412	Welding !	011388
6	62213	412	Welding I	032188
1	56218	412	Welding I	032188
2	41322	412	Welding I	011388
5	41321	412	Welding I	051888
4	23318	412	Welding I	032188

When you describe a key field, the default is ascending sequence. However, you can use the DESCEND DDS keyword to specify that you want to arrange a key field in descending sequence.

Using More Than One Key Field: You can use more than one key field to arrange the records in a file. The key fields do not have to use the same sequence. For example, when you use two key fields, one field can use ascending sequence while the other can use descending sequence. Consider the following records:

Record	Order	Ordate	Line	Item	Qtyord	Extens
1	52218	063088	01	88682	425	031875
2	41834	062888	03	42111	30	020550
3	41834	062888	02	61132	4	021700
4	52218	063088	02	40001	62	021700
5	41834	062888	01	00623	50	025000

If the access path uses the *Order* field, then the *Line* field as the key fields, both in ascending sequence, the order of the records in the access path is:

Record	Order	Ordate	Line	Item	Qtyord	Extens
5	41834	062888	01	00623	50	025000
3	41834	062888	02	61132	4	021700
2	41834	062888	03	42111	30	020550
1	52218	063088	01	88682	425	031875
4	52218	063088	02	40001	62	021700

If the access path uses the key field *Order* in ascending sequence, then the *Line* field in descending sequence, the order of the records in the access path is:

Record	Order	Ordate	Line	Item	Qtyord	Extens
2	41834	062888	03	42111	30	020550
3	41834	062888	02	61132	4	021700
5	41834	062888	01	00623	50	025000
4	52218	063088	02	40001	62	021700
1	52218	063088	01	88682	425	031875

When a record has key fields whose contents are the same as the key field in another record in the same file, then the file is said to have records with duplicate key values. However, the duplication must occur for *all* key fields for a record if they are to be called duplicate key values. For example, if a record format has two key fields *Order* and *Ordate*, duplicate key values occur when the contents of both the *Order* and *Ordate* fields are the same in two or more records. These records have duplicate key values:

Order	Ordate	Line	Item	Qtyord	Extens
41834	062888	03	42111	30	020550
41834	062888	02	61132	04	021700
41834	062888	01	00623	50	025000

Using the *Line* field as a third key field defines the file so that there are no duplicate keys:

(***First Key Field***) **Order**	(***Second Key Field***) **Ordate**	(***Third Key Field***) **Line**	**Item**	**Qtyord**	**Extens**
41834	062888	03	42111	30	020550
41834	062888	02	61132	04	021700
41834	062888	01	00623	50	025000

A logical file that has more than one record format can have records with duplicate key values, even though the record formats are based on different physical files. That is, even though the key values come from different record formats, they are considered duplicate key values.

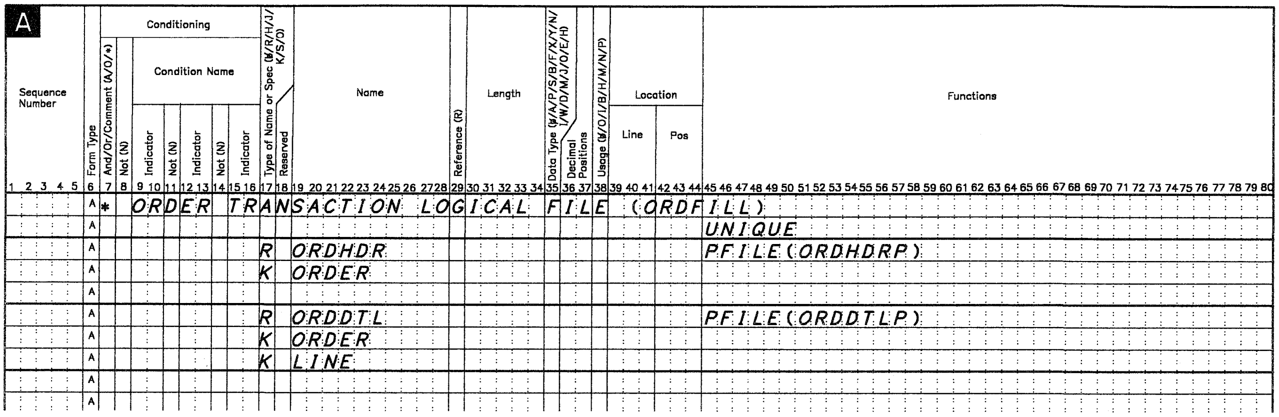
Preventing Duplicate Key Values: The AS/400 database management system allows records with duplicate key values in your files. However, you may want to prevent duplicate key values in some of your files. For example, you can create a file where the key field is defined as the customer number field. In this case, you want the system to ensure that each record in the file has a unique customer number.

You can prevent duplicate key values in your files by specifying the UNIQUE keyword in DDS. With the UNIQUE keyword specified, a record cannot be entered or copied into a file if its key value is the same as the key value of a record already existing in the file.

If records with duplicate key values already exist in a physical file, the associated logical file cannot have the UNIQUE keyword specified. If you try to create a logical file with the UNIQUE keyword specified, and the associated physical file contains duplicate key values, the logical file is not created. The system sends you a message stating this and sends you messages (as many as 20) indicating which records contain duplicate key values.

When the UNIQUE keyword is specified for a file, any record added to the file cannot have a key value that duplicates the key value of an existing record in the file, regardless of the file used to add the new record. For example, two logical files LF1 and LF2 are based on the physical file PF1. The UNIQUE keyword is specified for LF1. If you use LF2 to add a record to PF1, you cannot add the record if it causes a duplicate key value in LF1.

The following shows the DDS for a logical file that requires unique key values:



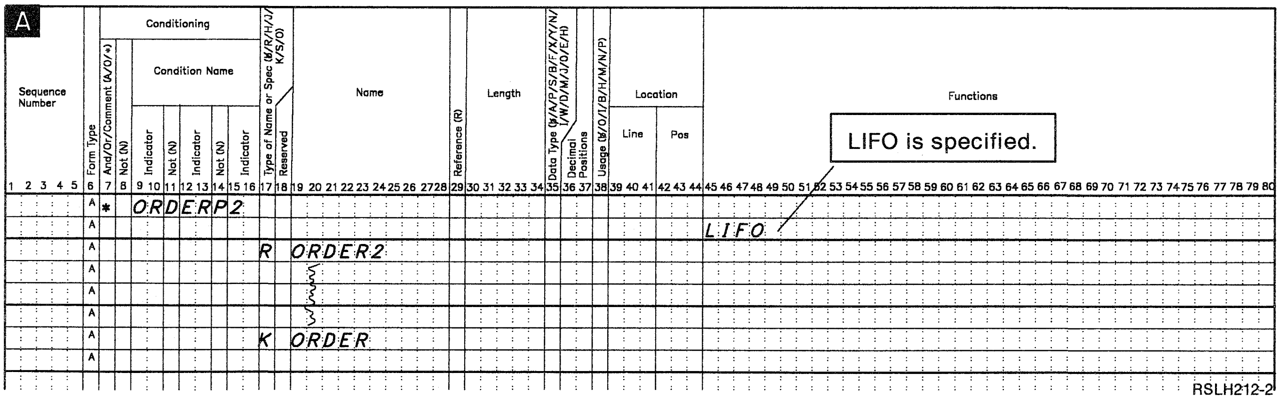
```
     A* ORDER TRANSACTION LOGICAL FILE (ORDFILL)
     A                                      UNIQUE
     A          R ORDHDR                    PFILE(ORDHDRP)
     A          K ORDER
     A
     A          R ORDDTL                    PFILE(ORDDTLP)
     A          K ORDER
     A          K LINE
     A
     A
```

RSLH210-1

In this example, the contents of the key fields (the *Order* field for the ORDHDR record format, and the *Order* and *Line* fields for the ORDDTL record format) must be unique whether the record is added through: the ORDHDRP file , the ORDDTLP file, or the logical file defined here. With the *Line* field specified as a second key field in the ORDDTL record format, the same value can exist in the *Order* key field in both physical files. Because the physical file ORDDTLP has two key fields and the physical file ORDHDRP has only one, the key values in the two files do not conflict.

Arranging Duplicate Keys: If you do not specify the UNIQUE keyword in DDS, you can specify how the system is to store records with duplicate key values, should they occur. You specify that records with duplicate key values are stored in the access path in one of the following ways:

- Last-in-first-out (LIFO). When the LIFO keyword is specified, records with duplicate key values are retrieved in last-in-first-out order by the physical sequence of the records. Below is an example of DDS using the LIFO keyword.



```
     A* ORDERP2
     A                                      LIFO
     A          R ORDER2
     A
     A
     A
     A          K ORDER
     A
```

LIFO is specified.

RSLH212-2

- First-in-first-out (FIFO). If the FIFO keyword is specified, records with duplicate key values are retrieved in first-in-first-out order by the physical sequence of the records.

- First-changed-first-out (FCFO). If the FCFO keyword is specified, records with duplicate key values are retrieved in first-changed-first-out order by the physical sequence of the keys.
- No specific order for duplicate key fields (the default). When the FIFO, FCFO, or LIFO keywords are not specified, no guaranteed order is specified for retrieving records with duplicate keys. No specific order for duplicate key fields allows more access path sharing, which can improve performance. For more information about access path sharing, see "Using Existing Access Paths" on page 6-14.

When a simple- or multiple-format logical file is based on more than one physical file member, records with duplicate key values are read in the order in which the files and members are specified on the DTAMBRS parameter on the Create Logical File (CRTLF) or Add Logical File Member (ADDLFM) command. Examples of logical files with more than one record format can be found in the *DDS Reference*.

The LIFO or FIFO order of records with duplicate key values is not determined by the sequence of updates made to the contents of the key fields, but solely by the physical sequence of the records in the file member. Assume that a physical file has the FIFO keyword specified (records with duplicate keys are in first-in-first-out order), and that the following shows the order in which records were added to the file:

Order Records Were Added to File	**Key Value**
1	A
2	B
3	C
4	C
5	D

The sequence of the access path is (FIFO, ascending key):

Record Number	**Key Value**
1	A
2	B
3	C
4	C
5	D

Records 3 and 4, which have duplicate key values, are in FIFO order. That is, because record 3 was added to the file before record 4, it is read before record 4. This would become apparent if the records were read in descending order. This could be done by creating a logical file based on this physical file, with the DESCEND keyword specified in the logical file.

The sequence of the access path is (FIFO, descending key):

Record Number	**Key Value**
5	D
3	C
4	C

Record Number	Key Value
2	B
1	A

If physical record 1 is changed such that the key value is C, the sequence of the access path for the physical file is (FIFO, ascending key):

Record Number	Key Value
2	B
1	C
3	C
4	C
5	D

Finally, changing to descending order, the new sequence of the access path for the logical file is (FIFO, descending key):

Record Number	Key Value
5	D
1	C
3	C
4	C
2	B

After the change, record 1 does not appear after record 4, even though the contents of the key field were updated after record 4 was added.

The FCFO order of records with duplicate key values is determined by the sequence of updates made to the contents of the key fields. In the example above, after record 1 is changed such that the key value is C, the sequence of the access path (FCFO, ascending key only) is:

Record Number	Key Value
2	B
3	C
4	C
1	C
5	D

For FCFO, the duplicate key ordering can change when the FCFO access path is rebuilt or when a rollback operation is performed. In some cases, your key field can change but the physical key does not change. In these cases, the FCFO ordering does not change, even though the key field has changed. For example, when the index ordering is changed to be based on the absolute value of the key, the FCFO ordering does not change. The physical value of the key does not change even though your key changes from negative to positive. Because the physical key does not change, FCFO ordering does not change.

Using Existing Access Path Specifications

You can use the DDS keyword REFACCPTH to use another file's access path specifications. When the file is created, the system determines which access path to share. The file using the REFACCPTH keyword does not necessarily share the access path of the file specified in the REFACCPTH keyword. The REFACCPTH keyword is used to simply reduce the number of DDS statements that must be specified. That is, rather than code the key field specifications for the file, you can specify the REFACCPTH keyword. When the file is created, the system copies the key field and select/omit specifications from the file specified on the REFACCPTH keyword to the file being created.

Using Floating Point Fields in Access Paths

The collating sequence for records in a keyed database file depends on the presence of the SIGNED, UNSIGNED, and ABSVAL DDS keywords. For floating-point fields, the sign is the farthest left bit, the exponent is next, and the significant is last. The collating sequence with UNSIGNED specified is:

- Positive real numbers—positive infinity
- Negative real numbers—negative infinity

A floating-point key field with the SIGNED keyword specified, or defaulted to, on the DDS has an algebraic numeric sequence. The collating sequence is negative infinity—real numbers—positive infinity.

A floating-point key field with the ABSVAL keyword specified on the DDS has an absolute value numeric sequence.

The following floating-point collating sequences are observed:

- Zero (positive or negative) collates in the same manner as any other positive/negative real number.
- Negative zero collates before positive zero for SIGNED sequences.
- Negative and positive zero collate the same for ABSVAL sequences.

You cannot use not-a-number (*NAN) values in key fields. If you attempt this, and a *NAN value is detected in a key field during file creation, the file is not created.

Creating Database Files

The system supports several methods for creating a database file:

- OS/400 IDDU
- Structured Query Language/400
- OS/400 control language (CL)

You can create a database file using IDDU. If you are using IDDU to describe your database files, you might also consider using it to create your files.

You can create a database file using the SQL/400 statements. SQL/400 is the IBM Systems Application Architecture relational database language, and can be used on the AS/400 system to interactively describe and create database files.

You can also create a database file using CL. The CL database file create commands are: Create Physical File (CRTPF), Create Logical File (CRTLF), and Create Source Physical File (CRTSRCPF).

Because additional system functions are available with CL, this guide focuses on creating files using CL.

Specifying Database File and Member Attributes

When you create a database file, database attributes are stored with the file and members. Some of these file and member attributes are discussed in this section. For a complete discussion about how to specify these attributes and parameters, see the CRTPF, CRTLF, CRTSRCPF, ADDPFM, ADDLFM, CHGPF, CHGLF, CHGPFM, CHGSRCPF, and CHGLFM commands in the *CL Reference*.

File and Member Name

FILE and MBR Parameters. When you create a file, you identify the name of the file on the create command (on the FILE parameter). You also identify the library in which the file will reside. When you create a physical or logical file, the system normally creates a member by the same name as the file. You can, however, specify your own member name on the MBR parameter on the create commands. You can also tell the system not to create any members when the file is created by specifying MBR(*NONE) on the create commands.

Note: When you create a source physical file, the system does *not* automatically create a member.

You can control the order in which physical file members are read using a logical file. You can use the DTAMBRS parameter of the Create Logical File (CRTLF) command to specify the order the physical file members are to be read. For more information about using logical files in this way, see “Logical File Members” on page 6-20.

Source File and Member

SRCFILE and SRCMBR Parameters. If the file being created is using DDS source statements to describe the file, the SRCFILE and SRCMBR parameters specify which source file and member, respectively, contains the DDS. If you do not specify a name, the default source file name is QDDSSRC, and the default member name is the same name you specified on the FILE parameter.

Type of Database File

FILETYPE Parameter. When you create a file, you can specify whether the file is a data file or a source file. The Create Physical File (CRTPF) and Create Logical File (CRTLF) commands use the default data file type (*DATA). The Create Source Physical File (CRTSRCPF) command uses the default source file type (*SRC).

Number of Members Allowed in the File

MAXMBRS Parameter. When you create a file, you can also specify the maximum number of members the file can hold. The MAXMBRS parameter is used to control this value. The default maximum number of members for physical and logical files is one, and the default for source physical files is *NOMAX.

Where the Data Is to Be Stored

UNIT Parameter. You need not specify the location of data in auxiliary storage. The system automatically finds a place for the file on auxiliary storage. However, if you wish to specify which disk you want the file located on, specify the UNIT parameter. The UNIT parameter specifies the location of the data records in physical files and the access path for both physical files and logical files.

If there is not enough space on the unit, or if the unit specified is not valid for your system, the data and access paths are placed on different units. An informational message indicating that the file was not placed on the requested unit is sent at the time the file members are added. (A message is *not* sent when the file member is extended.)

In general, you should not specify the UNIT parameter. The suggested approach is to allow the system to place the file on the disk unit of its choosing. This is normally a better approach for performance, and also relieves you of the task of managing auxiliary storage.

Frequency of Writing Data to Auxiliary Storage

FRCRATIO Parameter. Normally, the system determines when to write changed data from main storage to auxiliary storage. If you want to control when database changes are written to auxiliary storage, you can do so using the force write ratio (FRCRATIO) parameter on either the create, change, or override database file commands.

If you do not specify that records are to be forced to auxiliary storage, the system determines when the records are written. Closing the file (except for a shared close) and the force-end-of-data operation automatically forces remaining updates, deletions, and additions to auxiliary storage. If you are journaling the file, the FRCRATIO parameter should normally be *NONE.

Using the FRCRATIO parameter has performance and recovery considerations for your system. To understand these considerations, see Chapter 16 in this guide.

Frequency of Writing the Access Path to Auxiliary Storage

FRCACCPTH Parameter. The force access path (FRCACCPTH) parameter controls when an access path is written to auxiliary storage.

Specifying FRCACCPTH(*YES) forces the access path to auxiliary storage whenever the access path is changed. This can reduce the chance that the access path will need to be rebuilt should the system fail, because the system has a copy of the access path in auxiliary storage.

Specifying FRCACCPTH(*YES) can degrade performance when changes occur to the access path. An alternative to forcing the access path is journaling the access path. For more information about forcing access paths and journaling access paths, see Chapter 16 in this guide.

Checking for Changes to the Record Format Description

LVLCHK Parameter. The system checks, when the file is opened, if the definition of the database file you are using was changed to an extent that your program may not be able to process the file. The system normally notifies your program of this condition. For example, assume you compiled your program two months ago and, at that time, the file the program was using was defined as having three fields in each record. Last week another programmer decided to add a new field to the record format, so that now each record would have four fields. The system will notify your program, when it tries to open the file, that a significant change occurred to the definition of the file since the last time the program was compiled. This notification is known as a record format level check. You can specify if you want level checking when you create a file or using one of the change database file commands (the default is to do level checking). You can override the system and ignore the level check using the Override with Database File (OVRDBF) command.

Keeping the Access Path Current

MAINT Parameter. While a file is open, the system maintains the access paths as changes are made to the data in the file. However, because more than one access path can exist for the same data, changing data in one file might cause changes to be made in access paths for other files that are not currently open (in use). The three ways of maintaining access paths of files that are not currently open are:

- Immediate
- Rebuild
- Delayed

Immediate maintenance of an access path means that the access path is maintained as changes are made to its associated data, regardless if the file is open.

Rebuild maintenance of an access path means that the access path is only maintained while the file is open, not when the file is closed; the access path is rebuilt when the file is opened the next time. When a file with rebuild maintenance is closed, the system stops maintaining the access path. When the file is opened again, the access path is totally rebuilt. If one or more programs has opened a specific file member with rebuild maintenance specified, the system maintains the access path for that member until the last user closes the file member.

Delayed maintenance of an access path means that any maintenance for the access path is done after the file member is opened the next time and while it remains open. However, the access path is not rebuilt as it is with rebuild maintenance. Updates to the access path are collected from the time the member is closed until it is opened again. When it is opened, only the collected changes are merged into the access path.

Delayed maintenance can be faster than rebuilding the entire access path (as you would if you specify rebuild maintenance). However, if the collected changes to the access path exceed approximately 25% of the number of entries in the access path, the changes are no longer accumulated and the system rebuilds the access path at the next open (because the system can usually rebuild the entire access path faster than merging many collected changes into it). You should use delayed maintenance for files that have relatively few changes to the access path while the file members are closed. Delayed maintenance reduces system overhead by reducing the number of access paths that are maintained immediately. It also does not require the time needed to totally rebuild the access path when the member is opened.

The type of access path maintenance to specify depends on the number of records and the frequency of additions, deletions, and updates to a file while the file is closed. If you do not specify the type of maintenance for a file, the default is immediate maintenance.

Immediate maintenance decreases the open time, but increases the amount of time required to update records that affect changes to the access path while it is closed. You may want to specify immediate maintenance for access paths that are used frequently, or when you cannot wait for an access path to be rebuilt when the file is opened. You may want to specify delayed maintenance for access paths that are not used frequently, if infrequent changes are made to the record keys that make up the access path.

In general, for files used interactively, immediate maintenance results in good response time. For files used in batch jobs, either immediate, delayed, or rebuild maintenance is adequate, depending on the size of the members and the frequency of changes.

The following is a comparison of immediate, rebuild, and delayed maintenance as they affect opening and processing files:

Immediate	**Rebuild**	**Delayed**
Fast open because the access path is current.	Slow open because access path must be rebuilt.	Moderately fast open because the access path does not have to be rebuilt, but it must still be changed. Slow open if extensive changes are needed.
Slower update/output operations when many access paths with immediate maintenance are built over changing data (the system must maintain the access paths).	Faster update/output operations when many access paths with rebuild maintenance are built over changing data and are not open (the system does not have to maintain the access paths).	Moderately fast update/output operations when many access paths with delayed maintenance are built over changing data and are not open, (the system records the changes, but the access path itself is not maintained).

Notes:

1. Delayed or rebuild maintenance cannot be specified for a file that has unique keys.
2. Rebuild maintenance cannot be specified for a file if its access path is being journaled.

Recovering an Access Path If the System Fails

RECOVER Parameter. If the system fails, the access paths that had changes that were not forced to auxiliary storage when the system failed or that were not journaled cannot be used until they are rebuilt. Using the RECOVER parameter on the Create Physical File (CRTPF), the Create Logical File (CRTLF), and the Create Source Physical File (CRTSRCPF) commands, you can specify when that access path is to be rebuilt. Access paths are rebuilt either during the initial program load (IPL), after the IPL, or when a file is opened.

The following shows the relationship among duplicate key options, maintenance options, and recovery options:

Duplicate Key Options	Maintenance Options	Recovery Options
Unique	Immediate	Rebuild during the IPL (*IPL)
		Rebuild after the IPL (*AFTIPL, default)
		Do not rebuild at IPL, wait for first open (*NO).
Not unique	Immediate or delayed	Rebuild during the IPL (*IPL)
		Rebuild after the IPL (*AFTIPL)
		Do not rebuild at IPL, wait for first open (*NO, default)
Not unique	Rebuild	Do not rebuild at IPL, wait for first open (*NO, default)

If access paths must be recovered, a list of those files whose access paths need to be recovered is shown on the Override Access Path Recovery display at the next IPL. At that time, you can override the recovery option that was originally specified for the file by taking the appropriate option on the display.

Sharing a File in the Same Job

SHARE Parameter. By default, the database system lets a file be accessed and changed at the same time by many users. The SHARE parameter allows even closer sharing by sharing opened files in the same job. For example, sharing a file in the same job allows programs in the same job to share a file's status, record position, and buffer. Sharing files in the same job can improve performance by reducing the amount of storage the job needs and by reducing the time it takes to open and close the file. For more information about sharing files in the same job, see "Sharing Database Files in the Same Job" on page 8-7.

Specifying Wait Time for a Locked File or Record

WAITFILE and WAITRCD Parameter. When you create a file, you can specify how long a program should wait for either the file or a record in the file if another job has the file or record locked. If the wait time ends before the file or record is released, a message is sent to the program indicating that the job was not able to use the file or read the record. For more information about record and file locks and wait times, see "Record Locks" on page 8-6 and "File Locks" on page 8-6.

Specifying Public Authority

AUT Parameter. When you create a file, you can specify public authority. Public authority is the authority a user has to a file (or other object on the system) if that user does not have specific authority for the file or does not belong to a group with specific authority for the file. For more information about public authority, see "Public Authority" on page 7-3.

Specifying the System on Which the File Is Created

SYSTEM Parameter. You can specify if the file is to be created on the local system or a remote system that supports DDM. For more information about DDM, see the *DDM User's Guide*.

Specifying File and Member Text

TEXT Parameter. You can specify a text description for each file and member you create. The text data is useful in describing information about your file and members.

Chapter 5. Setting Up Physical Files

This chapter discusses some of the unique considerations for describing, then creating, a physical file.

For information about describing a physical file record format, see "Example of Describing a Physical File Using DDS" on page 4-6.

For information about describing a physical file access path, refer to "Describing the Access Path for the File" on page 4-16.

Creating a Physical File

To create a physical file, take the following steps:

1. If you are using DDS, enter DDS for the physical file into a source file. This can be done using the AS/400 Application Development Tools source entry utility (SEU). See "Working with Source Files" on page 17-4, for more information about how source statements are entered in source files.
2. Create the physical file. You can use the Create Physical File (CRTPF) command, or the Create Source Physical File (CRTSRCPF) command.

The following command creates a one-member file using DDS and places it in a library called DSTPRODLB.

```
CRTPF FILE(DSTPRODLB/ORDHDRP)
      TEXT('Order header physical file')
```

As shown, this command uses defaults. For the SRCFILE and SRCMBR parameters, the system uses DDS in the source file called QDDSSRC and the member named ORDHDRP (the same as the file name). The file ORDHDRP with one member of the same name is placed in the library DSTPRODLB.

Specifying Physical File and Member Attributes

Some of the attributes you can specify for physical files on the Create Physical File (CRTPF), Create Source Physical File (CRTSRCPF), Change Physical File (CHGPF), Change Source Physical File (CHGSRCPF), Add Physical File Member (ADDPFM), and Change Physical File Member (CHGPFM) commands include (command names are given in parentheses):

Expiration Date

EXPDATE Parameter. This parameter specifies an expiration date for each member in the file (ADDPFM, CHGPFM, CRTPF, CHGPF, CRTSRCPF, and CHGSRCPF commands). If the expiration date is past, the system operator is notified when the file is opened. The system operator can then override the expiration date and continue, or stop the job. Each member can have a different expiration date, which is specified when the member is added to the file. (The expiration date check can be overridden; see "Checking for the File's Expiration Date" on page 8-5.)

Size of the Physical File Member

SIZE Parameter. This parameter specifies the maximum number of records that can be placed in each member (CRTPF, CHGPF, CRTSRCPF, AND CHGSRCPF commands). The following formula can be used to determine the maximum:

R + (I * N)

where:

R	is the starting record count
I	is the number of records (increment) to add each time
N	is the number of times to add the increment

The defaults for the SIZE parameter are:

R	10,000
I	1,000
N	3 (CRTPF command) 499 (CRTSRCPF command)

For example, assume that R is a file created for 5000 records plus 3 increments of 1000 records each. The system can add 1000 to the initial record count of 5000 three times to make the total maximum 8000. When the total maximum is reached, the system operator either stops the job or tells the system to add another increment of records and continue. When increments are added, a message is sent to the system history log.

Instead of taking the default size or specifying a size, you can specify *NOMAX. For information about the maximum number of records allowed in a file, see Appendix A.

Storage Allocation

ALLOCATE Parameter. This parameter controls the storage allocated for members when they are added to the file (CRTPF, CHGPF, CRTSRCPF, and CHGSRCPF commands). The storage allocated would be large enough to contain the initial record count for a member. If you do not allocate storage when the members are added, the system will automatically extend the storage allocation as needed. You can use the ALLOCATE parameter only if you specified a maximum size on the SIZE parameter. If SIZE(*NOMAX) is specified, then ALLOCATE(*YES) cannot be specified.

Method of Allocating Storage

CONTIG Parameter. This parameter controls the method of allocating physical storage for a member (CRTPF and CRTSRCPF commands). If you allocate storage, you can request that the storage for the starting record count for a member be contiguous. That is, all the records in a member are to physically reside together. If there is not enough contiguous storage, contiguous storage allocation is not used and an informational message is sent to the job that requests the allocation, at the time the member is added.

Note: When a physical file is first created, the system always tries to allocate its initial storage contiguously. The only difference between using CONTIG(*NO) and CONTIG(*YES) is that with CONTIG(*YES) the system sends a message to the job log if it is unable to allocate contiguous storage when the file is created. No message is sent when a file is extended regardless of what you specified on the CONTIG parameter.

Record Length

RCDLEN Parameter. This parameter specifies the length of records in the file (CRTPF and CRTSRCPF commands). If the file is described to the record level only, then you specify the RCDLEN parameter when the file is created. This parameter cannot be specified if the file is described using DDS, IDDU, or SQL/400 (the system automatically determines the length of records in the file from the field level descriptions).

Deleted Records

DLTPCT Parameter. This parameter specifies the percentage of deleted records a file can contain before you want the system to send a message to the system history log (CRTPF, CHGPF, CRTSRCPF, and CHGSRCPF commands). When a file is closed, the system checks the member to determine the percentage of deleted records. If the percentage exceeds that value specified in the DLTPCT parameter, a message is sent to the history log. (For information about processing the history log, see the chapter on message handling in the *CL Programmer's Guide*.) One reason you might want to know when a file reaches a certain percentage of deleted records is to reclaim the space used by the deleted records. After you receive the message about deleted records, you could run the Reorganize Physical File Member (RGZPFM) command to reclaim the space. (For more information about RGZPFM, see "Reorganizing Data in Physical File Members" on page 13-3.) You can also specify to bypass the deleted records check by using the *NONE value for the DLTPCT parameter. *NONE is the default for the DLTPCT parameter.

Physical File Capabilities

ALWUPD and ALWDLT Parameters. File capabilities are used to control which input/output operations are allowed for a database file independent of database authority. For more information about database file capabilities and authority, see Chapter 7.

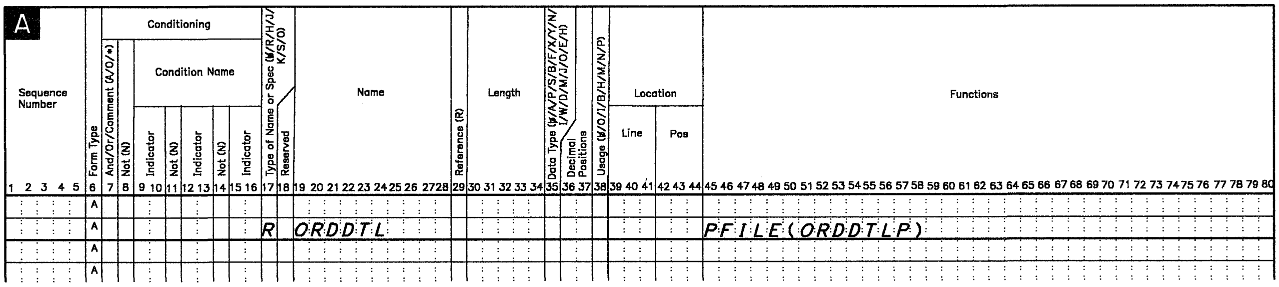
Chapter 6. Setting Up Logical Files

This chapter discusses some of the unique considerations for describing, then creating, a logical file. Many of the rules for setting up logical files apply to all categories of logical files. In this guide, rules that apply only to one category of logical file identify which category they refer to. Rules that apply to all categories of logical files do not identify the specific categories they apply to.

Describing Logical File Record Formats

For every logical file record format described with DDS, you must specify a record format name and either the PFILE keyword (for simple and multiple format logical files), or the JFILE keyword (for join logical files). The file names specified on the PFILE or JFILE keyword are the physical files that the logical file is based on. A simple or multiple-format logical file record format can be specified with DDS in any one of the following ways:

1. In the simple logical file record format, specify only the record format name and the PFILE keyword. The record format for the only (or first) physical file specified on the PFILE keyword is the record format for the logical file. The record format name specified in the logical file must be the same as the record format name in the only (or first) physical file.

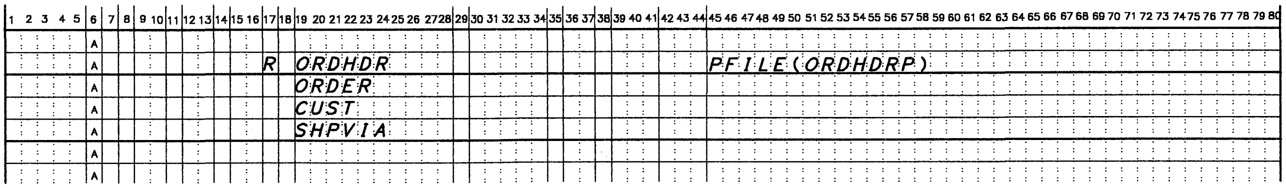

A
Sequence Number
Form Type
Conditioning
And/Or/Comment (A/O/*)
Condition Name
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

```
     A
     A          R ORDDTL                    PFILE(ORDDTLP)
     A
     A
```

RSLH234-1

Figure 6-1. Simple Logical File

2. In the following example, you describe your own record format by listing the field names you want to include. You can specify the field names in a different order, rename fields using the RENAME keyword, combine fields using the CONCAT keyword, and use specific positions of a field using the SST keyword. You can also override attributes of the fields by specifying different attributes in the logical file.



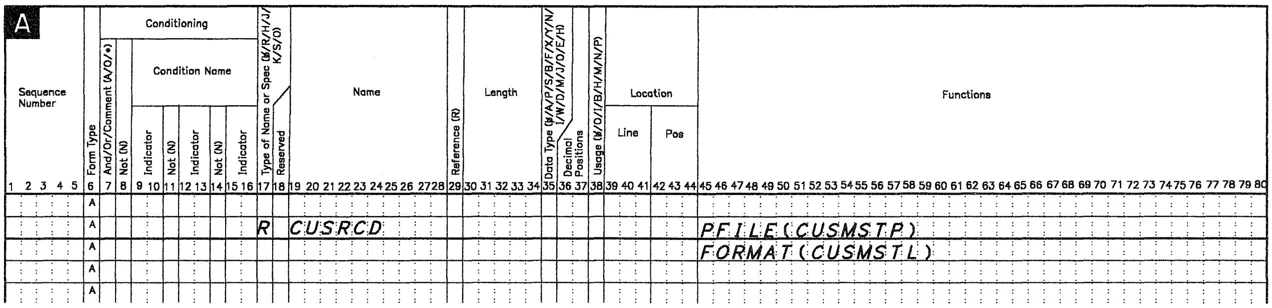
```
     A
     A          R ORDHDR                    PFILE(ORDHDRP)
     A            ORDER
     A            CUST
     A            SHPVIA
     A
     A
```

RSLH236-0

Figure 6-2. Simple Logical File with Fields Specified

3. In the following example, the file name specified on the FORMAT keyword is the name of a database file. A record format will be copied from this database file

for the logical file record format being described. The file name can be qualified by a library name. If a library name is not specified, the library list is used to find the file. The file must exist when the file you are describing is created. In addition, the record format name you specify in the logical file must be the same as one of the record format names in the file you specify on the FORMAT keyword.



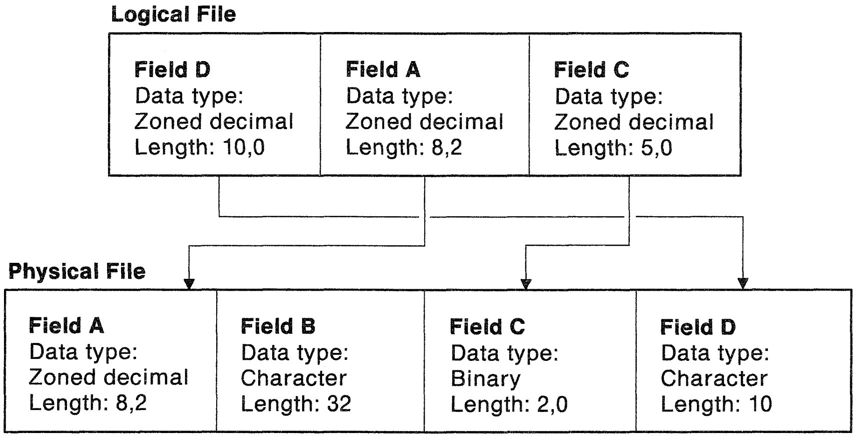
```
     A
     A          R CUSRCD                    PFILE(CUSMSTP)
     A                                      FORMAT(CUSMSTL)
     A
     A
```

RSLH235-1

In the following example, a program needs:

- The fields placed in a different order.
- A subset of the fields from the physical file.
- The data types changed for some fields.
- The field lengths changed for some fields.

You can use a logical file to make these changes.



Logical File

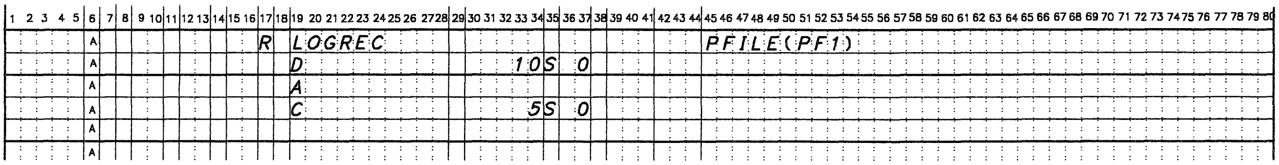
Field D Data type: Zoned decimal Length: 10,0	Field A Data type: Zoned decimal Length: 8,2	Field C Data type: Zoned decimal Length: 5,0

Physical File

Field A Data type: Zoned decimal Length: 8,2	Field B Data type: Character Length: 32	Field C Data type: Binary Length: 2,0	Field D Data type: Character Length: 10

RSLH205-1

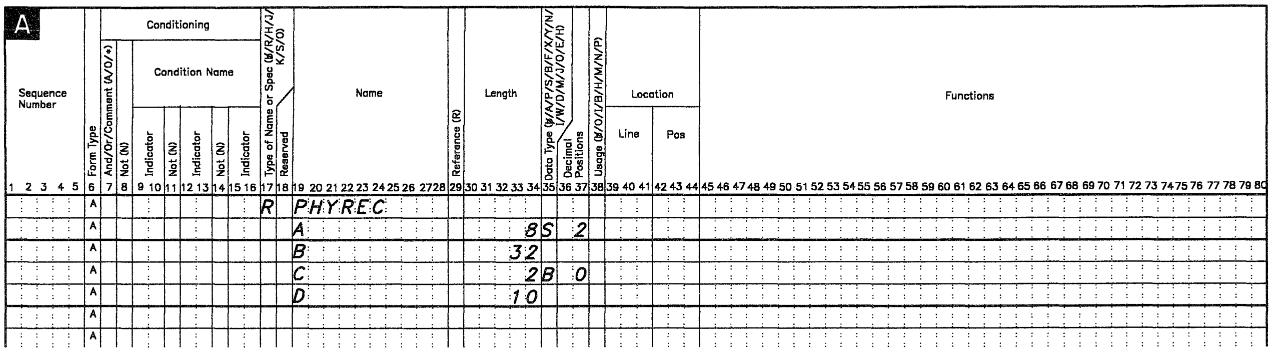
For the logical file, the DDS would be:



```
     A          R LOGREC                    PFILE(PF1)
     A            D             10S 0
     A            A
     A            C              5S 0
     A
     A
```

RSLH206-0

For the physical file, the DDS would be:



```
     A          R PHYREC
     A            A              8S 2
     A            B             32
     A            C              2B 0
     A            D             10
     A
     A
```

RSLH207-1

When a record is read from the logical file, the fields from the physical file are changed to match the logical file description. If the program updates or adds a record, the fields are changed back. For an add or update operation using a logical file, the program must supply data that conforms with the format used by the logical file.

The following chart shows what types of data mapping are valid between physical and logical files.

Physical File Data Type	**Logical File Data Type**				
	Character or Hexadecimal	**Zoned**	**Packed**	**Binary**	**Floating Point**
Character or Hexadecimal	Valid	See Note 1	Not valid	Not valid	Not valid
Zoned	See Note 1	Valid	Valid	See Note 2	Valid
Packed	Not valid	Valid	Valid	See Note 2	Valid
Binary	Not valid	See Note 2	See Note 2	See Note 3	See Note 2
Floating Point	Not valid	Valid	Valid	See Note 2	Valid

Notes:

1. Valid only if the number of characters or bytes equals the number of digits.
2. Valid only if the binary field has zero decimal positions.
3. Valid only if both binary fields have the same number of decimal positions.

Note: For information about mapping DBCS fields, see Appendix B.

Describing Field Use for Logical Files

You can specify that fields in database files are to be input-only, both (input/output), or neither fields. Do this by specifying one of the following in position 38:

Entry	Meaning
Blank	For simple or multiple format logical files, defaults to B (both) For join logical files, defaults to I (input only)
B	Both input and output allowed; not valid for join logical files
I	Input only (read only)
N	Neither input nor output; valid only for join logical files

Note: The usage value (in position 38) is not used on a reference function. When another file refers to a field (using a REF or REFFLD keyword) in a logical file, the usage value is not copied into that file.

Both

A both field can be used for both input and output operations. Your program can read data from the field and write data to the field. Both fields are not valid for join logical files, because join logical files are read-only files.

Input Only

An input only field can be used for read operations only. Your program can read data from the field, but cannot update the field in the file. Typical cases of input-only fields are key fields (to reduce maintenance of access paths by preventing changes to key field values), sensitive fields that a user can see but not update (for example, salary), and fields for which either the translation table (TRNTBL) keyword or the substring (SST) keyword is specified.

If your program updates a record in which you have specified input-only fields, the input-only fields are not changed in the file. If your program adds a record that has input-only fields, the input-only fields take default values (DFT keyword).

Neither

A neither field is used neither for input nor for output. It is valid only for join logical files. A neither field can be used as a join field in a join logical file, but your program cannot read or update a neither field.

Use neither fields when the attributes of join fields in the physical files do not match. In this case, one or both join fields must be defined again. However, you cannot include these redefined fields in the record format (the application program does not see the redefined fields.) Therefore, redefined join fields can be coded N so that they do not appear in the record format.

A field with N in position 38 does not appear in the buffer used by your program. However, the field description is displayed with the Display File Field Description (DSPFFD) command.

Neither fields cannot be used as select/omit or key fields.

For an example of a neither field, see "Describing Fields That Never Appear in the Record Format (Example 5)" on page 6-38.

Deriving New Fields from Existing Fields

Fields in a logical file can be derived from fields in the physical file the logical file is based on or from fields in the same logical file. For example, you can concatenate, using the CONCAT keyword, two or more fields from a physical file to make them appear as one field in the logical file. Likewise, you can divide one field in the physical file to make it appear as multiple fields in the logical file with the SST keyword.

Concatenated Fields

Using the CONCAT keyword, you can combine two or more fields from a physical file record format to make one field in a logical file record format. For example, a physical file record format contains the fields *Month*, *Day*, and *Year*. For a logical file, you concatenate these fields into one field, *Date*.

The field length for the resulting concatenated field is the sum of the lengths of the included fields (unless the fields in the physical file are binary or packed decimal, in which case they are changed to zoned decimal). The field length of the resulting field is automatically calculated by the system. A concatenated field can have:

- Column headings
- Validity checking
- Text description
- Edit code or edit word (numeric concatenated fields only)

Note: This editing and validity checking information is not used by the database management system but is retrieved when field descriptions from the database file are referred to in a display or printer file.

When fields are concatenated, the data types can change (the resulting data type is automatically determined by the system). The following rules and restrictions apply:

- OS/400 assigns the data type based on the data types of the fields that are being concatenated.
- The maximum length of a concatenated field varies depending on the data type of the concatenated field and the length of the fields being concatenated. If the concatenated field is zoned decimal (S), its total length cannot exceed 31 bytes; if it is character (A), its total length cannot exceed 32 766 bytes.
- In join logical files, the fields to be concatenated must be from the same physical file. The first field specified on the CONCAT keyword identifies which physical file is to be used. The first field must, therefore, be unique among the physical files on which the logical file is based, or you must also specify the JREF keyword to specify which physical file to use.
- The use of concatenated field must be I (input only).
- REFSHIFT cannot be specified on a concatenated field that has been assigned a data type of O or J.

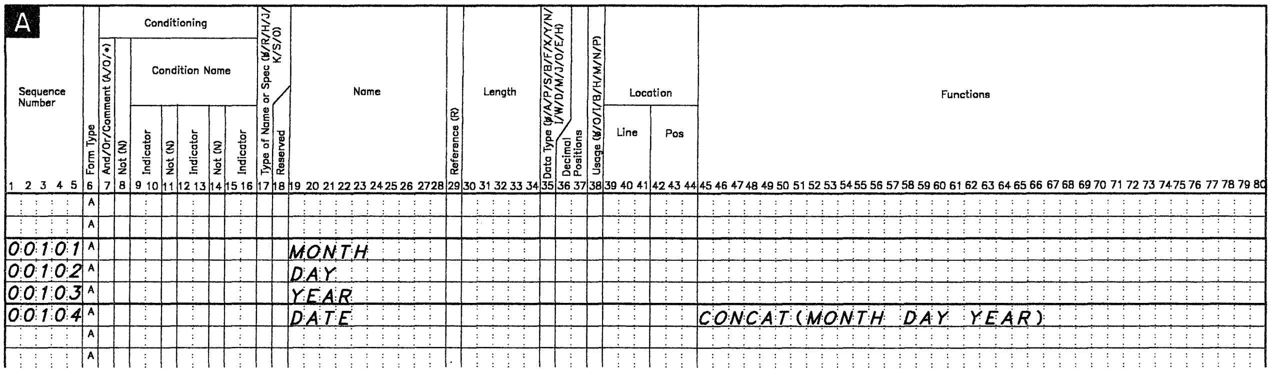
Note: For information about concatenating DBCS fields, see Appendix B.

When only numeric fields are concatenated, the sign of the last field in the group is used as the sign of the concatenated field.

Notes:

1. Numeric fields with decimal precision other than zero cannot be included in a concatenated field.
2. Floating-point fields cannot be included in a concatenated field.
3. For join logical files, all fields specified as parameter values on the CONCAT keyword must be from the same physical file. (The first field must identify the physical file it comes from by its uniqueness or from the JREF keyword.)

The following shows the field description in DDS for concatenation. (The CONCAT keyword is used to specify the fields to concatenate.)



```
00101A            MONTH
00102A            DAY
00103A            YEAR
00104A            DATE                      CONCAT(MONTH DAY YEAR)
```

RSLH247-1

In this example, the logical file record format includes the separate fields of *Month*, *Day*, and *Year*, as well as the concatenated *Date* field. Any of the following can be used:

- A format with the separate fields of *Month*, *Day*, and *Year*
- A format with only the concatenated *Date* field
- A format with the separate fields *Month*, *Day*, *Year* and the concatenated *Date* field

When both separate and concatenated fields exist in the format, any updates to the fields are processed in the sequence in which the DDS is specified. In the previous example, if the *Date* field contained 103188 and the *Month* field is changed to 12, when the record is updated, the month in the *Date* field would be used. The updated record would contain 103188. If the *Date* field were specified first, the updated record would contain 123188.

Concatenated fields can also be used as key fields and select/omit fields.

Substring Fields

You can use the SST keyword to specify which fields (character, hexadecimal, or zoned decimal) are in a substring. For example, assume you had defined the *Date* field in your physical file as 6 characters in length. You can describe the logical file with three fields, each 2 characters in length. You could use the SST keyword to define MM as 2 characters starting in position 1 of the *Date* field, DD as 2 characters starting in position 3 of the *Date* field, and YY as 2 characters starting in position 5 of the *Date* field.

Substring fields can also be used as key fields and select/omit fields.

Renamed Fields

You can name a field in a logical file differently than in a physical file using the RENAME keyword. You might want to rename a field in a logical file because the program was written using a different field name or because the original field name does not conform to the naming restrictions of the high-level language you are using.

Translated Fields

You can specify a translation table for a field using the TRNTBL keyword. When you read a logical file record and a translation table was specified for one or more fields in the logical file, the system translates the data from the field value in the physical file to the value determined by the translation table.

Describing Floating-Point Fields in Logical Files

You can use floating-point fields as mapped fields in logical files. A single- or double-precision floating-point field can be mapped to or from a zoned, packed, zero-precision binary, or another floating-point field. You cannot map between a floating-point field and a nonzero-precision binary field, a character field, a hexadecimal field, or a DBCS field.

Mapping between floating-point fields of different precision, single or double, or between floating-point fields and other numeric fields, can result in rounding or a loss of precision. Mapping a double-precision floating-point number to a single-precision floating-point number can result in rounding, depending on the particular number involved and its internal representation. Rounding is to the nearest (even) bit. The result always contains as much precision as possible. A loss of precision can also occur between two decimal numbers if the number of digits of precision is decreased.

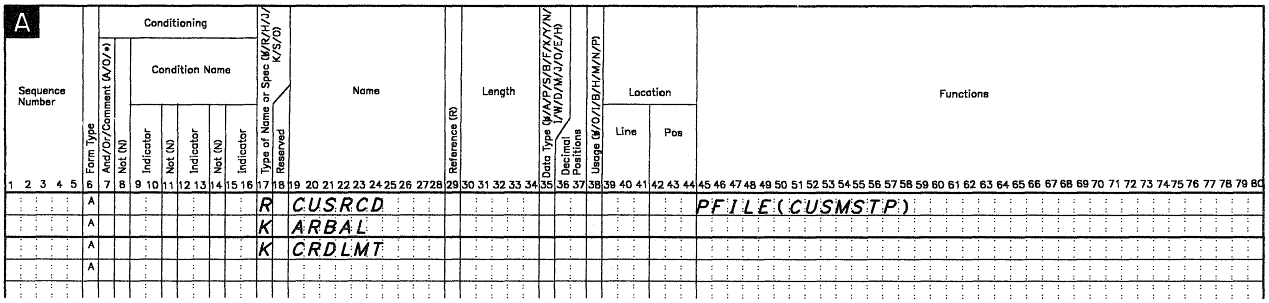
You can inadvertently change the value of a field which your program did not explicitly change. For floating-point fields, this can occur if a physical file has a double-precision field that is mapped to a single-precision field in a logical file, and you issue an update for the record through the logical file. If the internal representation of the floating-point number causes it to be rounded when it is mapped to the logical file, then the update of the logical record causes a permanent loss of precision in the physical file. If the rounded number is the key of the physical record, then the sequence of records in the physical file can also change.

A fixed-point numeric field can also be updated inadvertently if the precision is decreased in the logical file.

Describing Access Paths for Logical Files

The access path for a logical file record format can be specified in one of three ways:

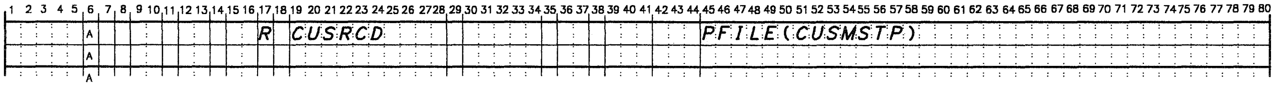
1. Keyed sequence access path specification. Specify key fields after the last record or field level specification. The key field names must be in the record format. For join logical files, the key fields must come from the first, or primary, physical file.


A
Sequence Number
Form Type
And/Or/Comment (A/O/*)
Conditioning
Condition Name
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

```
     A          R CUSRCD                    PFILE(CUSMSTP)
     A          K ARBAL
     A          K CRDLMT
     A
```

RSLH237-2

2. Arrival sequence access path specification. Specify no key fields. You can specify only one physical file on the PFILE keyword (and only one of the physical file's members when you add the logical file member).

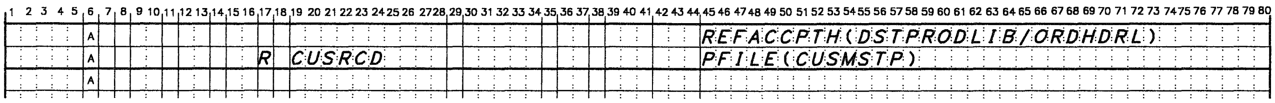


```
     A          R CUSRCD                    PFILE(CUSMSTP)
     A
     A
```

RSLH239-1

3. Previously defined keyed-sequence access path specification (for simple and multiple format logical files only). Specify the REFACCPTH keyword at the file level to identify a previously created database file whose access path and select/omit specifications are to be copied to this logical file. You cannot specify individual key or select/omit fields with the REFACCPTH keyword.

   Note: Even though the specified file's access path specifications are used, the system determines which file's access path, if any, will actually be shared. The system always tries to share access paths, regardless of whether the REFACCPTH keyword is used.



```
     A                                      REFACCPTH(DSTPRODLIB/ORDHDRL)
     A          R CUSRCD                    PFILE(CUSMSTP)
     A
```

RSLH238-2

When you define a record format for a logical file that shares key field specifications of another file's access path (using the DDS keyword, REFACCPTH), you can use any fields from the associated physical file record format. These fields do not have to be used in the file that describes the access path. However, all key and select/omit fields used in the file that describes the access path must be used in the new record format.

Selecting and Omitting Records Using Logical Files

The system can select and omit records when using a logical file. This can help you to exclude records in a file for processing convenience or for security.

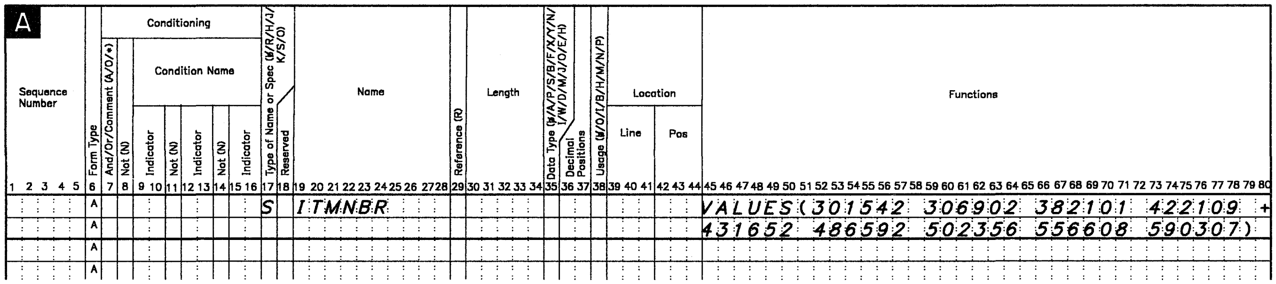
The process of selecting and omitting records is based on comparisons identified in position 17 of the *DDS Coding Form* for the logical file, and is similar to a series of comparisons coded in a high-level language program. For example, in a logical file that contains order detail records, you can specify that the only records you want to use are those in which the quantity ordered is greater than the quantity shipped. All other records are omitted from the access path. The omitted records remain in the physical file but are not retrieved for the logical file. If you are adding records to the physical file, all records are added but only the selected records are added to the access path.

In DDS, to specify select or omit, you specify an S (select) or O (omit) in position 17 of the *DDS Coding Form*. You then name the field (in positions 19 through 28) that will be used in the selection or omission process. In positions 45 through 80 you specify the comparison.

Note: Select/omit specifications appear after key specifications (if keys are specified).

Records can be selected and omitted by several types of comparisons:

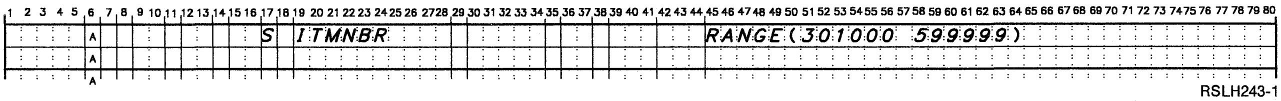
- VALUES. The contents of the field are compared to a list of not more than 100 values. If a match is found, the record is selected or omitted. In the following example, a record is selected if one of the values specified in the VALUES keyword is found in the *Itmnbr* field.


A
Sequence Number
Form Type
And/Or/Comment (A/O/*)
Conditioning
Condition Name
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

```
     A          S ITMNBR                VALUES(301542 306902 382101 422109 +
     A                                  431652 486592 502356 556608 590307)
     A
     A
```

RSLH242-2

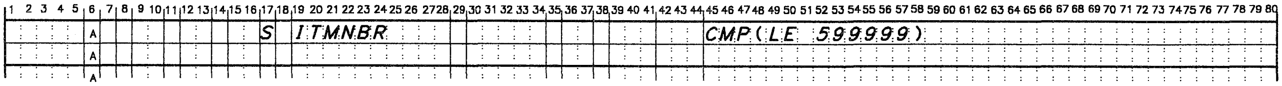
- RANGE. The contents of the field are compared to lower and upper limits. If the contents are greater than or equal to the lower limit and less than or equal to the upper limit, the record is selected or omitted. In the following example, all records with a range 301000 through 599999 in the *Itmnbr* field are selected.



```
     A          S ITMNBR                RANGE(301000 599999)
     A
     A
```

RSLH243-1

- CMP. The contents of a field are compared to a value or the contents of another field. Valid comparison codes are EQ, NE, LT, NL, GT, NG, LE, and GE. If the comparison is met, the record is selected or omitted. In the following example, a record is selected if its *Itmnbr* field is less than or equal to 599999:



```
     A          S ITMNBR                    CMP(LE 599999)
     A
     A
```

RSLH244-1

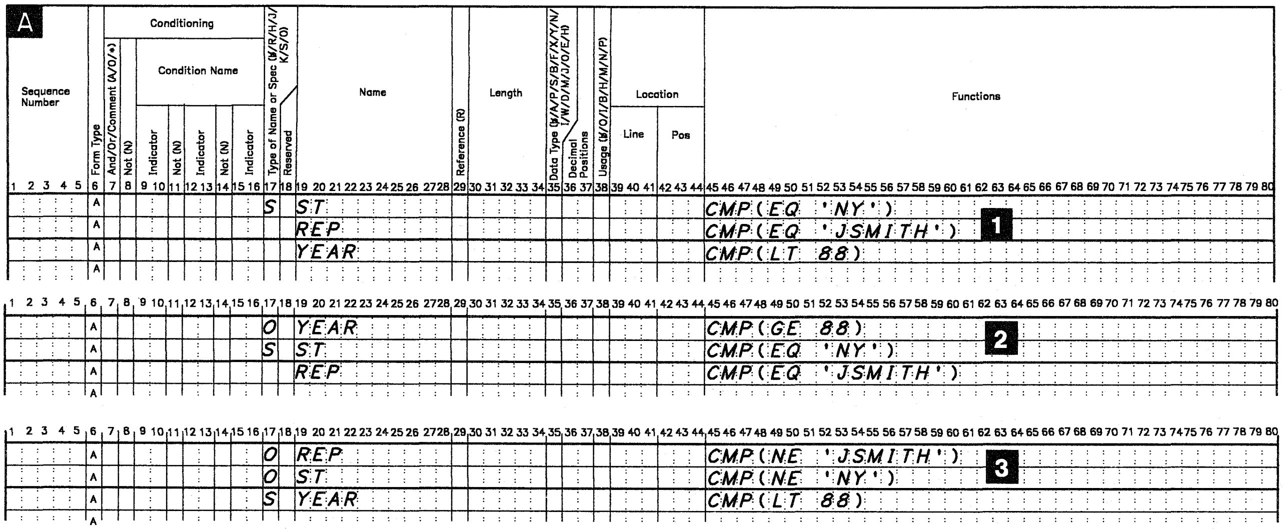
The value for a numeric field for which the CMP, VALUES, or RANGE keyword is specified is aligned based on the decimal positions specified for the field and filled with zeros where necessary. If decimal positions were not specified for the field, the decimal point is placed to the right of the farthest right digit in the value. For example, for a numeric field with length 5 and decimal position 2, the value 1.2 is interpreted as 001.20 and the value 100 is interpreted as 100.00.

The status of a record is determined by evaluating select/omit statements in the sequence you specify them. If a record qualifies for selection or omission, subsequent statements are ignored.

Normally the select and omit comparisons are treated independently from one another; the comparisons are ORed together. That is, if the select or omit comparison is met, the record is either selected or omitted. If the condition is not met, the system proceeds to the next comparison. To connect comparisons together, you simply leave a space in position 17 of the *DDS Coding Form*. Then, all the comparisons that were connected in this fashion must be met before the record is selected or omitted. That is, the comparisons are ANDed together.

The fewer comparisons, the more efficient the task is. So, when you have several select/omit comparisons, try to specify the one that selects or omits the most records first.

In the following examples, few records exist for which the *Rep* field is JSMITH. The examples show how to use DDS to select all the records before 1988 for a sales representative named JSMITH in the state of New York. All give the same results with different efficiency (in this example, **3** is the most efficient).


A
Sequence Number
Form Type
Conditioning
And/Or/Comment (A/O/*)
Condition Name
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions
1
2
3
RSLH241-2

```
     A          S ST                        CMP(EQ 'NY')
     A            REP                       CMP(EQ 'JSMITH')
     A            YEAR                      CMP(LT 88)
     A

     A          O YEAR                      CMP(GE 88)
     A          S ST                        CMP(EQ 'NY')
     A            REP                       CMP(EQ 'JSMITH')
     A

     A          O REP                       CMP(NE 'JSMITH')
     A          O ST                        CMP(NE 'NY')
     A          S YEAR                      CMP(LT 88)
     A
```

Figure 6-3. Three Ways to Code Select/Omit Function

1 All records must be compared with all of the select fields *St*, *Rep*, and *Year* before they can be selected or omitted.

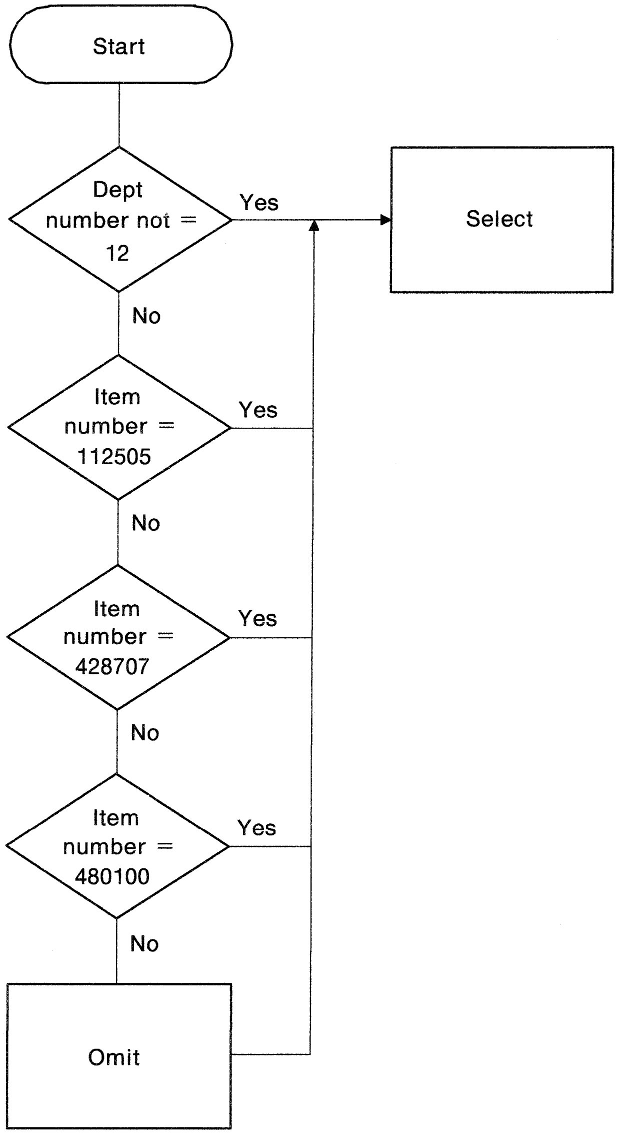
2 All records are compared with the *Year* field. Then, the records before 1988 have to be compared with the *St* and *Rep* fields.

3 All records are compared with the *Rep* field. Then, only the few for JSMITH are compared with the *St* field. Then, the few records that are left are compared to the *Year* field.

As another example, assume that you want to select the following:

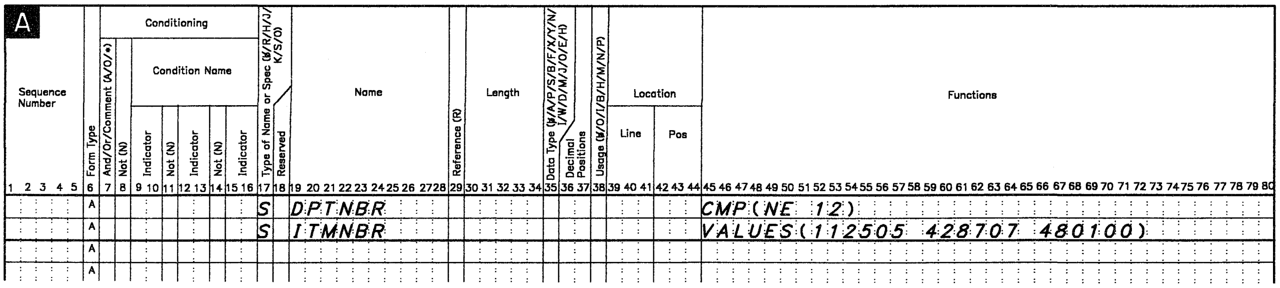
- All records for departments other than Department 12.
- Only those records for Department 12 that contain an item number 112505, 428707, or 480100. No other records for Department 12 are to be selected.

The following diagram shows the logic included in this example:



RSLH245-0

The following shows how to code this example using the DDS select and omit functions:



A		Conditioning																	
				Condition Name													Location		
Sequence Number	Form Type	And/Or/Comment (A/O/*)	Not (N)	Indicator	Not (N)	Indicator	Not (N)	Indicator	Type of Name or Spec (b/R/H/J/K/S/O)	Reserved	Name	Reference (R)	Length	Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)	Decimal Positions	Usage (b/O/I/B/H/M/N/P)	Line	Pos	Functions
1 2 3 4 5	6	7	8	9 10	11	12 13	14	15 16	17	18	19 20 21 22 23 24 25 26 27 28	29	30 31 32 33 34	35	36 37	38	39 40 41	42 43 44	45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
	A								S		DPTNBR								CMP(NE 12)
	A								S		ITMNBR								VALUES(112505 428707 480100)
	A																		
	A																		

RSLH246-2

It is possible to have an access path with select/omit values and process the file in arrival sequence. For example, a high-level language program can specify that the keyed access path is to be ignored. In this case, every record is read from the file in arrival sequence, but only those records meeting the select/omit values specified in the file are returned to the high-level language program.

A logical file with key fields and select/omit values specified can be processed in arrival sequence or using relative record numbers randomly. Records omitted by the select/omit values are not processed. That is, if an omitted record is requested by relative record number, the record is not returned to the high-level language program.

The system does not ensure that any additions or changes through a logical file will allow the record to be accessed again in the same logical file. For example, if the selection values of the logical file specifies only records with an A in *Fld1* and the program updates the record with a B in *Fld1*, the program cannot retrieve the record again using this logical file.

Note: You cannot select or omit based on the values of a floating-point field.

The two kinds of select/omit operations are: access path select/omit and dynamic select/omit. The default is access path select/omit. The select/omit specifications themselves are the same in each kind, but the system actually does the work of selecting and omitting records at different times.

Access Path Select/Omit

With access path select/omit, the access path only contains keys that meet the select/omit values specified for the logical file. When you specify key fields for a file, an access path is kept for the file and maintained by the system when you add or update records in the physical file(s) used by the logical file. The only index entries in the access path are those that meet the select/omit values.

Dynamic Select/Omit

With dynamic select/omit, when a program reads records from the file, the system only returns those records that meet the select/omit values. That is, the actual select/omit processing is done when records are read by a program, rather than when the records are added or changed. However, the keyed access path contains all the keys, not just keys from selected records. Access paths using dynamic select/omit allow more access path sharing, which can improve performance. For more information about access path sharing, see “Using Existing Access Paths” on page 6-14.

To specify dynamic select/omit, use the dynamic selection (DYNSLT) keyword. With dynamic select/omit, key fields are not required.

If you have a file that is updated frequently and read infrequently, you may not need to update the access path for select/omit purposes until your program reads the file. In this case, dynamic select/omit might be the correct choice. The following example helps describe this.

You use a code field (A=active, I=inactive), which is changed infrequently, to select/omit records. Your program processes the active records and the majority (over 80%) of the records are active. It can be more efficient to use DYNSLT to dynamically select records at processing time rather than perform access path maintenance when the code field is changed.

Using the Open Query File Command to Select/Omit Records

Another method of selecting records is using the QRYSLT parameter on the Open Query File (OPNQRYF) command. The open data path created by the OPNQRYF command is like a temporary logical file; that is, it is automatically deleted when it is closed. A logical file, on the other hand, remains in existence until you specifically delete it. For more details about the OPNQRYF command, see "Using the Open Query File (OPNQRYF) Command" on page 9-2.

Using Existing Access Paths

When two or more files are based on the same physical files and the same key fields in the same order, they automatically share the same keyed sequence access path. When access paths are shared, the amount of system activity required to maintain access paths and the amount of auxiliary storage used by the files is reduced.

When a logical file with a keyed sequence access path is created, the system always tries to share an existing access path. For access path sharing to occur, an access path must exist on the system that satisfies the following conditions:

- The logical file member to be added must be based on the same physical file members that the existing access path is based on.
- The length, data type, and number of decimal positions specified for each key field must be identical in both the new file and the existing file.
- If the FIFO, LIFO, or FCFO keyword is not specified, the new file can have fewer key fields than the existing access paths. That is, a new logical file can share an existing access path if the beginning part of the key is identical.
- The attributes of the access path (such as UNIQUE, LIFO, FIFO, or FCFO) and the attributes of the key fields (such as DESCEND, ABSVAL, UNSIGNED, and SIGNED) must be identical.

  Exceptions:

  1. A FIFO access path can share an access path in which the UNIQUE keyword is specified if all the other requirements for access path sharing are met.
  2. A UNIQUE access path can share a FIFO access path that needs to be rebuilt (for example, has *REBLD maintenance specified), if all the other requirements for access path sharing are met.
- If the new logical file has select/omit specifications, they must be identical to the select/omit specifications of the existing access path. However, if the new logical file specifies DYNSLT, it can share an existing access path if the existing access path has either:
  - The dynamic select (DYNSLT) keyword specified
  - No select/omit keywords specified
- The alternative collating sequence (ALTSEQ keyword) and the translation table (TRNTBL keyword) of the new logical file member, if any, must be identical to the alternative collating sequence and translation table of the existing access path.

Note: Logical files that contain concatenated or substring fields cannot share access paths with physical files.

The owner of any access path is the logical file member that originally created the access path. For a shared access path, if the logical member owning the access path is deleted, the first member to share the access path becomes the new owner. The FRCACCPTH, MAINT, RECOVER, and UNIT parameters on the Create Logical File (CRTLF) command need not match the same parameters on an existing access path for that access path to be shared. When an access path is shared by several logical file members, and the FRCACCPTH, MAINT, RECOVER, and UNIT parameters are not identical, the system maintains the access path by the most restrictive value for each of the parameters specified by the sharing members. The following illustrates how this occurs:

MBRA specifies the following:	MBRB specifies the following:	System does the following:
FRCACCPTH (*NO) MAINT (*IMMED) RECOVER (*AFTIPL)	FRCACCPTH (*YES) MAINT (*DLY) RECOVER (*NO)	FRCACCPTH (*YES) MAINT (*IMMED) RECOVER (*AFTIPL)

RSLH222-3

The UNIT parameter for the access path is always taken from the owning logical member.

Access path sharing does not depend on sharing between members; therefore, it does not restrict the order in which members can be deleted.

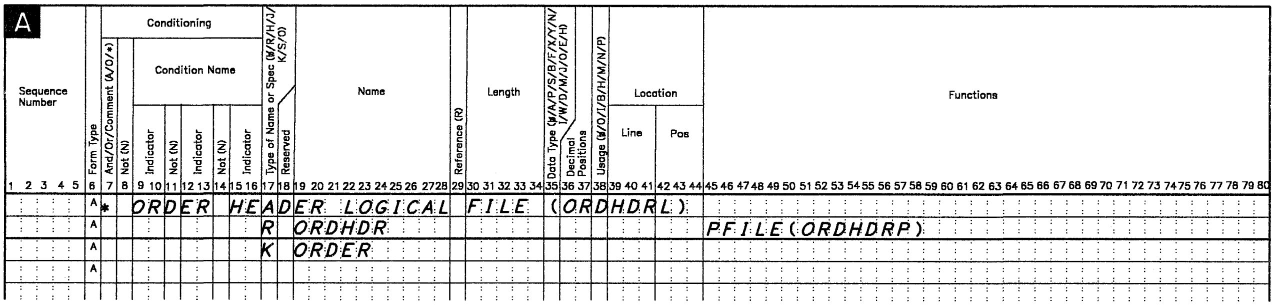
The Display File Description (DSPFD) and Display Database Relations (DSPDBR) commands show access path sharing relationships.

Creating a Logical File

Before creating a logical file, the physical file or files on which the logical file is based must already exist.

To create a logical file, take the following steps:

1. Type the DDS for the logical file into a source file. This can be done using SEU or another method. See “Working with Source Files” on page 17-4, for how source is placed in source files. The following shows the DDS for logical file ORDHDRL (an order header file):



```
     A* ORDER HEADER LOGICAL FILE (ORDHDRL)
     A          R ORDHDR                    PFILE(ORDHDRP)
     A          K ORDER
     A
```

RSLH248-1

   This file uses the key field *Order* (order number) to define the access path. The record format is the same as the associated physical file ORDHDRP. The record format name for the logical file must be the same as the record format name in the physical file because no field descriptions are given.

2. Create the logical file. You can use the Create Logical File (CRTLF) command.

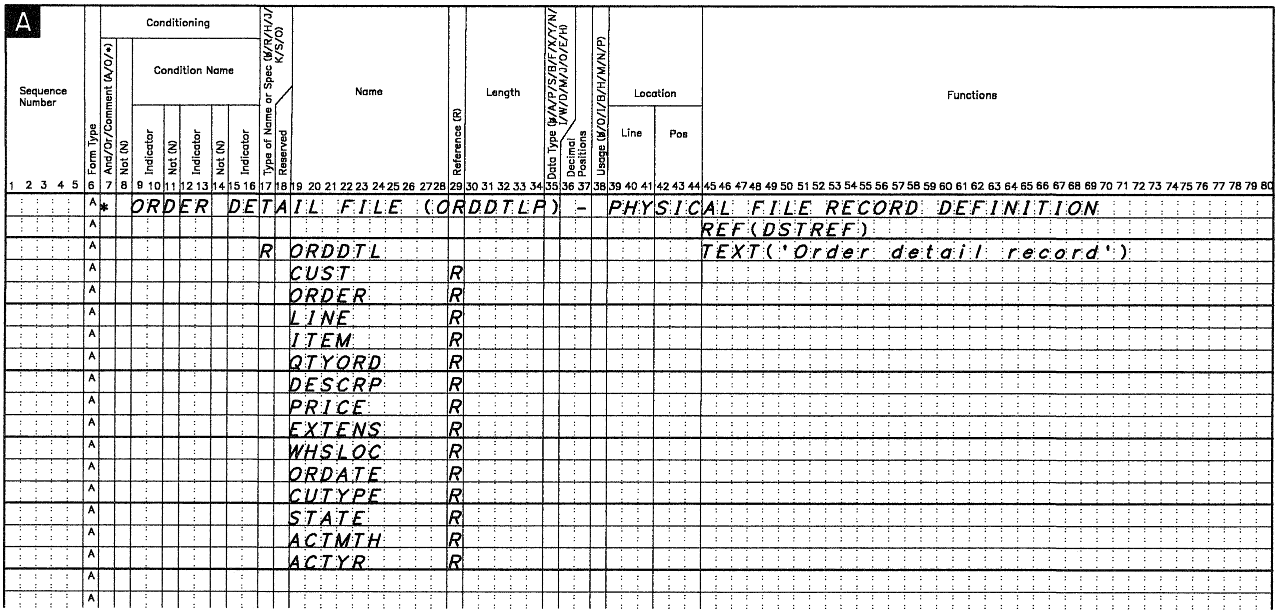
   The following shows how the CRTLF command could be typed:

```
CRTLF  FILE(DSTPRODLB/ORDHDRL)
       TEXT('Order header logical file')
```

As shown, this command uses some defaults. For example, because the SRCFILE and SRCMBR parameters are not specified, the system used DDS from the IBM-supplied source file QDDSSRC, and the source file member name is ORDHDRL (the same as the file name specified on the CRTLF command). The file ORDHDRL with one member of the same name is placed in the library DSTPRODLB.

Creating a Logical File with More Than One Record Format

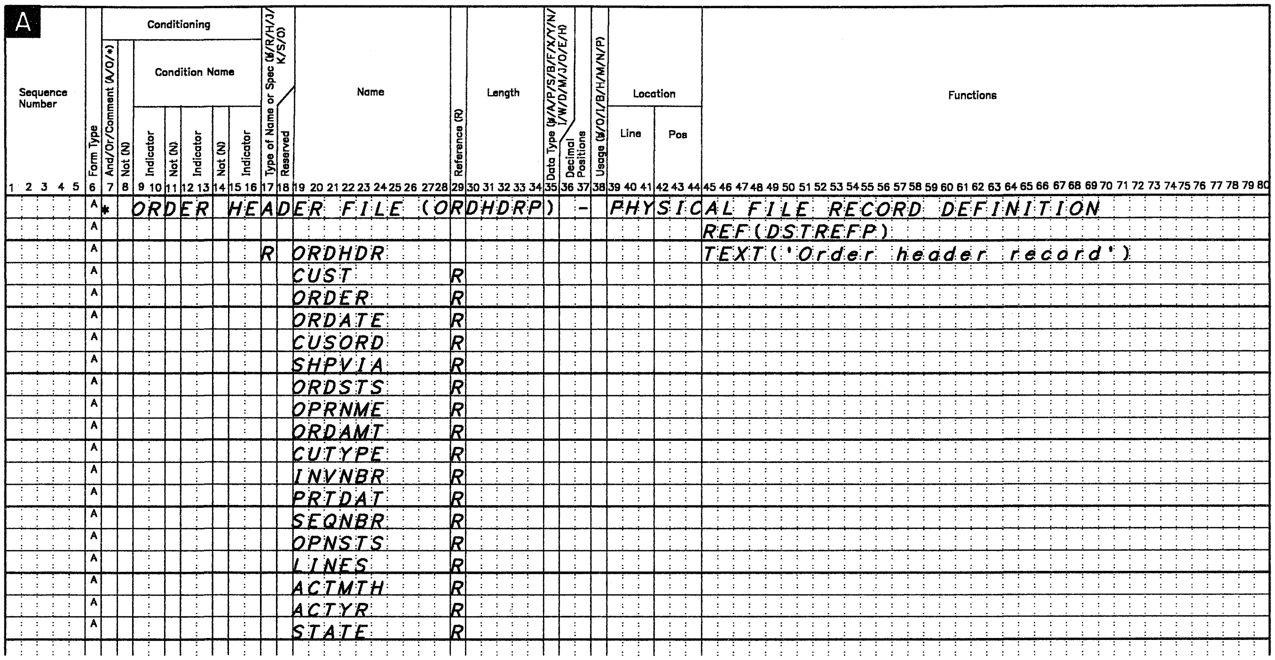
A multiple format logical file lets you use related records from two or more physical files by referring to only one logical file. Each record format is always associated with one or more physical files. You can use the same physical file in more than one record format.


A
Sequence Number
Form Type
And/Or/Comment (A/O/*)
Not (N)
Conditioning
Condition Name
Indicator
Not (N)
Indicator
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

```
     A* ORDER DETAIL FILE (ORDDTLP) - PHYSICAL FILE RECORD DEFINITION
     A                                      REF(DSTREF)
     A          R ORDDTL                    TEXT('Order detail record')
     A            CUST      R
     A            ORDER     R
     A            LINE      R
     A            ITEM      R
     A            QTYORD    R
     A            DESCRP    R
     A            PRICE     R
     A            EXTENS    R
     A            WHSLOC    R
     A            ORDATE    R
     A            CUTYPE    R
     A            STATE     R
     A            ACTMTH    R
     A            ACTYR     R
     A
     A
```

RSLH249-1

Figure 6-4. DDS for a Physical File (ORDDTLP) Built from a Field Reference File



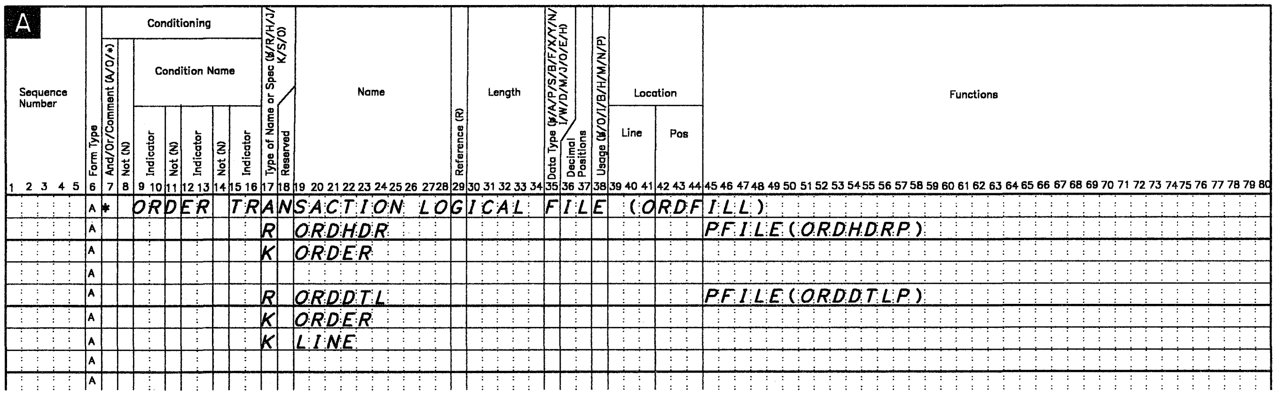
```
     A* ORDER HEADER FILE (ORDHDRP) - PHYSICAL FILE RECORD DEFINITION
     A                                      REF(DSTREFP)
     A          R ORDHDR                    TEXT('Order header record')
     A            CUST      R
     A            ORDER     R
     A            ORDATE    R
     A            CUSORD    R
     A            SHPVIA    R
     A            ORDSTS    R
     A            OPRNME    R
     A            ORDAMT    R
     A            CUTYPE    R
     A            INVNBR    R
     A            PRTDAT    R
     A            SEQNBR    R
     A            OPNSTS    R
     A            LINES     R
     A            ACTMTH    R
     A            ACTYR     R
     A            STATE     R
```

RSLH228-2

Figure 6-5. DDS for a Physical File (ORDHDRP) Built from a Field Reference File

The following example shows how to create a logical file ORDFILL with two record formats. One record format is defined for order header records from the physical file ORDHDRP; the other is defined for order detail records from the physical file ORDDTLP. (Figure 6-4 on page 6-16 shows the DDS for the physical file ORDDTLP, Figure 6-5 shows the DDS for the physical file ORDHDRP, and Figure 6-6 shows the DDS for the logical file ORDFILL.)

The logical file record format ORDHDR uses one key field, *Order*, for sequencing; the logical file record format ORDDTL uses two keys fields, *Order* and *Line*, for sequencing.



```
     A* ORDER TRANSACTION LOGICAL FILE (ORDFILL)
     A          R ORDHDR                    PFILE(ORDHDRP)
     A          K ORDER
     A
     A          R ORDDTL                    PFILE(ORDDTLP)
     A          K ORDER
     A          K LINE
     A
     A
```

RSLH250-1

Figure 6-6. DDS for the Logical File ORDFILL

To create the logical file ORDFILL with two associated physical files, use a Create Logical File (CRTLF) command like the following:

```
CRTLF  FILE(DSTPRODLB/ORDFILL)
       TEXT('Order transaction logical file')
```

The DDS source is in the member ORDFILL in the file QDDSSRC. The file ORDFILL with a member of the same name is placed in the DSTPRODLB library. The access path for the logical file member ORDFILL arranges records from both the ORDHDRP and ORDDTLP files. Record formats for both physical files are keyed on *Order* as the common field. Because of the order in which they were specified in the logical file description, they are merged in *Order* sequence with duplicates between files retrieved first from the header file ORDHDRP and second from the detail file ORDDTLP. Because FIFO, LIFO, or FCFO are not specified, the order of retrieval of duplicate keys in the same file is not guaranteed.

Controlling How Records Are Retrieved in a File with Multiple Formats

In a logical file with more than one record format, key field definitions are required. Each record format has its own key definition, and the record format key fields can be defined to merge the records of the different formats. Each record format does not have to contain every key field in the key. Consider the following records:

Header Record Format:

Record	Order	Cust	Ordate
1	41882	41394	050688
2	32133	28674	060288

Detail Record Format:

Record	Order	Line	Item	Qtyord	Extens
A	32133	01	46412	25	125000
B	32133	03	12481	4	001000
C	41882	02	46412	10	050000
D	32133	02	14201	110	454500
E	41882	01	08265	40	008000

In DDS, the header record format is defined before the detail record format. If the access path uses the *Order* field as the first key field for both record formats and the *Line* field as the second key field for only the second record format, both in ascending sequence, the order of the records in the access path is:

Record 2
Record A
Record D
Record B
Record 1
Record E
Record C

Note: Records with duplicate key values are arranged first in the sequence in which the physical files are specified. Then, if duplicates still exist within a record format, the duplicate records are arranged in the order specified by the FIFO, LIFO, or FCFO keyword. For example, if the logical file specified the DDS keyword FIFO, then duplicate records within the format would be presented in first-in-first-out sequence.

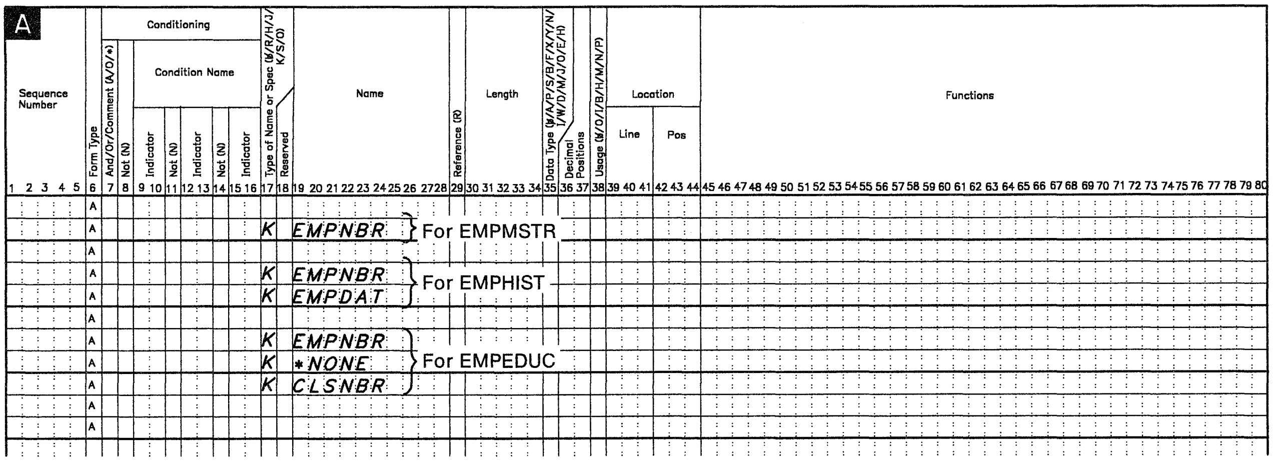
For logical files with more than one record format, you can use the *NONE DDS function for key fields to separate records of one record format from records of other record formats in the same access path. Generally, records from all record formats are merged based on key values. However, if *NONE is specified in DDS for a key field, only the records with key fields that appear in all record formats before the *NONE are merged.

The logical file in the following example contains three record formats, each associated with a different physical file:

Record Format	Physical File	Key Fields
EMPMSTR	EMPMSTR	Empnbr (employee number)
EMPHIST	EMPHIST	Empnbr, Empdat (employed date)
EMPEDUC	EMPEDUC	Empnbr, Clsnbr (class number)

Note: All record formats have one key field in common, the *Empnbr* field.

The DDS for this example is:


A
Sequence Number
Form Type
And/Or/Comment (A/O/*)
Conditioning
Condition Name
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

```
     A
     A          K EMPNBR   } For EMPMSTR
     A
     A          K EMPNBR   } For EMPHIST
     A          K EMPDAT   }
     A
     A          K EMPNBR   }
     A          K *NONE    } For EMPEDUC
     A          K CLSNBR   }
     A
     A
```

RSLH223-1

*NONE is assumed for the second and third key fields for EMPMSTR and the third key field for EMPHIST because no key fields follow these key field positions.

The following shows the arrangement of the records:

Empnbr	Empdat	Clsnbr	Record Format Name
426			EMPMSTR
426	6/15/74		EMPHIST
426		412	EMPEDUC
426		520	EMPEDUC
427			EMPMSTR
427	9/30/75		EMPHIST
427		412	EMPEDUC

*NONE serves as a separator for the record formats EMPHIST and EMPEDUC. All the records for EMPHIST with the same *Empnbr* field are grouped together and sorted by the *Empdat* field. All the records for EMPEDUC with the same *Empnbr* field are grouped together and sorted by the *Clsnbr* field.

Note: Because additional key field values are placed in the key sequence access path to guarantee the above sequencing, duplicate key values are not be predictable.

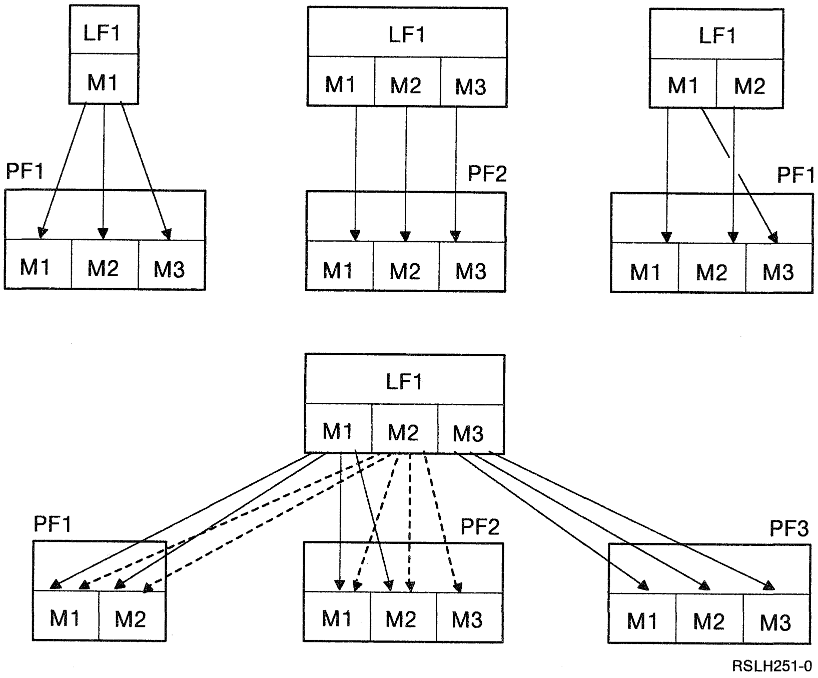
Controlling How Records Are Added to a File with Multiple Formats

When you add records to a multiple-format logical file and your application program uses a file name instead of a record format name, you need to write a format selector program. For more information about format selector programs, see "Identifying Which Record Format to Add in a File with Multiple Formats" on page 10-8.

Logical File Members

You can define members in logical files to separate the data into logical groups. The logical file member can be associated with one physical file member or with several physical file members.

The following illustrates this concept:



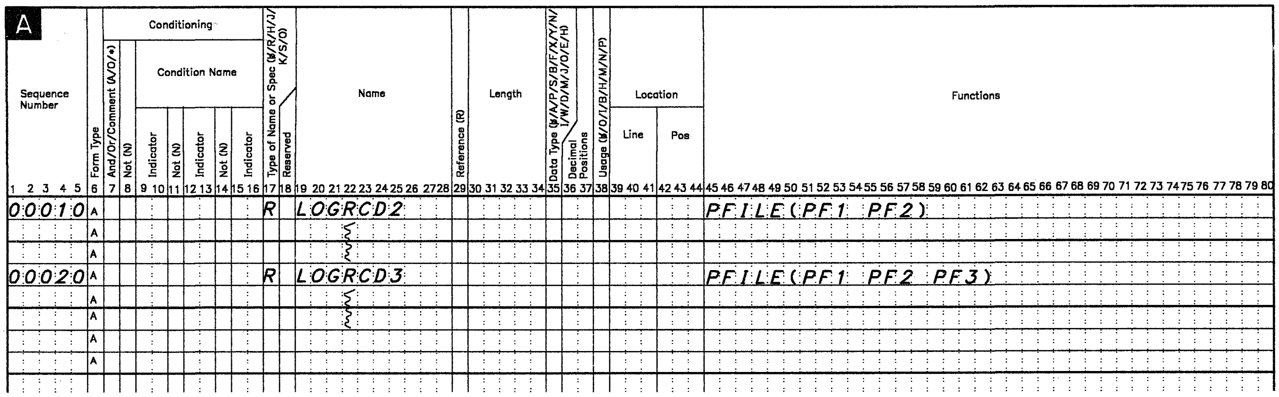
The record formats used with all logical members in a logical file must be defined in DDS when the file is created. If new record formats are needed, another logical file or record format must be created.

The attributes of an access path are determined by information specified in DDS and on commands when the logical file is created. The selection of data members is specified in the DTAMBRS parameter on the Create Logical File (CRTLF) and Add Logical File Member (ADDLFM) commands.

When a logical file is defined, the physical files used by the logical file are specified in DDS by the record level PFILE or JFILE keyword. If multiple record formats are defined in DDS, a PFILE keyword must be specified for each record format. You can specify one or more physical files for each PFILE keyword.

When a logical file is created or a member is added to the file, you can use the DTAMBRS parameter on the Create Logical File (CRTLF) or the Add Logical File Member (ADDLFM) command to specify which members of the physical files used by the logical file are to be used for data. *NONE can be specified as the physical file member name if no members from a physical file are to be used for data.

In the following example, the logical file has two record formats defined:



```
00010A          R LOGRCD2                   PFILE(PF1 PF2)
     A
     A
00020A          R LOGRCD3                   PFILE(PF1 PF2 PF3)
     A
     A
     A
     A
```

RSLH252-1

If the DTAMBRS parameter is specified on the CRTLF or ADDLFM command as in the following example:

```
DTAMBRS((PF1 M1) (PF2 (M1 M2)) (PF1 M1) (PF2 (*NONE)) (PF3 M3))
```

Record format LOGRCD2 is associated with physical file member M1 in PF1 and M1 and M2 in file PF2. Record format LOGRCD3 is associated with M1 in PF1 and M3 in PF3. No members in PF2 are associated with LOGRCD3. If the same physical file name is specified on more than one PFILE keyword, each occurrence of the physical file name is handled as a different physical file.

If a library name is not specified for the file on the PFILE keyword, the library list is used to find the physical file when the logical file is created. The physical file name and the library name then become part of the logical file description. The physical file names and the library names specified on the DTAMBRS parameter must be the same as those stored in the logical file description.

If a file name is not qualified by a library name on the DTAMBRS parameter, the library name defaults to *CURRENT, and the system uses the library name that is stored in the logical file description for the respective physical file name. This library name is either the library name that was specified for the file on the PFILE DDS keyword or the name of the library in which the file was found using the library list when the logical file was created.

When you add a member to a logical file, you can specify data members as follows:

- Specify no associated physical file members (DTAMBRS (*ALL) default). The logical file member is associated with all the physical file members of all physical files in all the PFILE keywords specified in the logical file DDS.
- Specify the associated physical file members (DTAMBRS parameter). If you do not specify library names, the logical file determines the libraries used. When more than one physical file member is specified for a physical file, the member names should be specified in the order in which records are to be retrieved when duplicate key values occur across those members. If you do not want to

include any members from a particular physical file, either do not specify the physical file name or specify the physical file name and *NONE for the member name. This method can be used to define a logical file member that contains a subset of the record formats defined for the logical file.

You can use the Create Logical File (CRTLF) command to create the first member when you create the logical file. Subsequent members must be added using the Add Logical File Member (ADDLFM) command. However, if you are going to add more members, you must specify more than 1 for the MAXMBRS parameter on the CRTLF command. The following example of adding a member to a logical file uses the CRTLF command used earlier in "Creating a Logical File" on page 6-15.

```
CRTLF   FILE(DSTPRODLB/ORDHDRL)
        MBR(*FILE) DTAMBRS(*ALL)
        TEXT('Order header logical file')
```

*FILE is the default for the MBR parameter and means the name of the member is the same as the name of the file. All the members of the associated physical file (ORDHDRP) are used in the logical file (ORDHDRL) in the logical file member. The text description is the text description of the member.

Join Logical File Considerations

This section covers the following topics:

- Basic concepts of joining two physical files (Example 1)
- Setting up a join logical file
- Using more than one field to join files (Example 2)
- Handling duplicate records in secondary files using the JDUPSEQ keyword (Example 3)
- Handling join fields whose attributes do not match (Example 4)
- Using fields that never appear in the record format to join files — neither fields (Example 5)
- Specifying key fields in join logical files (Example 6)
- Specifying select/omit statements in join logical files
- Joining three or more physical files (Example 7)
- Joining a physical file to itself (Example 8)
- Using default data for records missing from secondary files — the JDFTVAL keyword (Example 9)
- Describing a complex join logical file (Example 10)
- Performance considerations
- Data integrity considerations
- Summary of rules for join logical files

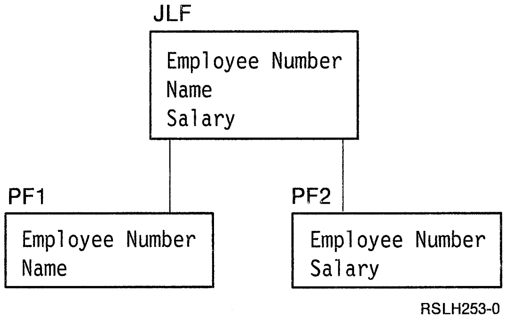
In general, the examples in this section include a picture of the files, DDS for the files, and sample data. For Example 1, several cases are given that show how to join files in different situations (when data in the physical files varies).

In the examples, for convenience and ease of recognition, join logical files are shown with the label JLF, and physical files are illustrated with the labels PF1, PF2, PF3, and so forth.

Basic Concepts of Joining Two Physical Files (Example 1)

A **join logical file** is a logical file that combines (in one record format) fields from two or more physical files. In the record format, not all the fields need to exist in all the physical files.

The following example illustrates a join logical file that joins two physical files. This example is used for the five cases discussed in Example 1.



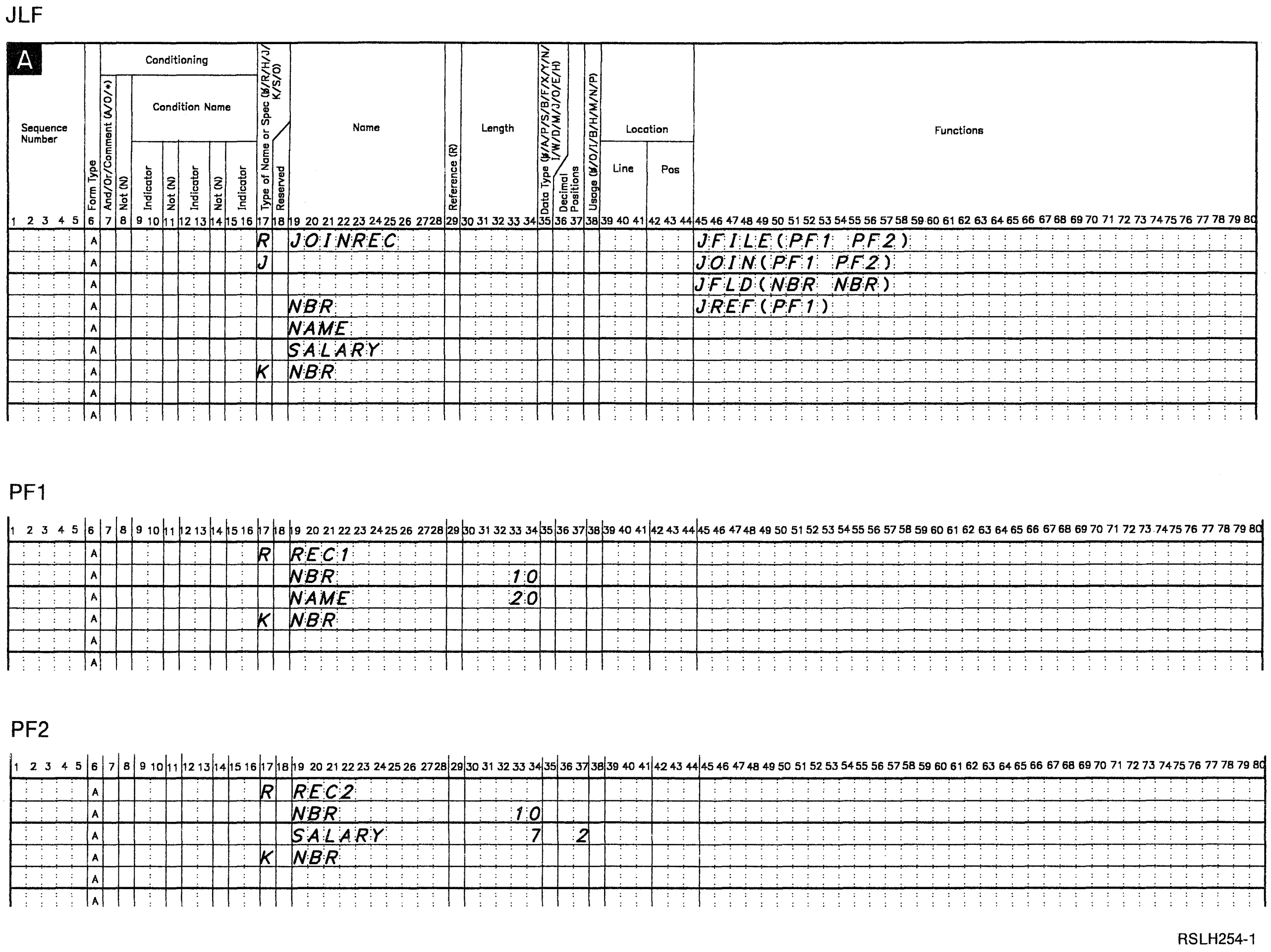
In this example, employee number is common to both physical files (PF1 and PF2), but name is found only in PF1, and salary is found only in PF2.

With a join logical file, the application program does one read operation (to the record format in the join logical file) and gets all the data needed from both physical files. Without the join specification, the logical file would contain two record formats, one based on PF1 and the other based on PF2, and the application program would have to do two read operations to get all the needed data from the two physical files. Thus, join provides more flexibility in designing your database.

However, a few restrictions are placed on join logical files:

- You cannot change a physical file through a join logical file. To do update, delete, or write (add) operations, you must create a second multiple format logical file and use it to change the physical files. You can also use the physical files, directly, to do the change operations.
- You cannot use DFU to display a join logical file.
- You can specify only one record format in a join logical file.
- The record format in a join logical file cannot be shared.
- A join logical file cannot share the record format of another file.
- Key fields must be fields defined in the join record format and must be fields from the first file specified on the JFILE keyword (which is called the primary file).
- Select/omit fields must be fields defined in the join record format, but can come from any of the physical files.
- Commitment control cannot be used with join logical files.

The following shows the DDS for Example 1:



JLF

```
     A          R JOINREC                   JFILE(PF1 PF2)
     A          J                           JOIN(PF1 PF2)
     A                                      JFLD(NBR NBR)
     A            NBR                       JREF(PF1)
     A            NAME
     A            SALARY
     A          K NBR
     A
     A
```

PF1

```
     A          R REC1
     A            NBR           10
     A            NAME          20
     A          K NBR
     A
     A
```

PF2

```
     A          R REC2
     A            NBR           10
     A            SALARY         7  2
     A          K NBR
     A
     A
```

RSLH254-1

Figure 6-7. DDS Example for Joining Two Physical Files

The following describes the DDS for the join logical file in Example 1 (see the *DDS Reference* for more information on the specific keywords):

The record level specification identifies the record format name used in the join logical file.

R Identifies the record format. Only one record format can be placed in a join logical file.

JFILE Replaces the PFILE keyword used in simple and multiple-format logical files. You must specify at least two physical files. The first file specified on the JFILE keyword is the **primary file**. The other files specified on the JFILE keyword are **secondary files**.

The join specification describes the way a pair of physical files is joined. The second file of the pair is always a secondary file, and there must be one join specification for each secondary file.

J Identifies the start of a join specification. You must specify at least one join specification in a join logical file. A join specification ends at the first field name specified in positions 19 through 28 or at the next J specified in position 17.

JOIN Identifies which two files are joined by the join specification. If only two physical files are joined by the join logical file, the JOIN keyword is optional. See “Joining Three or More Physical Files (Example 7)” on page 6-41 later in this section for an example of how to use this keyword.

JFLD Identifies the **join fields** that join records from the physical files specified on the JOIN JFLD must be specified at least once for each join specification. The join fields are fields common to the physical files. The first join field is a field from the first file specified on the JOIN keyword, and the second join field is a field from the second file specified on the JOIN keyword. The join fields must have the same attributes (data type, length, and decimal positions). If you are joining physical file fields that do not have the same attributes, you can redefine them for use in a join logical file. See “Using Join Fields Whose Attributes Are Different (Example 4)” on page 6-36 for a description and example.

The field level specification identifies the fields included in the join logical file.

Field names Specifies which fields (in this example, *Nbr*, *Name*, and *Salary*) are used by the application program. At least one field name is required. You can specify any field names from the physical files used by the logical file. You can also use keywords like RENAME, CONCAT, or SST as you would in simple and multiple format logical files.

JREF In the record format (which follows the join specification level and precedes the key field level, if any), the field names must uniquely identify which physical file the field comes from. In this example, the *Nbr* field occurs in both PF1 and PF2. Therefore, the JREF keyword is required to identify the file from which the *Nbr* field description will be used.

The key field level specification is optional, and includes the key field names for the join logical file.

K Identifies a key field specification. The K appears in position 17. Key field specifications are optional.

Key field names Key field names (in this example, *Nbr* is the only key field) are optional and make the join logical file an indexed (keyed sequence) file. Without key fields, the join logical file is an arrival sequence file. In join logical files, key fields must be fields from the primary file, and the key field name must be specified in positions 19 through 28 in the logical file record format.

The select/omit field level specification is optional, and includes select/omit field names for the join logical file.

S or O Identifies a select or omit specification. The S or O appears in position 17. Select/omit specifications are optional.

Select/omit field names Only those records meeting the select/omit values will be returned to the program using the logical file. Select/omit fields must be specified in positions 19 through 28 in the logical file record format.

Reading a Join Logical File

The following cases describe how the join logical file in Figure 6-7 on page 6-24 presents records to an application program.

The PF1 file is specified first on the JFILE keyword, and is therefore the primary file. When the application program requests a record, the system does the following:

1. Uses the value of the first join field in the primary file (the *Nbr* field in PF1).
2. Finds the first record in the secondary file with a matching join field (the *Nbr* field in PF2 matches the *Nbr* field in PF1).
3. For each match, joins the fields from the physical files into one record and provides this record to your program. Depending on how many records are in the physical files, one of the following conditions could occur:
   a. For all records in the primary file, only one matching record is found in the secondary file. The resulting join logical file contains a single record for each record in the primary file. See "Matching Records in Primary and Secondary Files (Case 1)" on page 6-27.
   b. For some records in the primary file, no matching record is found in the secondary file.

      If you specify the JDFTVAL keyword:

      - For those records in the primary file that have a matching record in the secondary file, the system joins to the secondary, or multiple secondaries. The result is one or more records for each record in the primary file.
      - For those records in the primary file that do not have a matching record in the secondary file, the system adds the default value fields for the secondary file and continues the join operation. You can use the DFT keyword in the physical file to define which defaults are used. See "Record Missing in Secondary File; JDFTVAL Keyword Not Specified (Case 2A)" on page 6-27 and "Record Missing in Secondary File; JDFTVAL Keyword Specified (Case 2B)" on page 6-28.

        Note: If the DFT keyword is specified in the secondary file, the value specified for the DFT keyword is used in the join. The result would be at least one join record for each primary record.

      - If a record exists in the secondary file, but the primary file has no matching value, no record is returned to your program. A second join logical file can be used that reverses the order of primary and secondary files to determine if secondary file records exist with no matching primary file records.

      If you do not specify the JDFTVAL keyword:

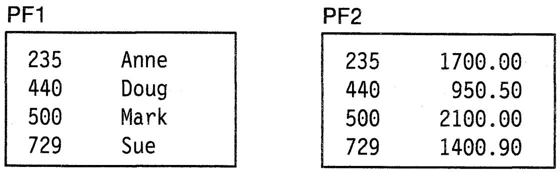
      - If a matching record in a secondary file exists, the system joins to the secondary, or multiple secondaries. The result is one or more records for each record in the primary file.
      - If a matching record in a secondary file does not exist, the system does not return a record.

      Note: When the JDFTVAL is not specified, the system returns a record only if a match is found in every secondary file for a record in the primary file.

      In the following examples, cases 1 through 4 describe sequential read operations, and case 5 describes reading by key.

Matching Records in Primary and Secondary Files (Case 1)

Assume that a join logical file is specified as in Figure 6-7 on page 6-24, and that four records are contained in both PF1 and PF2, as follows:



PF1

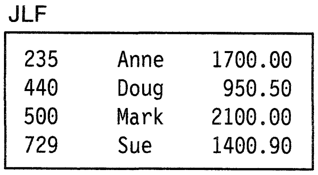
235	Anne
440	Doug
500	Mark
729	Sue

PF2

235	1700.00
440	950.50
500	2100.00
729	1400.90

RSLH255-0

The program does four read operations and gets the following records:



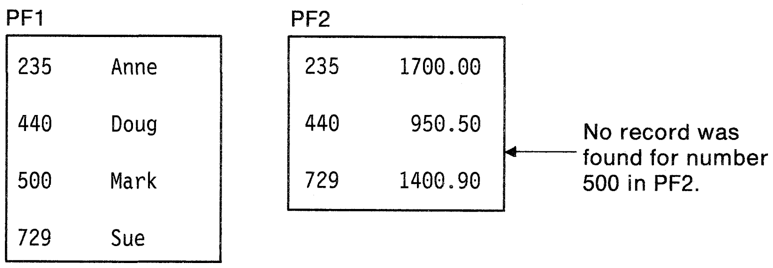
JLF

235	Anne	1700.00
440	Doug	950.50
500	Mark	2100.00
729	Sue	1400.90

RSLH256-0

Record Missing in Secondary File; JDFTVAL Keyword Not Specified (Case 2A)

Assume that a join logical file is specified as in Figure 6-7 on page 6-24, and that there are four records in PF1 and three records in PF2, as follows:



PF1

235	Anne
440	Doug
500	Mark
729	Sue

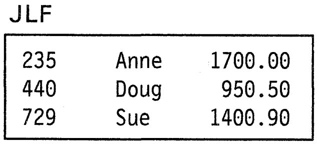
PF2

235	1700.00
440	950.50
729	1400.90

No record was found for number 500 in PF2.

RSLH257-1

With the join logical file shown in Example 1, the program reads the join logical file and gets the following records:



JLF

235	Anne	1700.00
440	Doug	950.50
729	Sue	1400.90

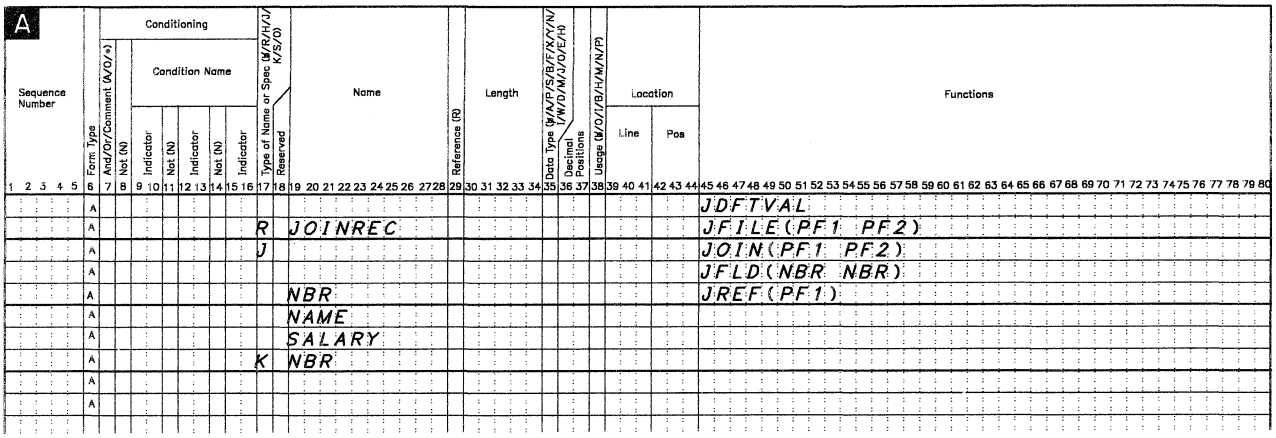
RSLH258-0

If you do not specify the JDFTVAL keyword and no match is found for the key field in the secondary file, the record is not included in the join logical file.

Record Missing in Secondary File; JDFTVAL Keyword Specified (Case 2B)

Assume that a join logical file is specified as in Figure 6-7 on page 6-24, except that the JDFTVAL keyword is specified, as shown in the following DDS:

JLF

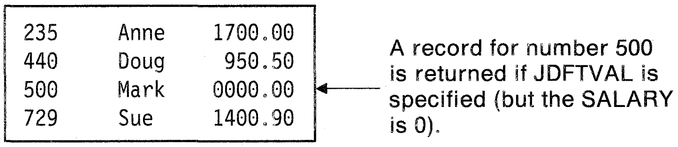


```
     A                                      JDFTVAL
     A          R JOINREC                   JFILE(PF1 PF2)
     A          J                           JOIN(PF1 PF2)
     A                                      JFLD(NBR NBR)
     A            NBR                       JREF(PF1)
     A            NAME
     A            SALARY
     A          K NBR
     A
     A
```

RSLH259-1

The program reads the join logical file and gets the following records:

JLF



```
235     Anne     1700.00
440     Doug      950.50
500     Mark     0000.00
729     Sue      1400.90
```

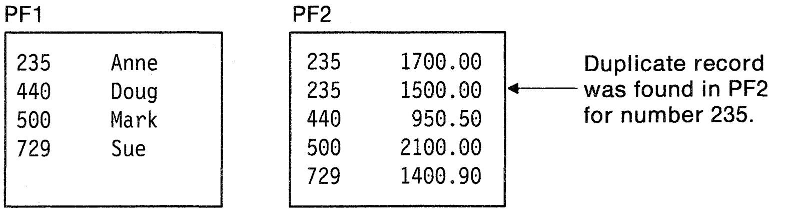
A record for number 500 is returned if JDFTVAL is specified (but the SALARY is 0).

RSLH260-0

With JDFTVAL specified, the system returns a record for 500, even though the record is missing in PF2. Without that record, some field values can be missing in the join record (in this case, the *Salary* field is missing). With JDFTVAL specified, missing character fields normally use blanks; missing numeric fields use zeroes. However, if the DFT keyword is specified for the field in the physical file, the default value specified on the DFT keyword is used.

Secondary File Has More Than One Match for a Record in the Primary File (Case 3)

Assume that a join logical file is specified as in Figure 6-7 on page 6-24, and that four records in PF1 and five records in PF2, as follows:



PF1

```
235     Anne
440     Doug
500     Mark
729     Sue
```

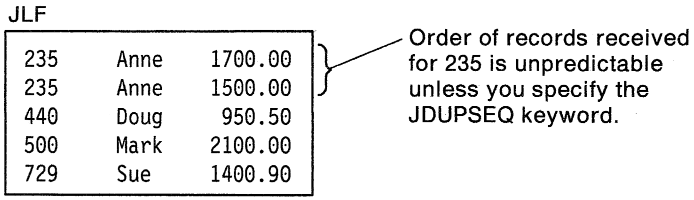
PF2

```
235     1700.00
235     1500.00
440      950.50
500     2100.00
729     1400.90
```

Duplicate record was found in PF2 for number 235.

RSLH261-1

The program gets five records:



JLF

235	Anne	1700.00
235	Anne	1500.00
440	Doug	950.50
500	Mark	2100.00
729	Sue	1400.90

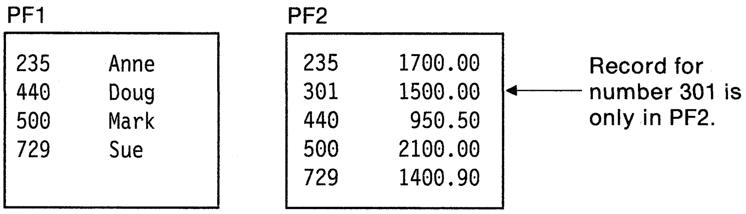
Order of records received for 235 is unpredictable unless you specify the JDUPSEQ keyword.

RSLH262-0

For more information, see "Reading Duplicate Records in Secondary Files (Example 3)" on page 6-34.

Extra Record in Secondary File (Case 4)

Assume that a join logical file is specified as in Figure 6-7 on page 6-24, and that four records are contained in PF1 and five records in PF2, as follows:



PF1

235	Anne
440	Doug
500	Mark
729	Sue

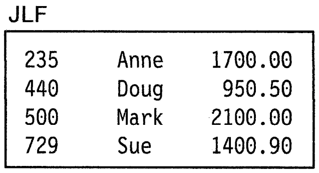
PF2

235	1700.00
301	1500.00
440	950.50
500	2100.00
729	1400.90

Record for number 301 is only in PF2.

RSLH263-1

The program reads the join logical file and gets only four records, which would be the same even if JDFTVAL was specified (because a record must always be contained in the primary file to get a join record):



JLF

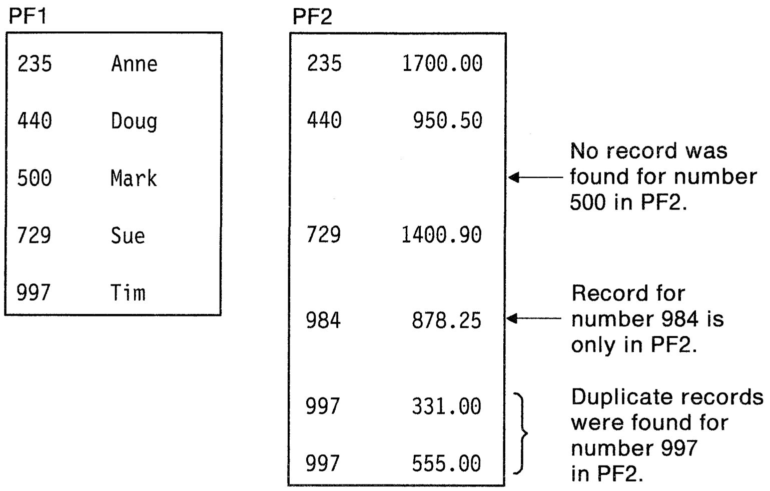
235	Anne	1700.00
440	Doug	950.50
500	Mark	2100.00
729	Sue	1400.90

RSLH264-0

Random Access (Case 5)

Assume that a join logical file is specified as in Figure 6-7 on page 6-24. Note that the join logical file has key fields defined. This case shows which records would be returned for a random access read operation using the join logical file.

Assume that PF1 and PF2 have the following records:



PF1

235	Anne
440	Doug
500	Mark
729	Sue
997	Tim

PF2

235	1700.00
440	950.50
729	1400.90
984	878.25
997	331.00
997	555.00

No record was found for number 500 in PF2.
Record for number 984 is only in PF2.
Duplicate records were found for number 997 in PF2.

RSLH265-1

The program can get the following records:

Given a value of 235 from the program for the *Nbr* field in the logical file, the system supplies the following record:

235	Anne	1700.00

RSLH266-0

Given a value of 500 from the program for the *Nbr* field in the logical file and with the JDFTVAL keyword specified, the system supplies the following record:

500	Mark	0.00

RSLH267-0

Note: If the JDFTVAL keyword was not specified in the join logical file, no record would be found for a value of 500 because no matching record is contained in the secondary file.

Given a value of 984 from the program for the *Nbr* field in the logical file, the system supplies no record and a `no record found` exception occurs because record 984 is not in the primary file.

Given a value of 997 from the program for the *Nbr* field in the logical file, the system returns one of the following records:

997	Tim	331.00

or

997	Tim	555.00

RSLH268-0

Which record is returned to the program cannot be predicted. To specify which record is returned, specify the JDUPSEQ keyword in the join logical file. See "Reading Duplicate Records in Secondary Files (Example 3)" on page 6-34.

Notes:

1. With random access, the application programmer must be aware that duplicate records could be contained in PF2, and ensure that the program does more than one read operation for records with duplicate keys. If the program were using sequential access, a second read operation would get the second record.
2. If you specify the JDUPSEQ keyword, the system can create a separate access path for the join logical file (because there is less of a chance the system will find an existing access path that it can share). If you omit the JDUPSEQ keyword, the system can share the access path of another file. (In this case, the system would share the access path of PF2.)

Setting Up a Join Logical File

To set up a join logical file, do the following:

1. Find the field names of all the physical file fields you want in the logical file record format. (You can display the fields contained in files using the Display File Field Description [DSPFFD] command.)
2. Describe the fields in the record format. Write the field names in a vertical list. This is the start of the record format for the join logical file.

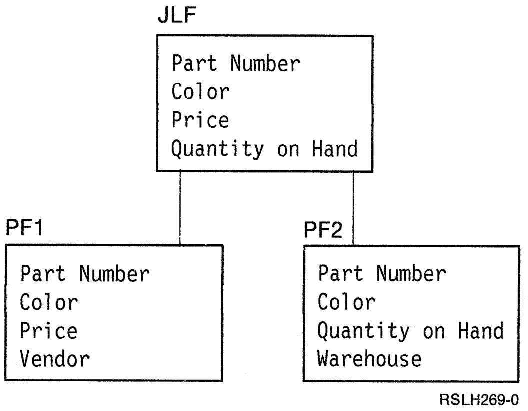
   Note: You can specify the field names in any order. If the same field names appear in different physical files, specify the name of the physical file on the JREF keyword for those fields. You can rename fields using the RENAME keyword, and concatenate fields from the same physical file using the CONCAT keyword. A subset of an existing character, hexadecimal, or zoned decimal field can be defined using the SST keyword. The substring of a character or zoned decimal field is a character field, and the substring of a hexadecimal field is also a hexadecimal field. You can redefine fields: changing their data type, length, or decimal positions.

3. Specify the names of the physical files as parameter values on the JFILE keyword. The first name you specify is the primary file. The others are all secondary files. For best performance, specify the secondary files with the least records first after the primary file.

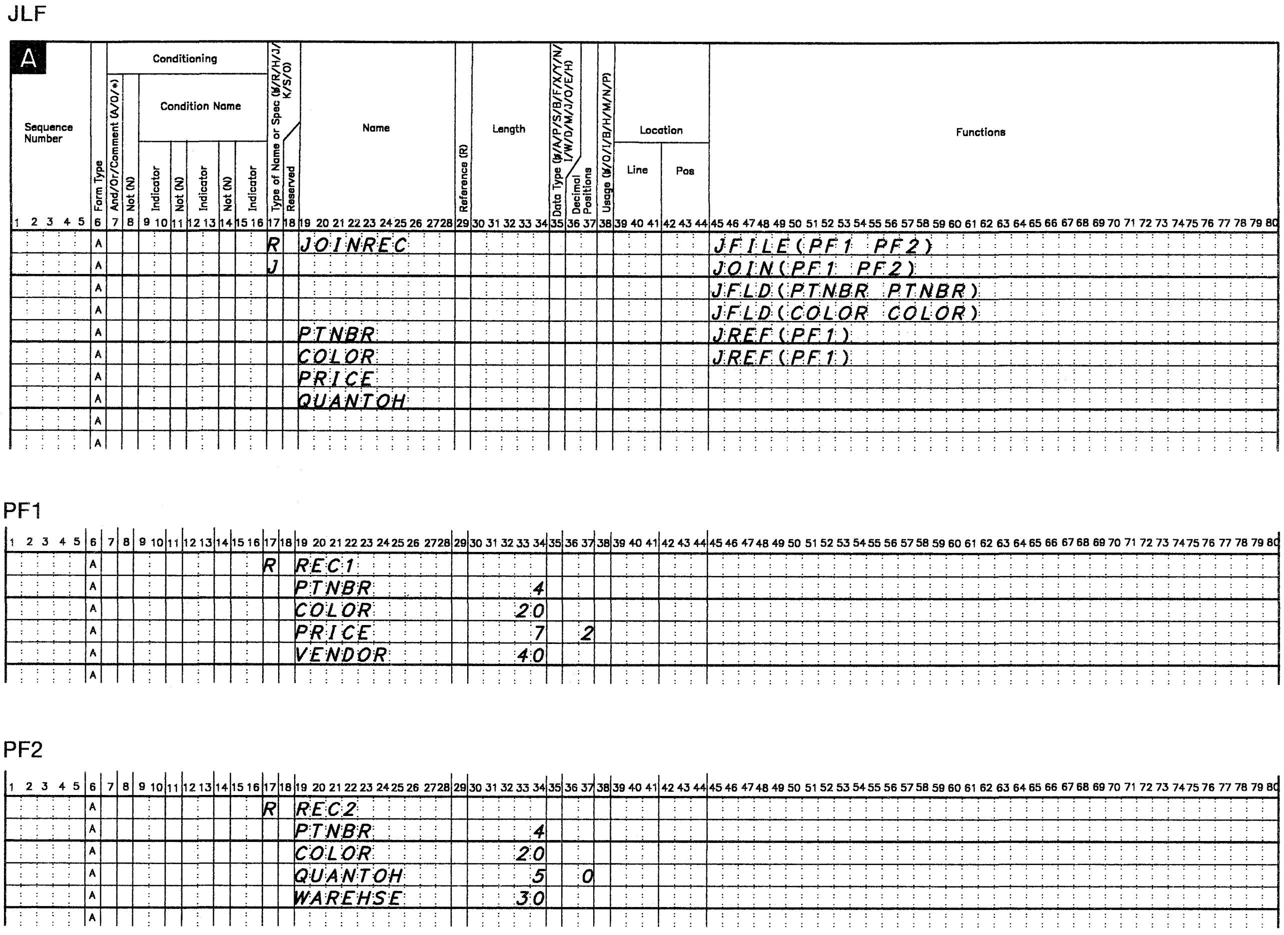
4. For each secondary file, code a join specification. On each join specification, identify which pair of files are joined (using the JOIN keyword; optional if only one secondary file), and identify which fields are used to join the pair (using the JFLD keyword; at least one required in each join specification).
5. Optionally, specify the following:
   a. The JDFTVAL keyword. Do this if you want to return a record for each record in the primary file even if no matching record exists in a secondary file.
   b. The JDUPSEQ keyword. Do this for fields that might have duplicate values in the secondary files. JDUPSEQ specifies on which field (other than one of the join fields) to sort these duplicates, and the sequence that should be used.
   c. Key fields. Key fields cannot come from a secondary file. If you omit key fields, records are returned in arrival sequence as they appear in the primary file.
   d. Select/omit fields. In some situations, you must also specify the dynamic selection (DYNSLT) keyword at the file level.
   e. Neither fields. For a description, see "Describing Fields That Never Appear in the Record Format (Example 5)" on page 6-38.

Using More Than One Field to Join Files (Example 2)

You can specify more than one join field to join a pair of files. The following shows the fields in the logical file and the two physical files.



The DDS for these files is as follows:



JLF

```
     A          R JOINREC                   JFILE(PF1 PF2)
     A          J                           JOIN(PF1 PF2)
     A                                      JFLD(PTNBR PTNBR)
     A                                      JFLD(COLOR COLOR)
     A            PTNBR                     JREF(PF1)
     A            COLOR                     JREF(PF1)
     A            PRICE
     A            QUANTOH
     A
     A
```

PF1

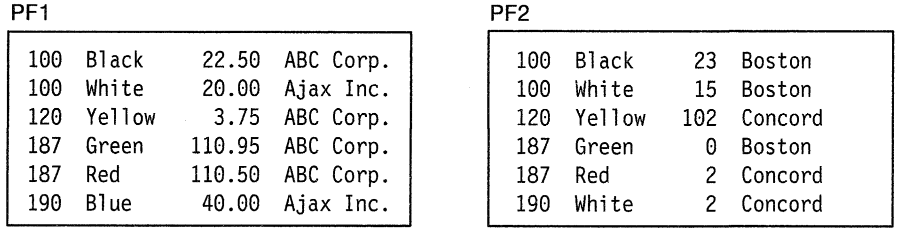
```
     A          R REC1
     A            PTNBR          4
     A            COLOR         20
     A            PRICE          7  2
     A            VENDOR        40
     A
```

PF2

```
     A          R REC2
     A            PTNBR          4
     A            COLOR         20
     A            QUANTOH        5  0
     A            WAREHSE       30
     A
```

RSLH270-2

Assume that the physical files have the following records:



PF1

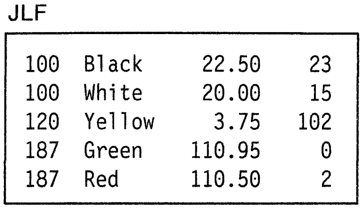
100	Black	22.50	ABC Corp.
100	White	20.00	Ajax Inc.
120	Yellow	3.75	ABC Corp.
187	Green	110.95	ABC Corp.
187	Red	110.50	ABC Corp.
190	Blue	40.00	Ajax Inc.

PF2

100	Black	23	Boston
100	White	15	Boston
120	Yellow	102	Concord
187	Green	0	Boston
187	Red	2	Concord
190	White	2	Concord

RSLH271-0

If the file is processed sequentially, the program receives the following records:



JLF

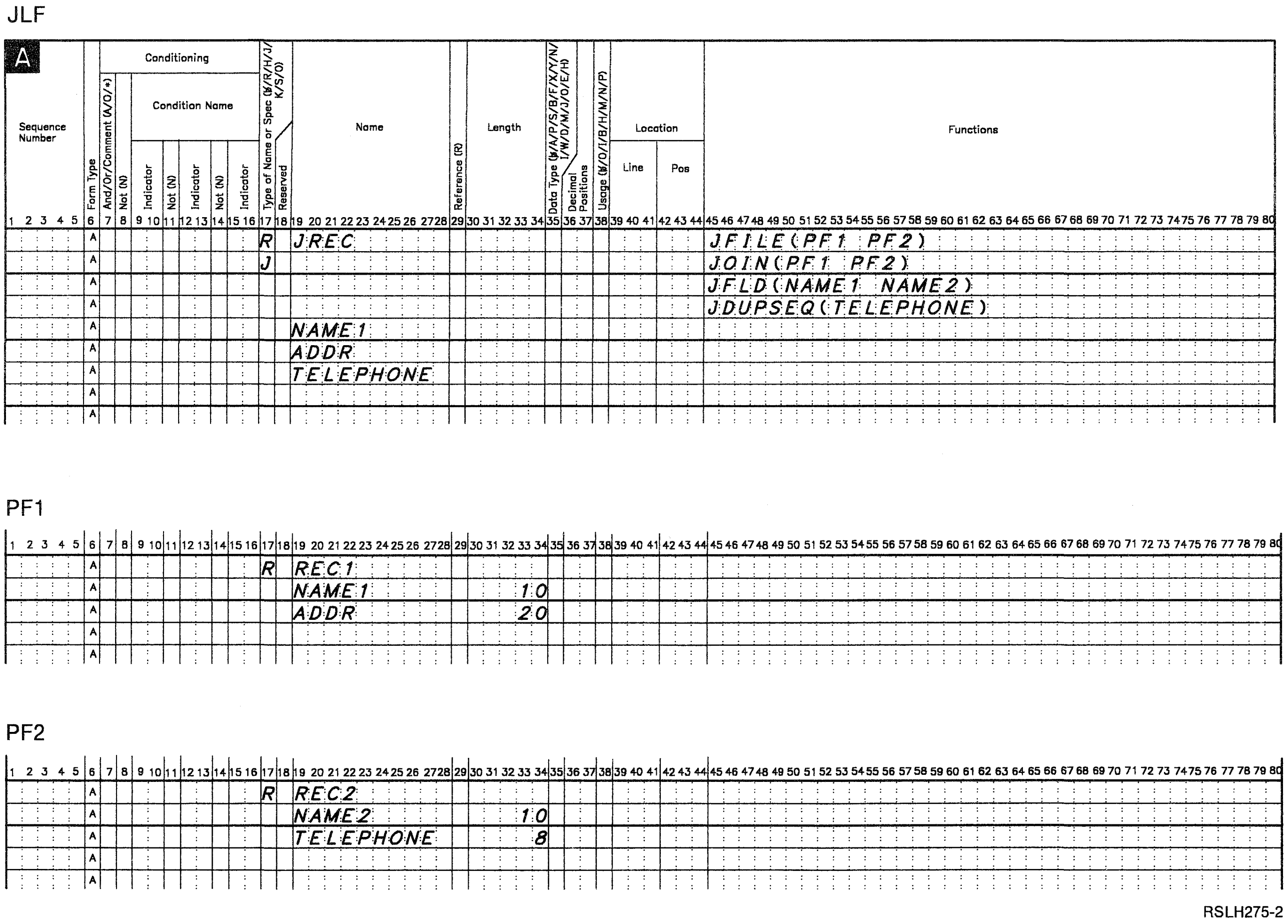
```
100  Black      22.50    23
100  White      20.00    15
120  Yellow      3.75   102
187  Green     110.95     0
187  Red       110.50     2
```

RSLH272-0

Note that no record for part number 190, color blue, is available to the program, because there a match was not found on both fields in the secondary file. Because JDFTVAL was not specified, no record is returned.

Reading Duplicate Records in Secondary Files (Example 3)

Sometimes a join to a secondary file produces more than one record from the secondary file. When this occurs, specifying the JDUPSEQ keyword in the join specification for that secondary file tells the system to base the order of the duplicate records on the specified field in the secondary file. The DDS for the physical files and for the join logical file are as follows:



JLF

```
     A          R JREC                      JFILE(PF1 PF2)
     A          J                           JOIN(PF1 PF2)
     A                                      JFLD(NAME1 NAME2)
     A                                      JDUPSEQ(TELEPHONE)
     A            NAME1
     A            ADDR
     A            TELEPHONE
```

PF1

```
     A          R REC1
     A            NAME1         10
     A            ADDR          20
```

PF2

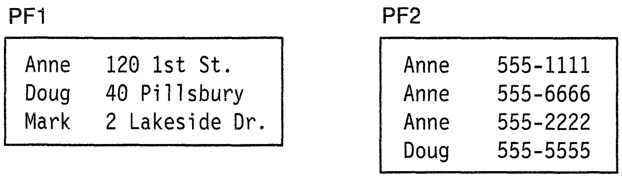
```
     A          R REC2
     A            NAME2         10
     A            TELEPHONE      8
```

A
Sequence Number
Form Type
And/Or/Comment (A/O/*)
Conditioning
Condition Name
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

RSLH275-2

Figure 6-8. DDS Example Using the JDUPSEQ Keyword

The physical files have the following records:



PF1

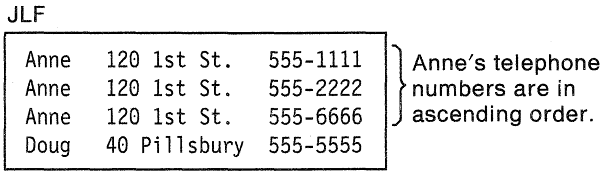
```
Anne   120 1st St.
Doug   40 Pillsbury
Mark   2 Lakeside Dr.
```

PF2

```
Anne     555-1111
Anne     555-6666
Anne     555-2222
Doug     555-5555
```

RSLH273-0

The join logical file returns the following records:



JLF

```
Anne   120 1st St.    555-1111
Anne   120 1st St.    555-2222
Anne   120 1st St.    555-6666
Doug   40 Pillsbury   555-5555
```

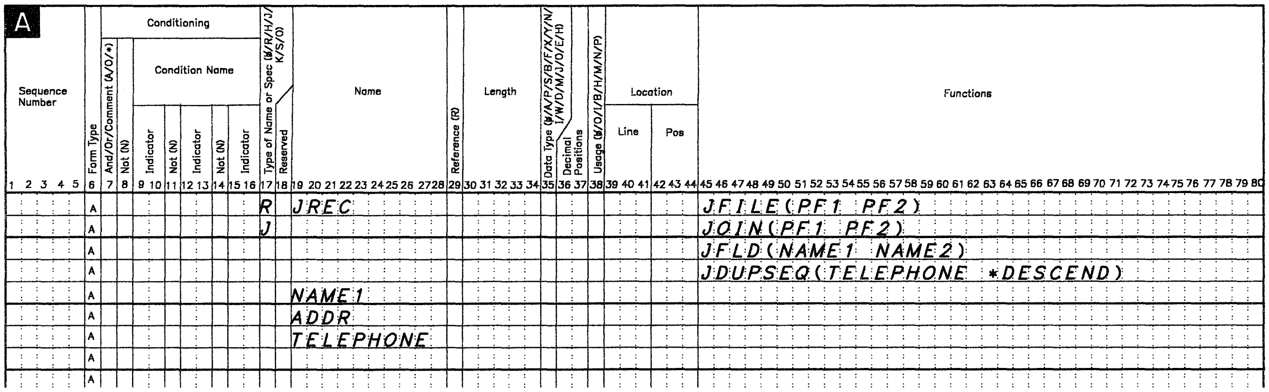
Anne's telephone numbers are in ascending order.

RSLH274-1

The program reads all the records available for Anne, then Doug, then Mark. Anne has one address, but three telephone numbers. Therefore, there are three records returned for Anne.

The records for Anne sort in ascending sequence by telephone number because the JDUPSEQ keyword sorts in ascending sequence unless you specify *DESCEND as the keyword parameter. The following example shows the use of *DESCEND in DDS:

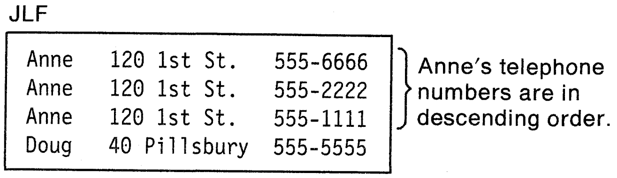
JLF



```
     A          R JREC                      JFILE(PF1 PF2)
     A          J                           JOIN(PF1 PF2)
     A                                      JFLD(NAME1 NAME2)
     A                                      JDUPSEQ(TELEPHONE *DESCEND)
     A            NAME1
     A            ADDR
     A            TELEPHONE
     A
     A
```

RSLH276-2

When you specify JDUPSEQ with *DESCEND, the records are returned as follows:



JLF

Anne	120 1st St.	555-6666
Anne	120 1st St.	555-2222
Anne	120 1st St.	555-1111
Doug	40 Pillsbury	555-5555

Anne's telephone numbers are in descending order.

RSLH277-1

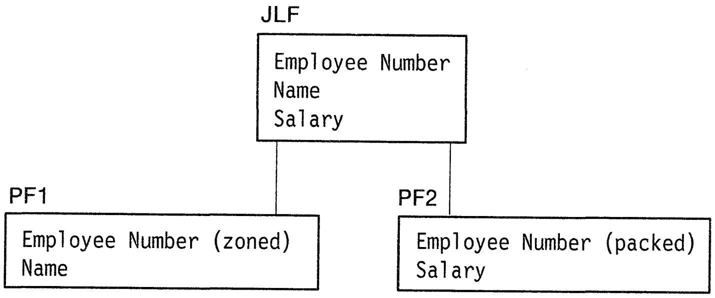
Note: The JDUPSEQ keyword applies only to the join specification in which it is specified. For an example showing the JDUPSEQ keyword in a join logical file with more than one join specification, see "A Complex Join Logical File (Example 10)" on page 6-47.

Using Join Fields Whose Attributes Are Different (Example 4)

Fields from physical files that you are using as join fields generally have the same attributes (length, data type, and decimal positions). For example, as in Figure 6-8 on page 6-34, the *Name1* field is a character field 10 characters long in physical file PF1, and can be joined to the *Name2* field, a character field 10 characters long in physical file PF2. The *Name1* and *Name2* fields have the same characteristics and, therefore, can easily be used as join fields.

The following is an example in which the join fields do not have the same attributes. The *Nbr* field in physical file PF1 and the *Nbr* field in physical file PF2 both have a length of 3 specified in position 34, but in the PF1 file the field is zoned (S in position 35), and in the PF2 file the field is packed (P in position 35). To join the two files using these fields as join fields, you must redefine one or both fields to have the same attributes.

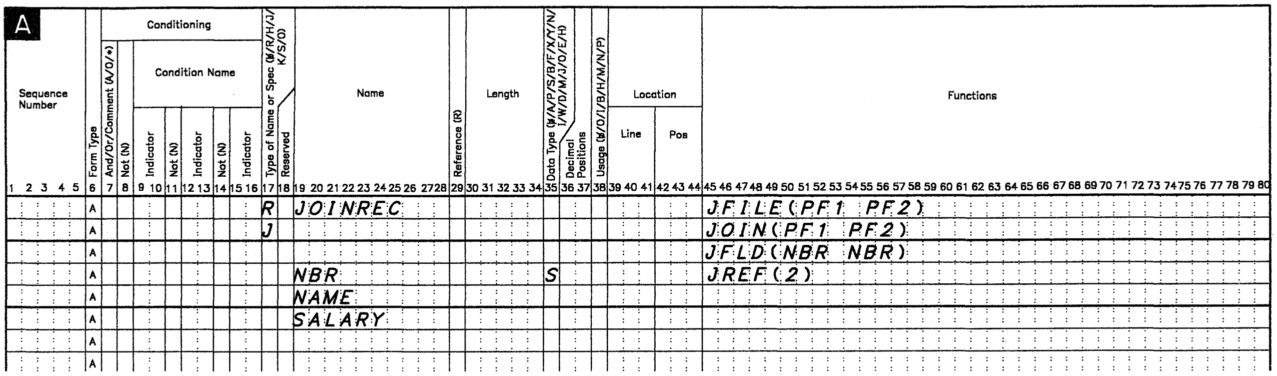
The following illustrates the fields in the logical and physical files:



RSLH278-0

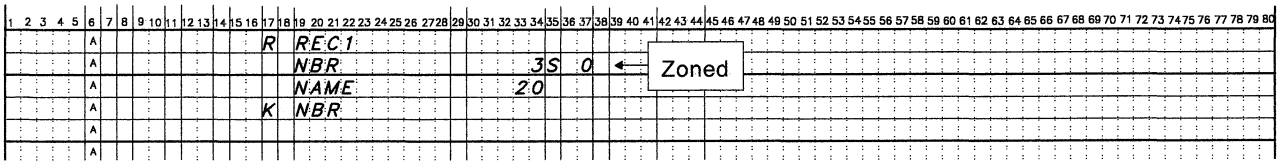
The DDS for these files is as follows:

JLF


A
Sequence Number
Form Type
And/Or/Comment (A/O/*)
Conditioning
Condition Name
Not (N)
Indicator
Not (N)
Indicator
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

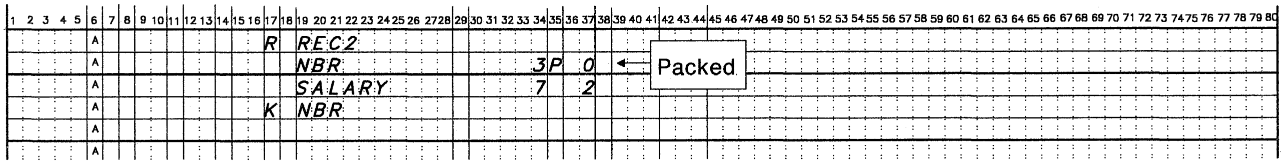
```
     A          R JOINREC                   JFILE(PF1 PF2)
     A          J                           JOIN(PF1 PF2)
     A                                      JFLD(NBR NBR)
     A            NBR                S      JREF(2)
     A            NAME
     A            SALARY
     A
     A
```

PF1



```
     A          R REC1
     A            NBR            3S 0
     A            NAME          20
     A          K NBR
     A
     A
```

PF2



```
     A          R REC2
     A            NBR            3P 0
     A            SALARY         7  2
     A          K NBR
     A
     A
```

RSLH279-1

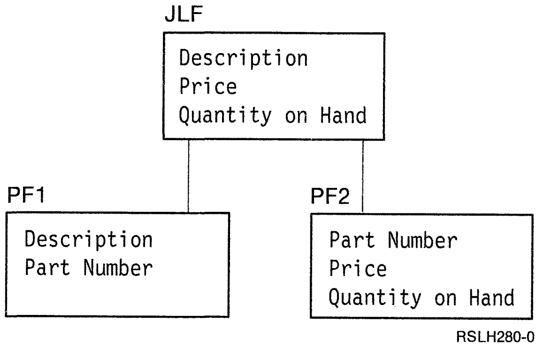
Note: In this example, the *Nbr* field in the logical file comes from PF2, because JREF(2) is specified. Instead of specifying the physical file name, you can specify a relative file number on the JREF keyword; in this example, the 2 indicates PF2.

Because the *Nbr* fields in the PF1 and PF2 files are used as the join fields, they must have the same attributes. In this example, they do not. Therefore, you must redefine one or both of them to have the same attributes. In this example, to resolve the difference in the attributes of the two employee number fields, the *Nbr* field in JLF (which is coming from the PF2 file) is redefined as zoned (S in position 35 of JLF).

Describing Fields That Never Appear in the Record Format (Example 5)

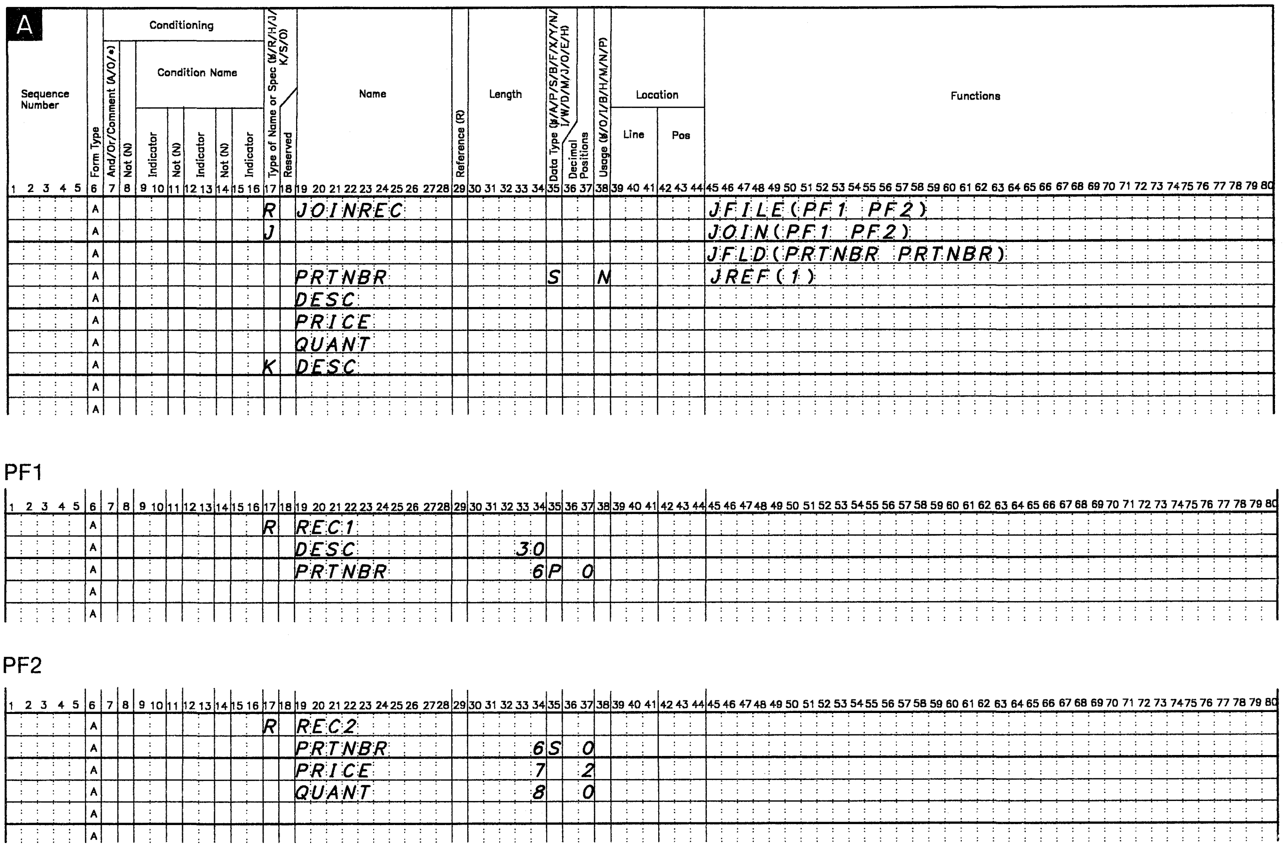
A neither field (N specified in position 38) can be used in join logical files for neither input nor output. Programs using the join logical file cannot see or read neither fields. Neither fields are not included in the record format. Neither fields cannot be key fields or used in select/omit statements in the joined file. You can use a neither field for a join field (specified at the join specification level on the JFLD keyword) that is redefined at the record level only to allow the join, but is not needed or wanted in the program.

In the following example, the program reads the descriptions, prices, and quantity on hand of parts in stock. The part numbers themselves are not wanted except to bring together the records of the parts. However, because the part numbers have different attributes, at least one must be redefined.



The DDS for these files is as follows:

JLF



```
     A          R JOINREC                   JFILE(PF1 PF2)
     A          J                           JOIN(PF1 PF2)
     A                                      JFLD(PRTNBR PRTNBR)
     A            PRTNBR         S  N       JREF(1)
     A            DESC
     A            PRICE
     A            QUANT
     A          K DESC
```

PF1

```
     A          R REC1
     A            DESC          30
     A            PRTNBR         6P 0
```

PF2

```
     A          R REC2
     A            PRTNBR         6S 0
     A            PRICE          7  2
     A            QUANT          8  0
```

RSLH281-1

In PF1, the *Prtnbr* field is a packed decimal field; in PF2, the *Prtnbr* field is a zoned decimal field. In the join logical file, they are used as join fields, and the *Prtnbr* field from PF1 is redefined to be a zoned decimal field by specifying an S in position 35 at the field level. The JREF keyword identifies which physical file the field comes from. However, the field is not included in the record format; therefore, N is specified in position 38 to make it a neither field. A program using this file would not see the field.

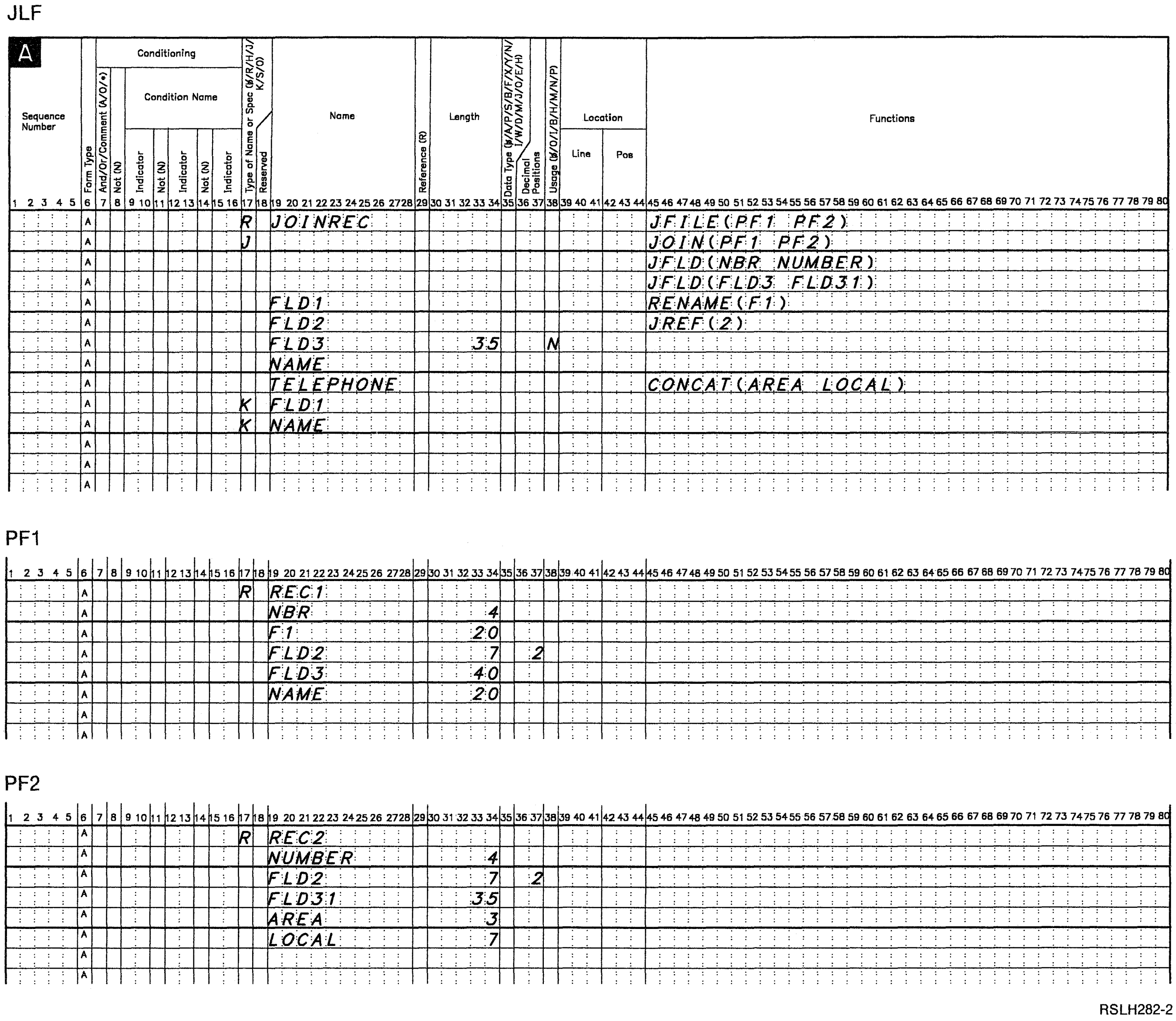
In this example, a sales clerk can type a description of a part. The program can read the join logical file for a match or a close match, and display one or more parts for the user to examine, including the description, price, and quantity. This application assumes that part numbers are not necessary to complete a customer order or to order more parts for the warehouse.

Specifying Key Fields in Join Logical Files (Example 6)

If you specify key fields in a join logical file, the following rules apply:

- The key fields must exist in the primary physical file.
- The key fields must be named in the join record format in the logical file in positions 19 through 28.
- The key fields cannot be fields defined as neither fields (N specified in position 38 for the field) in the logical file.

The following illustrates the rules for key fields:



JLF

Sequence Number	Form Type	And/Or/Comment (A/O/*)	Not (N)	Indicator	Not (N)	Indicator	Not (N)	Indicator	Type of Name or Spec (b/R/H/J/K/S/O)	Reserved	Name	Reference (R)	Length	Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)	Decimal Positions	Usage (b/O/I/B/H/M/N/P)	Location Line	Location Pos	Functions
	A								R		JOINREC								JFILE(PF1 PF2)
	A								J										JOIN(PF1 PF2)
	A																		JFLD(NBR NUMBER)
	A																		JFLD(FLD3 FLD31)
	A										FLD1								RENAME(F1)
	A										FLD2								JREF(2)
	A										FLD3		35			N			
	A										NAME								
	A										TELEPHONE								CONCAT(AREA LOCAL)
	A								K		FLD1								
	A								K		NAME								
	A																		
	A																		
	A																		

PF1

	A								R		REC1								
	A										NBR		4						
	A										F1		20						
	A										FLD2		7		2				
	A										FLD3		40						
	A										NAME		20						
	A																		
	A																		

PF2

	A								R		REC2								
	A										NUMBER		4						
	A										FLD2		7		2				
	A										FLD31		35						
	A										AREA		3						
	A										LOCAL		7						
	A																		
	A																		

RSLH282-2

The following fields **cannot** be key fields:

Nbr (not named in positions 19 through 28)
Number (not named in positions 19 through 28)
F1 (not named in positions 19 through 28)
Fld31 (comes from a secondary file)
Fld2 (comes from a secondary file)
Fld3 (is a neither field)
Area and *Local* (not named in positions 19 through 28)
Telephone (is based on fields from a secondary file)

Specifying Select/Omit Statements in Join Logical Files

If you specify select/omit statements in a join logical file, the following rules apply:

- The fields can come from any physical file the logical file uses (specified on the JFILE keyword).
- The fields you specify on the select/omit statements cannot be fields defined as neither fields (N specified in position 38 for the field).
- In some circumstances, you must specify the DYNSLT keyword when you specify select/omit statements in join logical files. For more information and examples, see the DYNSLT keyword in the *DDS Reference.*

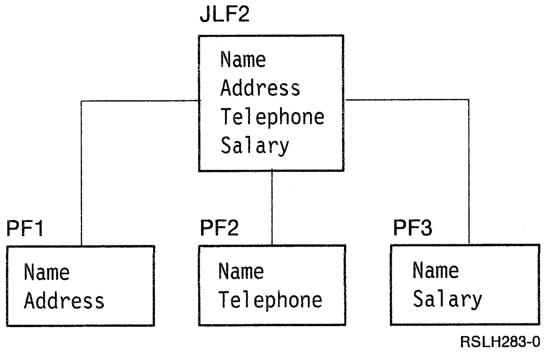
For an example showing select/omit statements in a join logical file, see "A Complex Join Logical File (Example 10)" on page 6-47.

Joining Three or More Physical Files (Example 7)

You can use a join logical file to join as many as 32 physical files. These files must be specified on the JFILE keyword. The first file specified on the JFILE keyword is the primary file; the other files are all secondary files.

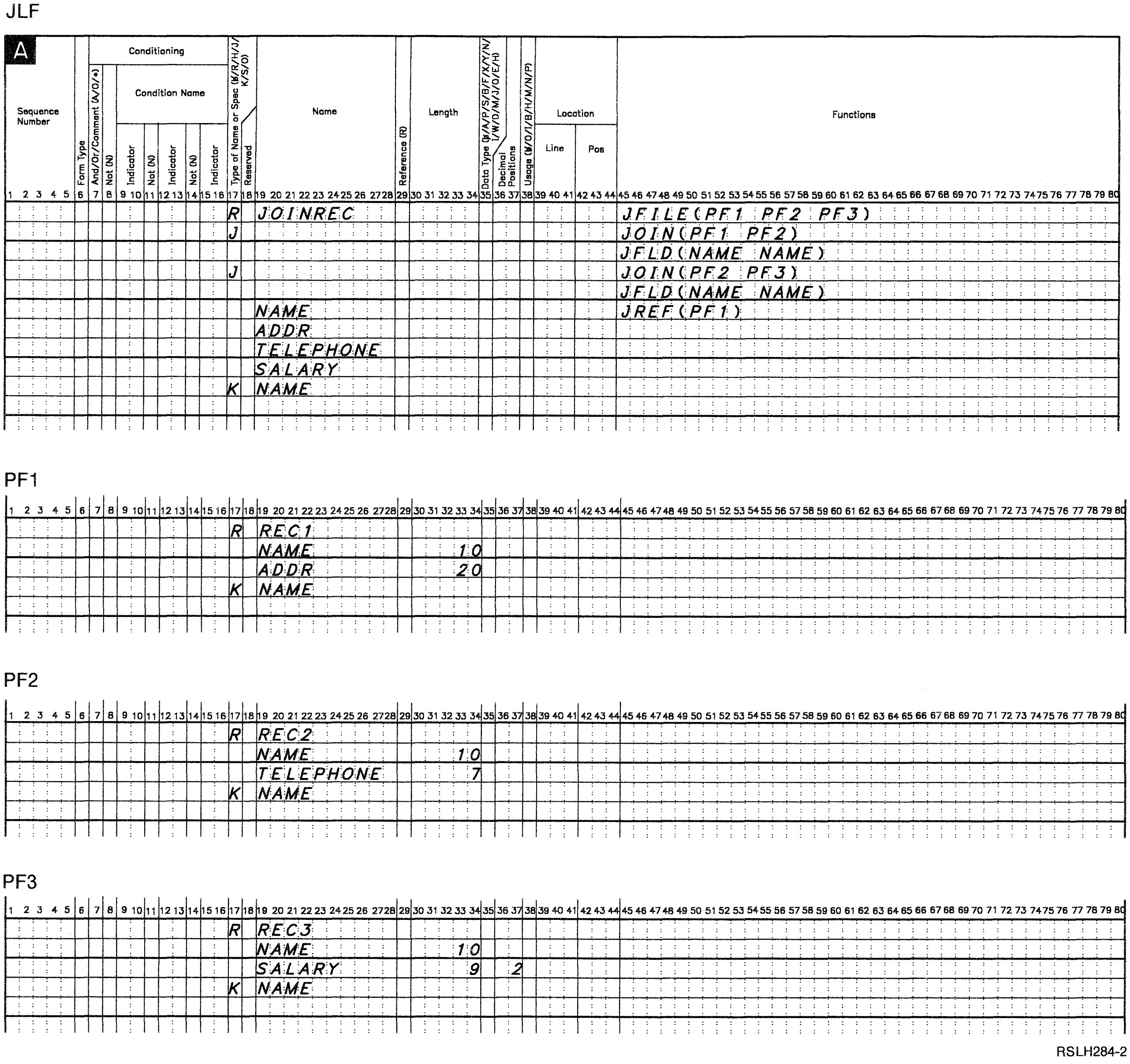
The physical files must be joined in pairs, with each pair described by a join specification. Each join specification must have one or more join fields identified.

The following shows the fields in the files and one field common to all the physical files in the logical file:



In this example, the *Name* field is common to all the physical files (PF1, PF2, and PF3), and serves as the join field.

The following shows the DDS for the physical and logical files:



JLF

```
                R JOINREC                   JFILE(PF1 PF2 PF3)
                J                           JOIN(PF1 PF2)
                                            JFLD(NAME NAME)
                J                           JOIN(PF2 PF3)
                                            JFLD(NAME NAME)
                  NAME                      JREF(PF1)
                  ADDR
                  TELEPHONE
                  SALARY
                K NAME
```

PF1

```
                R REC1
                  NAME          10
                  ADDR          20
                K NAME
```

PF2

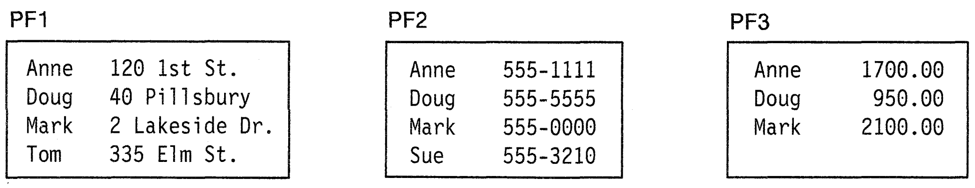
```
                R REC2
                  NAME          10
                  TELEPHONE      7
                K NAME
```

PF3

```
                R REC3
                  NAME          10
                  SALARY         9  2
                K NAME
```

RSLH284-2

Assume the physical files have the following records:



PF1

Anne	120 1st St.
Doug	40 Pillsbury
Mark	2 Lakeside Dr.
Tom	335 Elm St.

PF2

Anne	555-1111
Doug	555-5555
Mark	555-0000
Sue	555-3210

PF3

Anne	1700.00
Doug	950.00
Mark	2100.00

RSLH285-0

The program reads the following logical file records:

JLF

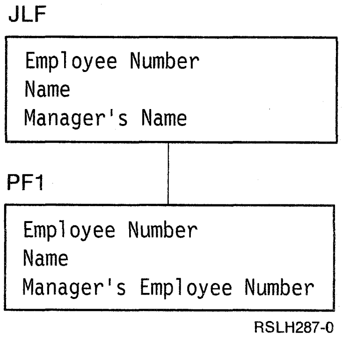
```
Anne   120 1st St.       555-1111    1700.00
Doug   40 Pillsbury      555-5555     950.50
Mark   2 Lakeside Dr.    555-0000    2100.00
```

RSLH286-0

No record is returned for Tom because a record is not found for him in PF2 and PF3 and the JDFTVAL keyword is not specified. No record is returned for Sue because the primary file has no record for Sue.

Joining a Physical File to Itself (Example 8)

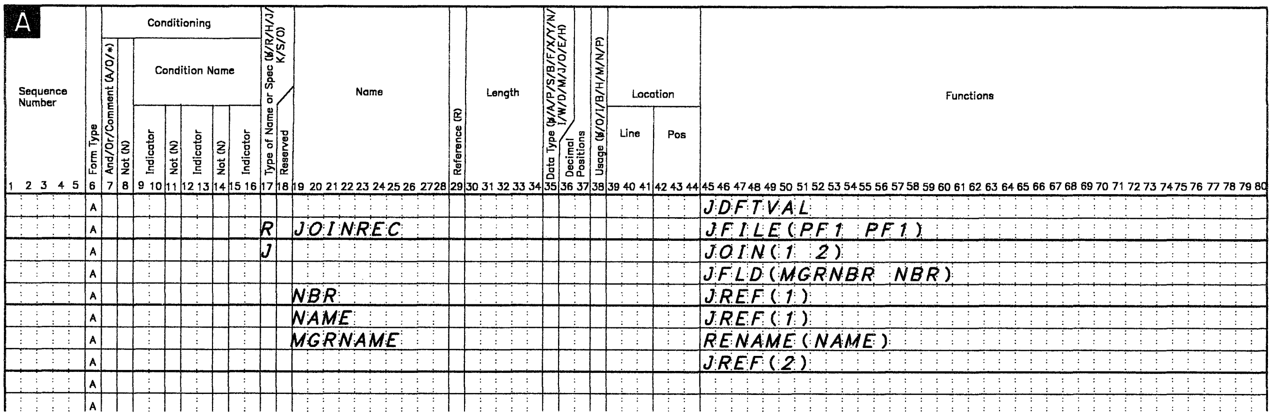
You can join a physical file to itself to read records that are formed by combining two or more records from the physical file itself. The following example shows how:



RSLH287-0

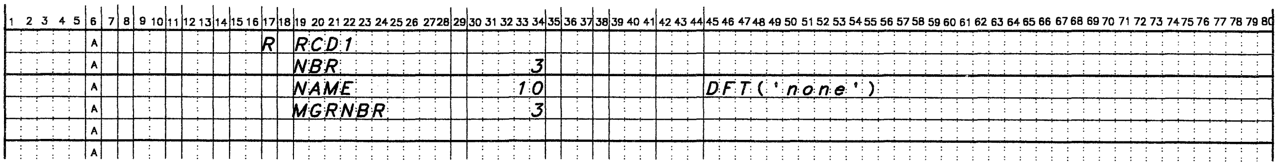
The following shows the DDS for these files:

JLF



```
     A                                      JDFTVAL
     A          R JOINREC                   JFILE(PF1 PF1)
     A          J                           JOIN(1 2)
     A                                      JFLD(MGRNBR NBR)
     A            NBR                       JREF(1)
     A            NAME                      JREF(1)
     A            MGRNAME                   RENAME(NAME)
     A                                      JREF(2)
     A
     A
```

PF1



```
     A          R RCD1
     A            NBR            3
     A            NAME          10          DFT('none')
     A            MGRNBR         3
     A
     A
```

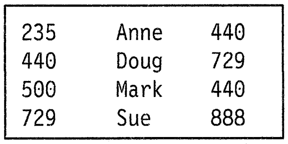
RSLH288-1

Notes:

1. Relative file numbers must be specified on the JOIN keyword because the same file name is specified twice on the JFILE keyword. Relative file number 1 refers to the first physical file specified on the JFILE keyword, 2 refers to the second, and so forth.
2. With the same physical files specified on the JFILE keyword, the JREF keyword is required for each field specified at the field level.

Assume the following records are contained in PF1:

PF1

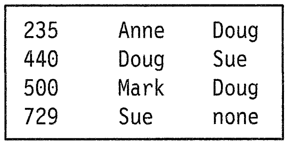


235	Anne	440
440	Doug	729
500	Mark	440
729	Sue	888

RSLH289-0

The program reads the following logical file records:

JLF



235	Anne	Doug
440	Doug	Sue
500	Mark	Doug
729	Sue	none

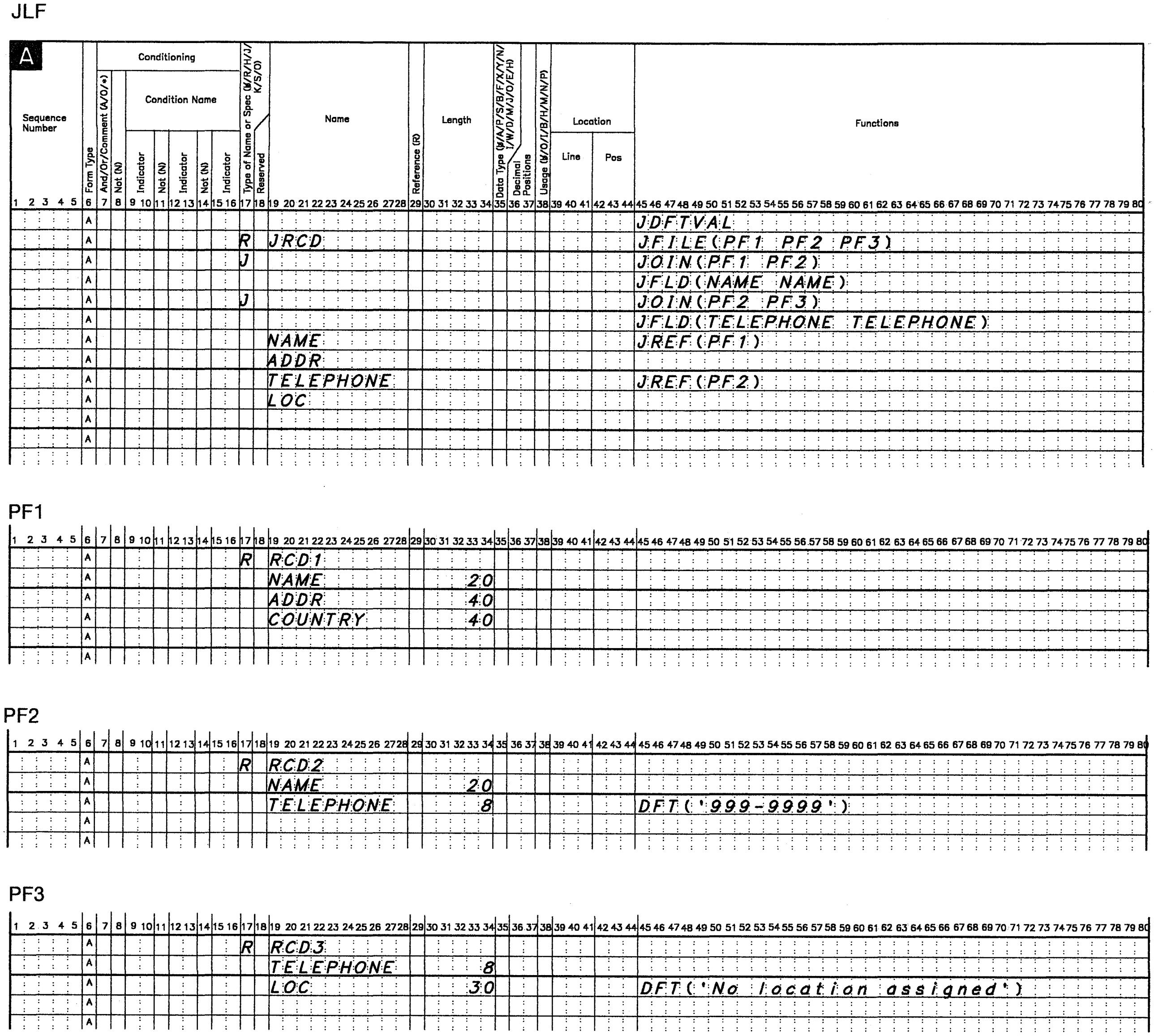
RSLH290-0

Note that a record is returned for the manager name of Sue because the JDFTVAL keyword was specified. Also note that the value none is returned because the DFT keyword was used on the *Name* field in the PF1 physical file.

Using Default Data for Missing Records from Secondary Files (Example 9)

If you are joining more than two files, and you specify the JDFTVAL keyword, the default value supplied by the system for a join field missing from a secondary file is used to join to other secondary files. If the DFT keyword is specified in the secondary file, the value specified for the DFT keyword is used in the logical file.

The DDS for the files is as follows:



JLF

```
     A                                      JDFTVAL
     A          R JRCD                      JFILE(PF1 PF2 PF3)
     A          J                           JOIN(PF1 PF2)
     A                                      JFLD(NAME NAME)
     A          J                           JOIN(PF2 PF3)
     A                                      JFLD(TELEPHONE TELEPHONE)
     A            NAME                      JREF(PF1)
     A            ADDR
     A            TELEPHONE                 JREF(PF2)
     A            LOC
     A
     A
```

PF1

```
     A          R RCD1
     A            NAME          20
     A            ADDR          40
     A            COUNTRY       40
     A
     A
```

PF2

```
     A          R RCD2
     A            NAME          20
     A            TELEPHONE      8          DFT('999-9999')
     A
     A
```

PF3

```
     A          R RCD3
     A            TELEPHONE      8
     A            LOC           30          DFT('No location assigned')
     A
     A
```

RSLH291-2

Assume that PF1, PF2, and PF3 have the following records:

PF1

Anne	120 1st St.	USA
Doug	40 Pillsbury	Canada
Mark	2 Lakeside Dr.	Canada
Sue	120 Broadway	USA

PF2

Anne	555-1234
Doug	555-2222
Sue	555-1144

No telephone number was found for Mark in PF2.

PF3

555-1234	Room 312
555-2222	Main lobby
999-9999	No telephone number

No record for telephone number 555-1144 was found in PF3.

RSLH292-1

With JDFTVAL specified in the join logical file, the program reads the following logical file records:

Anne	120 1st St.	555-1234	Room 312
Doug	40 Pillsbury	555-2222	Main lobby
Mark	2 Lakeside Dr.	999-9999	No telephone number
Sue	120 Broadway	555-1144	No location assigned

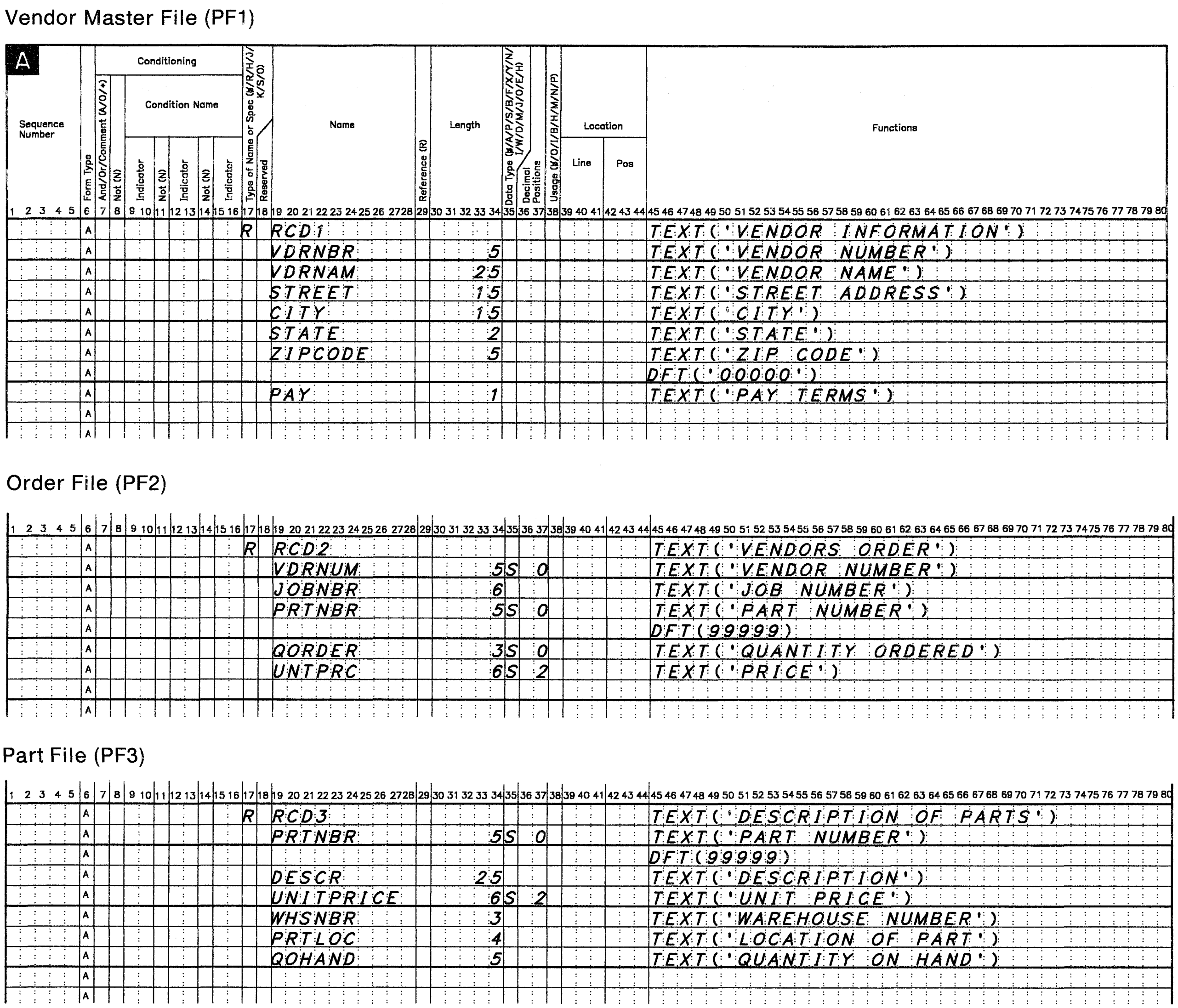
RSLH293-0

In this example, complete data is found for Anne and Doug. However, part of the data is missing for Mark and Sue.

- PF2 is missing a record for Mark because he has no telephone number. The default value for the *Telephone* field in PF2 is defined as 999-9999 using the DFT keyword. In this example, therefore, 999-9999 is the telephone number returned when no telephone number is assigned. The JDFTVAL keyword specified in the join logical file causes the default value for the *Telephone* field (which is 999-9999) in PF2 to be used to match with a record in PF3. (In PF3, a record is included to show a description for telephone number 999-9999.) Without the JDFTVAL keyword, no record would be returned for Mark.
- Sue's telephone number is not yet assigned a location; therefore, a record for 555-1144 is missing in PF3. Without JDFTVAL specified, no record would be returned for Sue. With JDFTVAL specified, the system supplies the default value specified on the DFT keyword in PF3 the *Loc* field (which is `No location assigned`).

A Complex Join Logical File (Example 10)

The following example shows a more complex join logical file. Assume the data is in the following three physical files:



Vendor Master File (PF1)

Form Type	Type of Name or Spec	Name	Length	Data Type	Decimal Positions	Functions
A	R	RCD1				TEXT('VENDOR INFORMATION')
A		VDRNBR	5			TEXT('VENDOR NUMBER')
A		VDRNAM	25			TEXT('VENDOR NAME')
A		STREET	15			TEXT('STREET ADDRESS')
A		CITY	15			TEXT('CITY')
A		STATE	2			TEXT('STATE')
A		ZIPCODE	5			TEXT('ZIP CODE')
A						DFT('00000')
A		PAY	1			TEXT('PAY TERMS')
A						
A						

Order File (PF2)

Form Type	Type of Name or Spec	Name	Length	Data Type	Decimal Positions	Functions
A	R	RCD2				TEXT('VENDORS ORDER')
A		VDRNUM	5	S	0	TEXT('VENDOR NUMBER')
A		JOBNBR	6			TEXT('JOB NUMBER')
A		PRTNBR	5	S	0	TEXT('PART NUMBER')
A						DFT(99999)
A		QORDER	3	S	0	TEXT('QUANTITY ORDERED')
A		UNTPRC	6	S	2	TEXT('PRICE')
A						
A						

Part File (PF3)

Form Type	Type of Name or Spec	Name	Length	Data Type	Decimal Positions	Functions
A	R	RCD3				TEXT('DESCRIPTION OF PARTS')
A		PRTNBR	5	S	0	TEXT('PART NUMBER')
A						DFT(99999)
A		DESCR	25			TEXT('DESCRIPTION')
A		UNITPRICE	6	S	2	TEXT('UNIT PRICE')
A		WHSNBR	3			TEXT('WAREHOUSE NUMBER')
A		PRTLOC	4			TEXT('LOCATION OF PART')
A		QOHAND	5			TEXT('QUANTITY ON HAND')
A						
A						

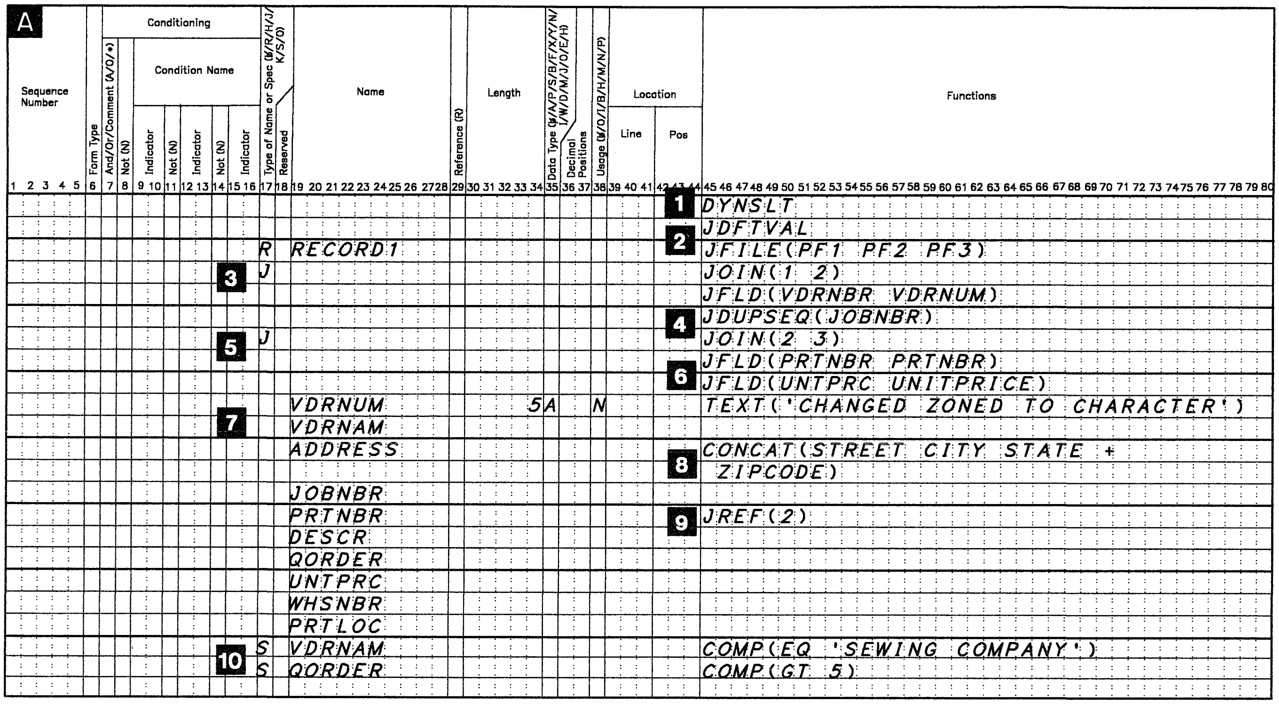
RSLH294-1

The join logical file record format should contain the following fields:

Vdrnam (vendor name)
Street, City, State, and *Zipcode* (vendor address)
Jobnbr (job number)
Prtnbr (part number)
Descr (description of part)
Qorder (quantity ordered)
Untprc (unit price)
Whsnbr (warehouse number)
Prtloc (location of part)

The DDS for this join logical file is as follows:

Join Logical File (JLF)



1
2
3
4
5
6
7
8
9
10

```
                                            DYNSLT
                                            JDFTVAL
                R RECORD1                   JFILE(PF1 PF2 PF3)
                J                           JOIN(1 2)
                                            JFLD(VDRNBR VDRNUM)
                                            JDUPSEQ(JOBNBR)
                J                           JOIN(2 3)
                                            JFLD(PRTNBR PRTNBR)
                                            JFLD(UNTPRC UNITPRICE)
                  VDRNUM         5A  N      TEXT('CHANGED ZONED TO CHARACTER')
                  VDRNAM
                  ADDRESS                   CONCAT(STREET CITY STATE +
                                             ZIPCODE)
                  JOBNBR
                  PRTNBR                    JREF(2)
                  DESCR
                  QORDER
                  UNTPRC
                  WHSNBR
                  PRTLOC
                S VDRNAM                    COMP(EQ 'SEWING COMPANY')
                S QORDER                    COMP(GT 5)
```

RSLH295-1

1 The DYNSLT keyword is required because the JDFTVAL keyword and select fields are specified.

2 The JDFTVAL keyword is specified to pick up default values in physical files.

3 First join specification.

4 The JDUPSEQ keyword is specified because duplicate vendor numbers occur in PF2.

5 Second join specification.

6 Two JFLD keywords are specified to ensure the correct records are joined from the PF2 and PF3 files.

7 The *Vdrnum* field is redefined from zoned decimal to character (because it is used as a join field and it does not have the same attributes in PF1 and PF2).

8 The CONCAT keyword concatenates four fields from the same physical file into one field.

9 The JREF keyword must be specified because the *Prtnbr* field exists in two physical files and you want to use the one in PF2.

10 The select/omit fields are *Vdrnam* and *Qorder*. (Note that they come from two different physical files.)

Performance Considerations

You can do the following to improve the performance of join logical files:

- If the physical files you are joining have a different number of records, specify the physical file with fewest records first (first parameter following the JOIN keyword).
- Consider using the DYNSLT keyword. See "Dynamic Select/Omit" on page 6-13 for more details.
- Consider describing your join logical file so it can automatically share an existing access path. See "Using Existing Access Paths" on page 6-14 for more details.

  Note: Join logical files always have access paths using the second field of the pair of fields specified in the JFLD keyword. This field acts like a key field in simple logical files. If an access path does not already exist, the access path is implicitly created with immediate maintenance.

Data Integrity Considerations

Unless you have a lock on the physical files used by the join logical file, the following can occur:

- Your program reads a record for which there are two or more records in a secondary file. The system supplies one record to your program.
- Another program updates the record in the primary file that your program has just read, changing the join field.
- Your program issues another read request. The system supplies the next record based on the current (new) value of the join field in the primary file.

These same considerations apply to secondary files as well.

Summary of Rules for Join Logical Files

Requirements

The principal requirements for join logical files are:

- Each join logical file must have:
  - Only one record format, with the JFILE keyword specified for it.
  - At least two physical file names specified on the JFILE keyword. (The physical file names on the JFILE keyword do not have to be different files.)
  - At least one join specification (J in position 17 with the JFLD keyword specified).
  - A maximum of 31 secondary files.
  - At least one field name with field use other than N (neither) at the field level.
- If only two physical files are specified for the JFILE keyword, the JOIN keyword is not required. Only one join specification can be included, and it joins the two physical files.
- If more than two physical files are specified for the JFILE keyword, the following rules apply:
  - The primary file must be the first file of the pair of files specified on the first JOIN keyword (the primary file can also be the first of the pair of files specified on other JOIN keywords).

    Note: Relative file numbers must be specified on the JOIN keyword and any JREF keyword when the same file name is specified twice on the JFILE keyword.

  - Every secondary file must be specified only once as the second file of the pair of files on the JOIN keyword. This means that for every secondary file on the JFILE keyword, one join specification must be included (two secondary files would mean two join specifications, three secondary files would mean three join specifications).
  - The order in which secondary files appear in join specifications must match the order in which they are specified on the JFILE keyword.

Join Fields

The rules to remember about join fields are:

- Every physical file you are joining must be joined to another physical file by at least one join field. A join field is a field specified as a parameter value on the JFLD keyword in a join specification.
- Join fields (specified on the JFLD keyword) must have identical attributes (length, data type, and decimal positions) or be redefined in the record format of the join logical file to have the same attributes.
- Join fields need not be specified in the record format of the join logical file (unless you must redefine one or both so that their attributes are identical).
- If you redefine a join field, you can specify N in position 38 (making it a neither field) to prevent a program using the join logical file from using the redefined field.

Fields in Join Logical Files

The rules to remember about fields in join logical files are:

- Fields in a record format for a join logical file must exist in one of the physical files used by the logical file or, if CONCAT, RENAME, TRNTBL, or SST is specified for the field, be a result of fields in one of the physical files.
- Fields specified as parameter values on the CONCAT keyword must be from the same physical file. If the first field name specified on the CONCAT keyword is not unique among the physical files, you must specify the JREF keyword for that field to identify which file contains the field descriptions you want to use.
- If a field name in the record format for a join logical file is specified in more than one of the physical files, you must uniquely specify on the JREF keyword which file the field comes from.
- Key fields, if specified, must come from the primary file. Key fields in the join logical file need not be key fields in the primary file.
- Select/omit fields can come from any physical file used by the logical file, but in some circumstances the DYNSLT keyword is required.
- If specified, key fields and select/omit fields must be defined in the record format.
- Relative file numbers must be used for the JOIN and JREF keywords if the name of the physical file is specified more than once on the JFILE keyword.

Miscellaneous

Other rules to keep in mind when using join logical files are:

- Join logical files are read-only files.
- Join record formats cannot be shared, and cannot share other record formats.
- The following are not allowed in a join logical file:
  - The REFACCPTH and FORMAT keywords
  - Both fields (B specified in position 38)

Chapter 7. Database Security

This chapter describes some of the database file security functions. The topics covered include database file security, public authority considerations, restricting the ability to change or delete any data in a file, and using logical files to secure data. For more information about using the security function on the AS/400 system, see *Security Concepts and Planning.*

File and Data Authority

The following describes the types of authority that can be granted to a user for a database file.

Object Operational Authority

Object operational authority is required to:

- Open the file for processing. (You must also have at least one data authority.)
- Compile a program which uses the file description.
- Display descriptive information about active members of a file.
- Open the file for query processing. For example, the Open Query File (OPNQRYF) command opens a file for query processing.

Note: You must also have the appropriate data authorities required by the options specified on the open operation.

Object Existence Authority

Object existence authority is required to:

- Delete the file.
- Save, restore, and free the storage of the file. If the object existence authority has not been explicitly granted to the user, the *SAVSYS special user authority allows the user to save, restore, and free the storage of a file. *SAVSYS is not the same as object existence authority.
- Remove members from the file.
- Transfer ownership of the file.

Note: All these functions except save/restore also require object operational authority to the file.

Object Management Authority

Object management authority is required to:

- Create a logical file with a keyed access path (object management authority is required for the physical file referred to by the logical file).
- Grant and revoke authority. You can grant and revoke only the authority that you already have. (You must also have object operational authority to the file.)
- Change the file.
- Add members to the file. (The owner of the file becomes the owner of the new member.)
- Change the member in the file.
- Move the file.
- Rename the file.
- Rename a member of the file.
- Clear a member of the file. (Delete data authority is also required.)

- Initialize a member of the file. (Add data authority is also required to initialize with default records; delete data authority is required to initialize with deleted records.)
- Reorganize a member of the file. (All data authorities are also required.)

Data Authorities

Data authorities can be granted only to physical files. Logical files use the data authority granted to the physical files associated with the logical file.

Read Authority: You can read the records in the file.

Add Authority: You can add new records to the file.

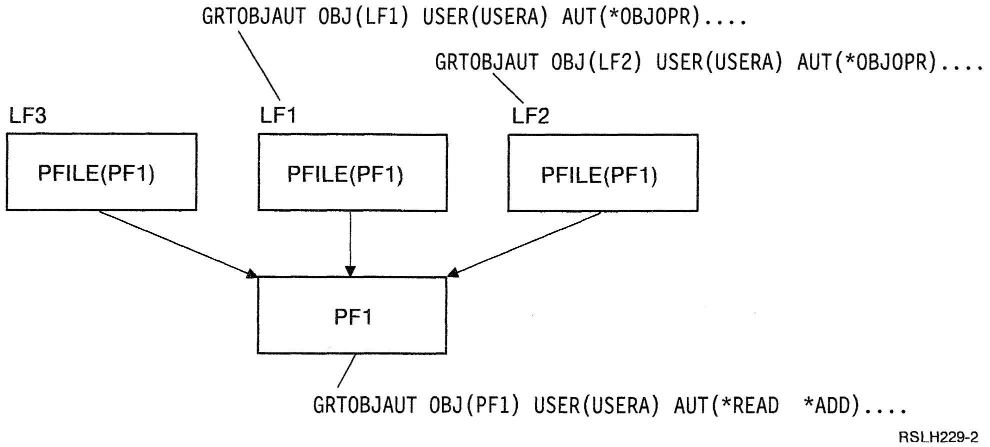
Update Authority: You can update existing records. (To read a record for update, you must also have read authority.)

Delete Authority: You can delete existing records. (To read a record for deletion, you must also have read authority.)

Normally, the authority you have to the data in the file is not verified until you actually perform the input/output operation. However, the Open Query File (OPNQRYF) and Open Database File (OPNDBF) commands also verify data authority when the file is opened.

For a physical file, authority to the file and to the data in the file can be granted. For a logical file, authority to the file can be granted, but because logical files contain no data, data authority cannot be granted. If object operational authority is not granted to a user for a physical file, that user cannot open the physical file.

The following example shows the relationship between authority granted for logical files and the physical files used by the logical file. The logical files LF1, LF2, and LF3 are based on the physical file PF1. USERA has read (*READ) and add (*ADD) authority to the data in PF1 and object operational authority for LF1 and LF2. This means that USERA cannot open PF1 or use its data directly in any way because the user does *not* have object operational authority (*OBJOPR) to PF1; USERA can open LF1 and LF2 and read records from and add records to PF1 through LF1 and LF2. Note that the user was not given authority for LF3 and, therefore, cannot use it.


GRTOBJAUT OBJ(LF1) USER(USERA) AUT(*OBJOPR)....
GRTOBJAUT OBJ(LF2) USER(USERA) AUT(*OBJOPR)....
LF3
LF1
LF2
PFILE(PF1)
PFILE(PF1)
PFILE(PF1)
PF1
GRTOBJAUT OBJ(PF1) USER(USERA) AUT(*READ *ADD)....
RSLH229-2

Public Authority

When you create a file, you can specify public authority through the AUT parameter on the create command. **Public authority** is authority available to any user who does not have specific authority to the file or who is not a member of a group that has specific authority to the file. Public authority is the last authority check made. That is, if the user has specific authority to a file or the user is a member of a group with specific authority, then the public authority is not checked. Public authority can be specified as:

- *CHANGE. All users that do not have specific user or group authority to the file have authority to change data in the file. The *CHANGE value is the default public authority.
- *USE. All users that do not have specific user or group authority to the file have authority to read data in the file.
- *EXCLUDE. Only the owner, security officer, users with specific authority, or users who are members of a group with specific authority can use the file.
- *ALL. All users that do not have specific user or group authority to the file have all data authorities along with object operational, object management, and object existence authorities.
- Authorization list name. An authorization list is a list of users and their authorities. The list allows users and their different authorities to be grouped together.

Note: When creating a logical file, no data authorities are granted. Consequently, *CHANGE is the same as *USE, and *ALL does not grant any data authority.

You can use the Edit Object Authority (EDTOBJAUT), Grant Object Authority (GRTOBJAUT), or Revoke Object Authority (RVKOBJAUT) commands to grant or revoke the public authority of a file.

Database File Capabilities

File capabilities are used to control which input/output operations are allowed for a database file independent of database file authority.

When you create a physical file, you can specify if the file is update-capable and delete-capable by using the ALWUPD and ALWDLT parameters on the Create Physical File (CRTPF) and Create Source Physical File (CRTSRCPF) commands. By creating a file that is not update-capable and not delete-capable, you can effectively enforce an environment where data cannot be changed or deleted from a file once the data is written.

File capabilities cannot be explicitly set for logical files. The file capabilities of a logical file are determined by the file capabilities of the physical files it is based on.

You cannot change file capabilities after the file is created. You must delete the file then recreate it with the desired capability. The Display File Description (DSPFD) command can be used to determine the capability of a file.

Using Logical Files to Secure Data

You can use a logical file to prevent a field in a physical file from being viewed. This is accomplished by describing a logical file record format that does not include fields you do not want the user to see. For more information about this subject, see "Describing Logical File Record Formats" on page 6-1.

You can also use a logical file to prevent one or more fields from being changed in a physical file by specifying, for those fields you want to protect, an I (input only) in position 38 of the *DDS Coding Form*. For more information about this subject, see "Describing Field Use for Logical Files" on page 6-4.

You can use a logical file to secure records in a physical file based on the contents of one or more fields in that record. To secure records based on the contents of a field, use the select and omit keywords when describing the logical file. For more information about this subject, see "Selecting and Omitting Records Using Logical Files" on page 6-9.

Part 3. Processing Database Files in Programs

The chapters in this part include information on processing database files in your programs. This includes information on planning how the file will be used in the program or job and improving the performance of your program. Descriptions of the file processing parameters and run-time options that you can specify for more efficient file processing are included in this section.

Another topic covered in this part is sharing database files across jobs so that they can be accessed by many users at the same time. Locks on files, records, or members that can prevent them from being shared across jobs are also discussed.

Using the Open Query File (OPNQRYF) command and the Open Database File (OPNDBF) command to open database file members in a program is discussed. Examples, performance considerations, and guidelines to follow when writing a high-level language program are also included. Also, typical errors that can occur are discussed.

Finally, basic database file operations are discussed. This discussion includes setting a position in the database file, and reading, updating, adding, and deleting records in a database file. A description of several ways to read database records is also included. Information on updating discusses how to change an existing database record in a logical or physical file. Information on adding a new record to a physical database member using the write operation is included. This section also includes ways you can close a database file when your program completes processing a database file member, disconnecting your program from the file. Messages to monitor when handling database file errors in a program are also discussed.

Chapter 8. Run Time Considerations

Before a file is opened for processing, you should consider how you will use the file in the program and job. A better understanding of the run-time file processing parameters can help you avoid unexpected results. In addition, you might improve the performance of your program.

When a file is opened, the attributes in the database file description are merged with the parameters in the program. Normally, most of the information the system needs for your program to open and process the file is found in the file attributes and in the application program itself.

Sometimes, however, it is necessary to override the processing parameters found in the file and in the program. For example, if you want to process a member of the file other than the first member, you need a way to tell the system to use the member you want to process. The Override with Database File (OVRDBF) command allows you to do this. The OVRDBF command also allows you to specify processing parameters that can improve the performance of your job, but that cannot be specified in the file attributes or in the program. The OVRDBF command parameters take precedence over the file and program attributes.

This chapter describes the file processing parameters. The parameter values are determined by the high-level language program, the file attributes, and any open or override commands processed before the high-level language program is called. A summary of these parameters and where you specify them can be found in "Run Time Summary" on page 8-16. For more information about processing parameters from commands, see the *CL Reference* for the following commands:

- Create Physical File (CRTPF)
- Create Logical File (CRTLF)
- Create Source Physical File (CRTSRCPF)
- Add Physical File Member (ADDPFM)
- Add Logical File Member (ADDLFM)
- Change Physical File (CHGPF)
- Change Physical File Member (CHGPFM)
- Change Logical File (CHGLF)
- Change Logical File Member (CHGLFM)
- Change Source Physical File (CHGSRCPF)
- Override with Database File (OVRDBF)
- Open Database File (OPNDBF)
- Open Query File (OPNQRYF)
- Close File (CLOF)

File and Member Name

FILE and MBR Parameter. Before you can process data in a database file, you must identify which file and member you want to use. Normally, you specify the file name and, optionally, the member name in your high-level language program. The system then uses this name when your program requests the file to be opened. To override the file name specified in your program and open a different file, you can use the TOFILE parameter on the Override with Database File (OVRDBF) command. If no member name is specified in your program, the first member of the file (as defined by the creation date and time) is processed.

If the member name cannot be specified in the high-level language program (some high-level languages do not allow a member name), or you want a member other than the first member, you can use an Override with Database File (OVRDBF) command or an open command (OPNDBF or OPNQRYF) to specify the file and member you want to process (using the FILE and MBR parameters).

To process all the members of a file, use the OVRDBF command with the MBR(*ALL) parameter specified. For example, if FILEX has three members and you want to process all the members, you can specify:

```
OVRDBF   FILE(FILEX)  MBR(*ALL)
```

If you specify MBR(*ALL) on the OVRDBF command, your program reads the members in the order they were created. For each member, your program reads the records in keyed or arrival sequence, depending on whether the file is an arrival sequence or keyed sequence file.

File Processing Options

The following section describes several run-time processing options including identifying what file operations will be used by the program, the starting file position, ignoring the keyed sequence access path, specifying how to handle end of file processing, and identifying the length of the record in the file.

Specifying the Type of Processing

OPTION Parameter. When you use a file in a program, the system needs to know what types of operations you plan to use for that file. For example, the system needs to know if you plan to just read data in the file or if you plan to read and update the data. The valid operation options are: input, output, update, and delete. The system determines the options you are using from information you specify in your high-level language program or from the OPTION parameter on the Open Database File (OPNDBF) and Open Query File (OPNQRYF) commands.

The system uses the options to determine which operations are allowed in your program. For example, if you open a file for input only and your program tries an output operation, your program receives an error.

Normally, the system verifies that you have the required data authority when you do an input/output operation in your program. However, when you use the Open Query File (OPNQRYF) or Open Database File (OPNDBF) commands, the system verifies at the time the file is opened that you have the required data authority to perform the operations specified on the OPTION parameter. For more information about data authority, see "Data Authorities" on page 7-2.

The system also uses these options to determine the locks to use to protect the data integrity of the files and records being processed by your program. For more information on locks, see "Sharing Database Files Across Jobs" on page 8-5.

Specifying the Initial File Position

POSITION Parameter. The system needs to know where it should start processing the file after it is opened. The default is to start just before the first record in the file (the first sequential read operation will read the first record). But, you can tell the system to start at the end of the file, or at a certain record in the middle of the file using the Override with Database File (OVRDBF) command. You can also dynamically set a position for the file in your program. For more information on setting position for a file in a program, see "Setting a Position in the File" on page 10-1.

Ignoring the Keyed Sequence Access Path

ACCPTH Parameter. When you process a file with a keyed sequence access path, you normally want to use that access path to retrieve the data. The system automatically uses the keyed sequence access path if a key field is defined for the file. However, sometimes you can achieve better performance by ignoring the keyed sequence access path and processing the file in arrival sequence.

You can tell the system to ignore the keyed sequence access path in some high-level languages, or on the Open Database File (OPNDBF) command. When you ignore the keyed sequence access path, operations that read data by key are not allowed. Operations are done sequentially along the arrival sequence access path. (If this option is specified for a logical file with select/omit values defined, the arrival sequence access path is used and only those records meeting the select/omit values are returned to the program. The processing is done as if the DYNSLT keyword was specified for the file.)

Note: You cannot ignore the keyed sequence access path for logical file members that are based on more than one physical file member.

Delaying End of File Processing

EOFDLY Parameter. When you are reading a database file and your program reaches the end of the data, the system normally signals your program that there is no more data to read. Occasionally, instead of telling the program there is no more data, you might want the system to hold your program until more data arrives in the file. When more data arrives in the file, the program can read the newly arrived records. If you need that type of processing, you can use the EOFDLY parameter on the Override with Database File (OVRDBF) command. For more information on this parameter, see "Waiting for More Records When End of File Is Reached" on page 10-4.

Specifying the Record Length

The system needs to know the length of the record your program will be processing, but you do not have to specify record length in your program. The system automatically determines this information from the attributes and description of the file named in your program. However, as an option, you can specify the length of the record in your high-level language program.

If the file that is opened contains records that are longer than the length specified in the program, the system allocates a storage area to match the file member's record length and this option is ignored. In this case, the entire record is passed to the program. (However, some high-level languages allow you to access only that portion of the record defined by the record length specified in the program.) If the file that is opened contains records that are less than the length specified in the program, the system allocates a storage area for the program-specified record length. The program can use the extra storage space, but only the record lengths defined for the file member are used for input/output operations.

Ignoring Record Formats

When you use a multiple format logical file, the system assumes you want to use all formats defined for that file. However, if you do not want to use all of the formats, you can specify which formats you want to use and which ones you want to ignore. If you do not use this option to ignore formats, your program can process all formats defined in the file. For more information about this processing option, see your high-level language guide.

Determining if Duplicate Keys Exist

DUPKEYCHK Parameter. The set of keyed sequence access paths used to determine if the key is a duplicate key differs depending on the I/O operation that is performed.

For input operations (reads), the keyed sequence access path used is the one that the file is opened with. Any other keyed sequence access paths that can exist over the physical file are not considered. Also, any records in the keyed sequence access path omitted because of select/omit specifications are not considered when deciding if the key operation is a duplicate.

For output (write) and update operations, all non-unique keyed sequence access paths of *IMMED maintenance that exist over the physical file are searched to determine if the key for this output or update operation is a duplicate. Those keyed sequence access paths that have *REBLD or *DLY maintenance are only considered if the file to which that access path pertains is open at the time the feedback is desired.

When you process a keyed file with a COBOL program, you can specify duplicate key feedback to be returned to your program through the COBOL language, or on the Open Database File (OPNDBF) or Open Query File (OPNQRYF) commands. However, in COBOL having duplicate key feedback returned can cause a decline in performance.

Data Recovery and Integrity

The following section describes data integrity run-time considerations.

Protecting Your File with the Journal and Commitment Control

COMMIT Parameter. Journaling and commitment control are the preferred methods for data and transaction recovery on the AS/400 system. Database file journaling is started by running the Start Journaling Physical File (STRJRNPF) command for the file. Access path journaling is started by running the Start Journaling Access Path (STRJRNAP) command for the file. You tell the system that you want your files to run under commitment control through the Start Commitment Control (STRCMTCTL) command and through high-level language specifications. You can also specify the commitment control (COMMIT) parameter on the Open Database File (OPNDBF) and Open Query File (OPNQRYF) commands. For more information on journaling and commitment control, see Chapter 16.

Writing Data and Access Paths to Auxiliary Storage

FRCRATIO and FRCACCPTH Parameters. Normally, the AS/400 integrated database management system determines when to write changed data from main storage to auxiliary storage. If you want to control when database changes are written to auxiliary storage, you can use the force write ratio (FRCRATIO) parameter on either the create, change, or override database file commands, and the force access path (FRCACCPTH) parameter on the create and change database file commands. Using the FRCRATIO and FRCACCPTH parameters have performance and recovery considerations for your system. To understand these considerations, see Chapter 16.

Checking Changes to the Record Format Description

LVLCHK Parameter. The system checks, when you open the file, if the description of the record format you are using was changed since the program was compiled to an extent that your program cannot process the file. The system normally notifies your program of this condition. This condition is known as a level check. When you use the create or change file commands, you can specify that you want level checking. You can also override the level check attribute defined for the file using the LVLCHK parameter on the the Override with Database File (OVRDBF) command. For more information about this parameter, see "Effect of Changing Fields in a File Description" on page 14-1.

Checking for the File's Expiration Date

EXPDATE and EXPCHK Parameters. The system can verify that the data in the file you specify is still current. You can specify the expiration date for a file or member using the EXPDATE parameter on the create and change file commands, and you can specify whether or not the system is to check that date using the EXPCHK parameter on the Override with Database File (OVRDBF) command. If you do check the expiration date and the current date is greater than the expiration date, a message is sent to the system operator when the file is opened.

Preventing the Job from Changing Data in the File

INHWRT Parameter. If you want to test your program, but do not want to actually change data in files used for the test, you can tell the system to not write (inhibit) any changes to the file that the program attempts to make. To inhibit any changes to the file, specify INHWRT(*YES) on the OVRDBF command.

Sharing Database Files Across Jobs

By definition, all database files can be used by many users at the same time. However, some operations can lock the file, member, or data in a member that prevent it from being shared across jobs.

Each database command or high-level language program allocates the file, member, and data that it uses. Depending on the operation requested, other users can not be able to use the allocated file, member, or records. An operation on a logical file or logical file member can allocate the file(s) and member(s) that the logical file depends on for data or an access path.

For example, the open of a logical file allocates the data of the physical file member that the logical file is based on. If the program updates the logical file member, another user may not request, at the same time, that the physical file member used by that logical file member be cleared of data.

For a list of commonly used database functions and the types of locks they place on database files, see Appendix C.

Record Locks

WAITRCD Parameter. The AS/400 database has built-in integrity for records. For example, if PGMA reads a record for update, it locks that record. Another program may not read the same record for update until PGMA releases the record, but another program could read the record just for inquiry. In this way, the system ensures the integrity of the database.

The system determines the lock condition based on the type of file processing specified in your program and the operation requested. For example, if your open options include update or delete, each record read is locked so that any number of users can read the record at the same time, but only one user can update the record.

The system normally waits a specific number of seconds for a locked record to be released before it sends your program a message that it cannot get the record you are requesting. The default record wait time is 60 seconds; however, you can set your own wait time through the WAITRCD parameter on the create and change file commands and the override database file command. If your program is notified that the record it wants is locked by another operation, you can have your program take the appropriate action (for example, you could send a message to the operator that the requested record is currently unavailable).

The system automatically releases a lock when the locked record is updated or deleted. However, you can release record locks without updating the record. For information on how to release a record lock, see your high-level language guide.

Note: Using commitment control changes the record locking rules. See the *Backup and Recovery Guide* for more information on commitment control and its effect on the record locking rules.

You can use the Display Record Lock (DSPRCDLCK) command to display the current lock status (wait or held) of records for a physical file member. Depending on the parameters you specify, this command displays the lock status for a specific record or displays the lock status of all records in the member. You can also display record locks from the Work with Job (WRKJOB) display.

You can determine if your job currently has any records locked using the Check Record Lock (CHKRCDLCK) command. This command returns a message (which you can monitor) if your job has any locked records. The command is useful if you are using group jobs. For example, you could check to see if you had any records locked, before transferring to another group job. If you determined you did have records locked, your program could release those locks.

File Locks

WAITFILE Parameter. Some file operations exclusively allocate the file for the length of the operation. During the time the file is allocated exclusively, any program trying to open that file has to wait until the file is released. You can control the amount of time a program waits for the file to become available by specifying a wait time on the WAITFILE parameter of the create and change file commands and the override database file command. If you do not specifically request a wait time, the system defaults the file wait time to zero seconds.

A file is exclusively allocated when an operation that changes its attributes is run. These operations (such as move, rename, grant or revoke authority, change owner, or delete) cannot be run at the same time with any other operation on the same file or on members of that file. Other file operations (such as display, open, dump, or check object) only use the file definition, and thus lock the file less exclusively. They can run at the same time with each other and with input/output operations on a member.

Member Locks

Member operations (such as add and remove) automatically allocate the file exclusively enough to prevent other file operations from occurring at the same time. Input/output operations on the same member cannot be run, but input/output operations on other members of the same file can run at the same time. In addition, clearing or reorganizing a physical file member requires exclusive allocation of the member's data.

Record Format Data Locks

RCDFMTLCK Parameter. If you want to lock the entire set of records associated with a record format (for example, all the records in a physical file), you can use the RCDFMTLCK parameter on the OVRDBF command.

Sharing Database Files in the Same Job

SHARE Parameter. By default, the database management system lets one file be read and changed by many users at the same time. You can also share a file in the same job by opening the database file more than once in the same program, or in different programs in the same job.

The SHARE parameter on the create file, change file, and override commands allows closer sharing in a job, including sharing the file, its status, its positions, and its storage area. Sharing files in the job can improve performance by reducing the amount of main storage the job needs and by reducing the time it takes to open and close the file.

Using the SHARE(*YES) parameter lets two or more programs running in the same job share an open data path. An open data path is the path through which all input/output operations for the file are performed. In a sense, it connects the program to a file. If not specified, every time a file is opened a new open data path is created. You can specify (on the create file, change file, or override commands) that, if a file is opened more than once and an open data path is still active for it in the same job, the active ODP for the file can be used with the current open of the file, and a new open data path does not have to be created. This reduces the amount of time required to open the file after the first open, and the amount of main storage required by the job. SHARE(*YES) must be specified for the first open and other opens of the same file for the open data path to be shared. A well-designed (for performance) application normally does a shared open on files that are opened in multiple programs in the same job.

Specifying SHARE(*NO) tells the system not to share the open data path for a file. Normally, this is specified only for those files that are seldom used or require unique processing in specific programs.

Note: A high-level language program processes an open or a close operation as though the file were not being shared. You do not specify that the file is being shared in the high-level language program. You indicate that the file is being shared in the same job through the SHARE parameter. The SHARE parameter is specified only on the create, change, and override database file commands.

Open Considerations for Files Shared in a Job

The following items should be considered when opening a database file that is shared in the same job.

- You must make sure that when the shared file is opened for the first time in a job, all the open options needed for subsequent opens of the file are specified. If the open options specified for subsequent opens of a shared file do not match those specified for the first open of a shared file, an error message is sent to the program. (You can correct this by making changes to your program or to the OPNDBF or OPNQRYF command parameters, to remove any incompatible options.)

  For example, PGMA is the first program to open FILE1 in the job and PGMA only needs to read the file. However, PGMA calls PGMB which will delete records from the same shared file. Because PGMB will delete records from the shared file, PGMA will have to open the file as if it, PGMA, is also going to delete records. You can accomplish this by using the correct specifications in the high-level language. (To accomplish this in some high-level languages, you may have to use file operation statements that are never run. See your high-level language guide for more details.) You can also specify the file processing option on the OPTION parameter on the Open Database File (OPNDBF) and Open Query File (OPNQRYF) commands.
- Sometimes sharing a file within a job is not desirable. For example, one program can need records from a file in arrival sequence and another program needs the records in keyed sequence. In this situation, you should not share the open data path. You would specify SHARE(*NO) on the Override with Database File (OVRDBF) command to ensure the file was not shared within the job.
- If debug mode is entered with UPDPROD(*NO) after the first open of a shared file in a production library, subsequent shared opens of the file share the original open data path and allow the file to be changed. To prevent this, specify SHARE(*NO) on the OVRDBF command before opening files being debugged.
- The use of commitment control for the first open of a shared file requires that all subsequent shared opens also use commitment control.
- Key feedback, insert key feedback, or duplicate key feedback must be specified on the full open if any of these feedback types are desired on the subsequent shared opens of the file.
- If you did not specify a library name in the program or on the Override with Database File (OVRDBF) command (*LIBL is used), the system assumes the library list has not changed since the last open of the same shared file with *LIBL specified. If the library list has changed, you should specify the library name on the OVRDBF command to ensure the correct file is opened.
- The record length that is specified on the full open is the record length that is used on subsequent shared opens even if a larger record length value is specified on the shared opens of the file.
- Overrides and program specifications specified on the first open of the shared file are processed. Overrides and program specifications specified on subsequent opens, other than those that change the file name or the value specified on the SHARE or LVLCHK parameters on the OVRDBF command, are ignored.
- Overrides specified for a first open using the OPNQRYF command can be used to change the names of the files, libraries, and members that should be processed by the Open Query File command. Any parameter values specified on

the Override with Database command other than TOFILE, MBR, LVLCHK, and SEQONLY are ignored by the OPNQRYF command.

The CPF4123 diagnostic message lists the mismatches that can be encountered between the full open and the subsequent shared opens. These mismatches do not cause the shared open to fail.

Note: The Open Query File (OPNQRYF) command never shares an existing shared open data path in the job. If a shared ODP already exists in the job with the same file, library, and member name as the one specified on the Open Query File command, the system sends an error message and the query file is not opened.

Input/Output Considerations for Files Shared in a Job

The following items should be considered when processing a database file that is shared in the same job.

- Because only one open data path is allowed for a shared file, only one record position is maintained for all programs in the job that are sharing the file. If a program establishes a position for a record using a read or a read-for-update operation, then calls another program that also uses the shared file, the record position may have moved or a record lock been released when the called program returns to the calling program. This can cause errors in the calling program because of an unexpected record position or lock condition. When sharing files, it is your responsibility to manage the record position and record locking considerations by re-establishing position and locks.
- If a shared file is first opened for update, this does not necessarily cause every subsequent program that shares the file to request a record lock. The system determines the type of record lock needed for each program using the file. The system tries to keep lock contention to a minimum, while still ensuring data integrity.

  For example, PGMA is the first program in the job to open a shared file. PGMA intends to update records in the file; therefore, when the program reads a record for update, it will lock the record. PGMA then calls PGMB. PGMB also uses the shared file, but it does not update any records in the file; PGMB just reads records. Even though PGMA originally opened the shared file as update-capable, PGMB will not lock the records it reads, because of the processing specifications in PGMB. Thus, the system ensures data integrity, while minimizing record lock contention.

Close Considerations for Files Shared in a Job

The following items should be considered when closing a database file that is shared in the same job.

- The complete processing of a close operation (including releasing file, member, and record locks; forcing changes to auxiliary storage; and destroying the open data path) is done only when the last program to open the shared open data path closes it.
- If the file was opened with the Open Database File (OPNDBF) or the Open Query File (OPNQRYF) command, use the Close File (CLOF) command to close the file. The Reclaim Resources (RCLRSC) command can be used to close a file opened by the Open Query File (OPNQRYF) command when TYPE(*NORMAL) is specified. If TYPE(*PERM) is specified, the file remains open even if the Reclaim Resources (RCLRSC) command is run.

Examples of Closing Shared Files

The following examples show some of the things to consider when closing a file that is shared in the same job.

Example 1: Using a single set of files with similar processing options.

In this example, the user signs on and most of the programs used process the same set of files.

A CL program (PGMA) is used as the first program (to set up the application, including overrides and opening the shared files). PGMA then transfers control to PGMB, which displays the application menu. Assume, in this example, that files A, B, and C are used, and files A and B are to be shared. Files A and B were created with SHARE(*NO); therefore an OVRDBF command should precede each of the OPNDBF commands to specify the SHARE(*YES) option. File C was created with SHARE(*NO) and File C is not to be shared in this example.

```
PGMA:   PGM      /* PGMA - Initial program  */
        OVRDBF   FILE(A) SHARE(*YES)
        OVRDBF   FILE(B) SHARE(*YES)
        OPNDBF   FILE(A) OPTION(*ALL) ....
        OPNDBF   FILE(B) OPTION(*INP) ...
        TFRCTL   PGMB
        ENDPGM



PGMB:   PGM      /* PGMB - Menu program */
        DCLF     FILE(DISPLAY)
BEGIN:  SNDRCVF  RCDFMT(MENU)
        IF       (&RESPONSE *EQ '1') CALL PGM11
        IF       (&RESPONSE *EQ '2') CALL PGM12
        .
        .
        IF       (&RESPONSE *EQ '90') SIGNOFF
        GOTO     BEGIN
        ENDPGM
```

In this example, assume that:

- PGM11 opens files A and B. Because these files were opened as shared by the OPNDBF commands in PGMA, the open time is reduced. The close time is also reduced when the shared open data path is closed. The Override with Database File (OVRDBF) commands remain in effect even though control is transferred (with the Transfer Control [TFRCTL] command in PGMA) to PGMB.
- PGM12 opens files A, B, and C. File A and B are already opened as shared and the open time is reduced. Because file C is used only in this program, the file is not opened as shared.

In this example, the Close File (CLOF) was not used because only one set of files is required. When the operator signs off, the files are automatically closed. It is assumed that PGMA (the initial program) is called only at the start of the job.

Note: The display file (DISPLAY) in PGMB can also be specified as a shared file, which would improve the performance for opening the display file in any programs that use it later.

In Example 1, the OPNDBF commands are placed in a separate program (PGMA) so the other processing programs in the job run as efficiently as possible. That is, the important files used by the other programs in the job are opened in PGMA. After the files are opened by PGMA, the main processing programs (PGMB, PGM11, and PGM12) can share the files; therefore, their open and close requests will process faster. In addition, by placing the open commands (OPNDBF) in PGMA rather than in PGMB, the amount of main storage used for PGMB is reduced.

Any overrides and opens can be specified in the initial program (PGMA); then, that program can be removed from the job (for example, by transferring out of it). However, the open data paths that the program created when it opened the files remain in existence and can be used by other programs in the job.

Note the handling of the OVRDBF commands in relation to the OPNDBF commands. Overrides must be specified before the file is opened. Some of the parameters on the OVRDBF command also exist on the OPNDBF command. If conflicts arise, the OVRDBF value is used.

Example 2: Using multiple sets of files with similar processing options.

Assume that a menu requests the operator to specify the application program (for example, accounts receivable or accounts payable) that uses the Open Database File (OPNDBF) command to open the required files. When the application is ended, the Close File (CLOF) command closes the files. The CLOF command is used to help reduce the amount of main storage needed by the job. In this example, different files are used for each application. The user normally works with one application for a considerable length of time before selecting a new application.

An example of the accounts receivable programs follows:

```
PGMC:   PGM      /* PGMC PROGRAM */
        DCLF     FILE(DISPLAY)
BEGIN:  SNDRCVF  RCDFMT(TOPMENU)
        IF       (&RESPONSE *EQ '1') CALL ACCRECV
        IF       (&RESPONSE *EQ '2') CALL ACCPAY
        .
        .
        IF       (&RESPONSE *EQ '90') SIGNOFF
        GOTO     BEGIN
        ENDPGM
```

```
ACCREC: PGM      /* ACCREC PROGRAM */
        DCLF     FILE(DISPLAY)
        OVRDBF   FILE(A) SHARE(*YES)
        OVRDBF   FILE(B) SHARE(*YES)
        OPNDBF   FILE(A) OPTION(*ALL) ....
        OPNDBF   FILE(B) OPTIONS(*INP) ...
BEGIN:  SNDRCVF  RCDFMT(ACCRMENU)
        IF       (&RESPONSE *EQ '1') CALL PGM21
        IF       (&RESPONSE *EQ '2') CALL PGM22
        .
        .
        IF       (&RESPONSE *EQ '88') DO /* Return */
                 CLOF FILE(A)
                 CLOF FILE(B)
                 RETURN
                 ENDDO
        GOTO     BEGIN
        ENDPGM
```

The program for the accounts payable menu would be similar, but with a different set of OPNDBF and CLOF commands.

For this example, files A and B were created with SHARE(*NO). Therefore, an OVRDBF command must precede the OPNDBF command. As in Example 1, the amount of main storage used by each job could be reduced by placing the OPNDBF commands in a separate program and calling it. A separate program could also be created for the CLOF commands. The OPNDBF commands could be placed in an application setup program that is called from the menu, which transfers control to the specific application program menu (any overrides specified in this setup program are kept). However, calling separate programs for these functions also uses system resources and, depending on the frequency with which the different menus are used, it can be better to include the OPNDBF and CLOF commands in each application program menu as shown in this example.

Another choice is to use the Reclaim Resources (RCLRSC) command in PGMC (the setup program) instead of using the Close File (CLOF) commands. The RCLRSC command closes any files and frees any leftover storage associated with any files and programs that were called and have since returned to the calling program. However, RCLRSC will *not* close files opened with TYPE(*PERM) specified on the Open Database File (OPNDBF) or Open Query File (OPNQRYF) commands.

The following example shows the RCLRSC command used to close files:

```
 .
 .
IF          (&RESPONSE *EQ '1') DO
            CALL ACCRECV
            RCLRSC
            ENDDO
IF          (&RESPONSE *EQ '2') DO
            CALL ACCPAY
            RCLRSC
            ENDDO
 .
 .
```

Example 3: Using a single set of files with different processing requirements.

If some programs need read-only file processing and others need some or all of the options (input/update/add/delete), one of the following methods can be used. The same methods apply if a file is to be processed with certain command parameters in some programs and not in others (for example, sometimes the commit option should be used).

A single Open Database File (OPNDBF) command could be used to specify OPTION(*ALL) and the open data path would be opened shared (if, for example, a previous OVRDBF command was used to specify SHARE(*YES)). Each program could then open a subset of the options. The program requests the type of open depending on the specifications in the program. In some cases this does not require any more considerations because a program specifying an open for input only would operate similarly as if it had not done a shared open (for example, no additional record locking occurs when a record is read).

However, some options specified on the OPNDBF command can affect how the program operates. For example, SEQONLY(*NO) is specified on the open command for a file in the program. An error would occur if the OPNDBF command used SEQONLY(*YES) and a program attempted an operation that was not valid with sequential-only processing.

The ACCPTH parameter must also be consistent with the way programs will use the access path (arrival or keyed).

If COMMIT(*YES) is specified on the Open Database File (OPNDBF) command and the Start Commitment Control (STRCMTCTL) command specifies LCKLVL(*ALL), any read operation of a record locks that record (per commitment control record locking rules). This can cause records to be locked unexpectedly and cause errors in the program.

Two OPNDBF commands could be used for the same data (for example, one with OPTION(*ALL) and the other specifying OPTION(*INP)). The second use must be a logical file pointing to the same physical file(s). This logical file can then be opened as SHARE(*YES) and multiple uses made of it during the same job.

Sequential-Only Processing

SEQONLY and NBRRCDS Parameters. If your program processes a database file sequentially for input only or output only, you might be able to improve performance using the sequential-only processing (SEQONLY) parameter on the Override with Database File (OVRDBF) or the Open Database File (OPNDBF) commands. To use SEQONLY processing, the file must be opened for input-only or output-only. The NBRRCDS parameter can be used with any combination of open options. (The Open Query File [OPNQRYF] command uses sequential-only processing whenever possible.) Depending on your high-level language specifications, the high-level language can also use sequential-only processing as the default. For example, if you open a file for input only and the only file operations specified in the high-level language program are sequential read operations, then the high-level language automatically requests sequential-only processing.

Note: File positioning operations are not considered sequential read operations; therefore, a high-level language program containing positioning operations will *not* automatically request sequential-only processing. (The SETLL operation in RPG/400 and the START operation in COBOL/400 are examples of file positioning operations.) Even though the high-level language program can not automatically request sequential-only processing, you can request it using the SEQONLY parameter on the OVRDBF command.

If you specify sequential-only processing, you can also specify the number of records to be moved as one unit between the system database main storage area and the job's internal data main storage area. If you do not specify the sequential-only number of records to be moved, the system calculates a number based on the number of records that fit into a 4096-byte buffer.

The system also provides you a way to control the number of records that are moved as a unit between auxiliary storage and main storage. If you are reading the data in the file in the same order as the data is physically stored, you can improve the performance of your job using the NBRRCDS parameter on the OVRDBF command.

Open Considerations for Sequential-Only Processing

The following considerations apply for opening files when sequential-only processing is specified. If the system determines that sequential-only processing is not allowed, a message is sent to the program to indicate that the request for sequential-only processing is not being accepted; however, the file is still opened for processing.

- If the program opened the member for output only, and if SEQONLY(*YES) was specified (number of records was not specified) and either the opened member is a logical member, a uniquely keyed physical member, or there are other access paths to the physical member, SEQONLY(*YES) is changed to SEQONLY(*NO) so the program can handle possible errors (for example, duplicate keys, conversion mapping, and select/omit errors) at the time of the output operation. If you want the system to run sequential-only processing, change the SEQONLY parameter to include both the *YES value and number of records specification.
- Sequential-only processing can be specified only for input-only or output-only (add) operations. If the program specifies update or delete operations, sequential-only processing is not allowed by the system.
- If a file is being opened for output, it must be a physical file or a logical file based on one physical file member.

- If the record is opened for input-only, the record length plus the feedback area must be less than or equal to 32 767 bytes. (The feedback area size is 11 + key length * number of records specified on the SEQONLY parameter.)
- The member cannot be opened for input – only with commitment control; otherwise, sequential-only processing is not allowed by the system.
- For output-only, the number of records specified to be moved as a unit and the force ratio are compared and automatically adjusted as necessary. If the number of records is larger than the force ratio, the number of records is reduced to equal the force ratio. If the opposite is true, the force ratio is reduced to equal the number of records.
- If the program opened the member for output only, and if SEQONLY(*YES) was specified (number of records was not specified), and duplicate or insert key feedback has been requested, SEQONLY(*YES) will be changed to SEQONLY(*NO) to provide the feedback on a record-by-record basis when the records are inserted into the file.

The following considerations apply when sequential-only processing is not specified and the file is opened using the Open Query File (OPNQRYF) command. If these conditions are satisfied, a message is sent to indicate that sequential-only processing will be performed and the query file is opened.

- If the OPNQRYF command specifies the name of one or more fields on the group field (GRPFLD) parameter, or OPNQRYF requires group processing.
- If the OPNQRYF command specifies one or more fields, or *ALL on the UNIQUEKEY parameter.
- If a view is used with the DISTINCT option on the SQL/400 SELECT statement, then SEQONLY(*YES) processing is automatically performed.

For more details about the OPNQRYF command, see "Using the Open Query File (OPNQRYF) Command" on page 9-2.

Input/Output Considerations for Sequential-Only Processing

The following considerations apply for input/output operations on files when sequential-only processing is specified.

- For input, your program receives one record at a time from the input buffer. When all records in the input buffer are processed, the system automatically reads the next set of records.
- For output, your program must move one record at a time to the output buffer. When the output buffer is full, the system automatically adds the records to the database.

  Note: If you are using a journal, the entire buffer is written to the journal at one time as if the entries had logically occurred together.

  If you use sequential-only processing for output, you might not see all the changes made to the file as they occur. For example, if sequential-only is specified for a file being used by PGMA, and PGMA is adding new records to the file and the SEQONLY parameter was specified with 5 as the number of records in the buffer, then only when the buffer is filled will the newly added records be transferred to the database. In this example, only when the fifth record was added, would the first five records be transferred to the database, and be available for processing by other jobs in the system.

  In addition, if you use sequential-only processing for output, some additions might not be made to the database if you do not handle the errors that could occur when records are moved from the buffer to the database. For example, assume the buffer holds five records, and the third record in the buffer had a key that was a duplicate of another record in the file and the file was defined as a unique-key file. In this case, when the system transfers the buffer to the database it would add the first two records and then get a duplicate key error for the third. Because of this error, the third, fourth, and fifth records in the buffer would *not* be added to the database.

- The force-end-of-data function can be used for output operations to force all records in the buffer to the database (except those records that would cause a duplicate key in a file defined as having unique keys, as described previously). The force-end-of-data function is only available in certain high-level languages.

Close Considerations for Sequential-Only Processing

When a file for which sequential-only processing is specified is closed, all records still in the output buffer are added to the database. However, if an error occurs for a record, any records following that record are not added to the database.

If multiple programs in the same job are sharing a sequential-only output file, the output buffer is not emptied until the final close occurs. Consequently, a close (other than the last close in the job) does not cause the records still in the buffer to appear in the database for this or any other job.

Run Time Summary

The following tables list parameters that control your program's use of the database file member, and indicates where these parameters can be specified. For parameters that can be specified in more than one place, the system merges the values. The Override with Database File (OVRDBF) command parameters take precedence over program parameters, and Open Database File (OPNDBF) or Open Query File (OPNQRYF) command parameters take precedence over create or change file parameters.

Note: Any override parameters other than TOFILE, MBR, LVLCHK, SEQONLY, and INHWRT are ignored by the OPNQRYF command.

Figure 8-1 (Page 1 of 2). Database Processing Options Specified on CL Commands

Description	Parameter	CRTPF or CRTLF Cmd	CHGPF or CHGLF Cmd	OPNDBF Cmd	OPNQRYF Cmd	OVRDBF Cmd
File name	FILE	X	X[1]	X	X	X
Library name		X	X[2]	X	X	X
Member name	MBR	X		X	X	X
Member processing options	OPTION			X	X	
Record format lock state	RCDFMTLCK					X
Starting file position after open	POSITION					X
Program performs only sequential processing	SEQONLY			X	X	X
Ignore keyed access path	ACCPTH			X		

Figure 8-1 (Page 2 of 2). Database Processing Options Specified on CL Commands

Description	**Parameter**	**CRTPF** **or** **CRTLF** **Cmd**	**CHGPF** **or** **CHGLF** **Cmd**	**OPNDBF** **Cmd**	**OPNQRYF** **Cmd**	**OVRDBF** **Cmd**
Time to wait for file locks	WAITFILE	X	X			X
Time to wait for record locks	WAITRCD	X	X			X
Prevent overrides	SECURE					X
Number of records to be transferred from auxiliary to main storage	NBRRCDS					X
Share open data path with other programs	SHARE	X	X			X
Format selector	FMTSLR	X[3]	X[3]			X
Force ratio	FRCRATIO	X	X			X
Inhibit write	INHWRT					X
Level check record formats	LVLCHK	X	X			X
Expiration date checking	EXPCHK					X
Expiration date	EXPDATE	X[4]	X[4]			X
Force access path	FRCACCPTH	X	X			
Commitment control	COMMIT			X	X	
End-of-file delay	EOFDLY					X
Duplicate key check	DUPKEYCHK			X	X	

[1] File name: The CHGPF and CHGLF commands use the file name for identification only. You cannot change the file name.

[2] Library name: The CHGPF and CHGLF commands use the library name for identification only. You cannot change the library name.

[3] Format selector: Used on the CRTLF and CHGLF commands only.

[4] Expiration date: Used on the CRTPF and CHGPF commands only.

Figure 8-2. Database Processing Options Specified in Programs

Description	RPG/400	COBOL/400	AS/400 BASIC	AS/400 PL/I	AS/400 Pascal
File name	X	X	X	X	X
Library name			X	X	X
Member name			X	X	X
Program record length	X	X	X	X	X
Member processing options	X	X	X	X	X
Record format lock state			X	X	
Record formats the program will use	X		X		
Clear physical file member of records		X	X		X
Program performs only sequential processing	X	X		X	X
Ignore keyed access path	X	X	X	X	X
Share open data path with other programs				X	X
Level check record formats	X	X	X	X	
Commitment control	X	X		X	
Duplicate key check		X			

Note: Control language (CL) programs can also specify many of these parameters. See Figure 8-1 on page 8-16 for more information about the database processing options that can be specified on CL commands.

Chapter 9. Opening a Database File

This chapter discusses opening a database file. In addition, the CL commands Open Database File (OPNDBF) and Open Query File (OPNQRYF) are discussed.

Opening a Database File Member

To use a database file in a program, your program must issue an open operation to the database file. If you do not specify an open operation in some programming languages, they automatically open the file for you. If you did not specify a member name in your program or on an Override with Database File (OVRDBF) command, the first member (as defined by creation date and time) in the file is used. After finding the member, the system connects your program to the database file. This allows your program to perform input/output operations to the file. For more information about opening files in your high-level language program, see the appropriate high-level language guide.

You can open a database file with statements in your high-level language program. You can also use the CL open commands: Open Database File (OPNDBF) and Open Query File (OPNQRYF). The OPNDBF command is useful in an initial program in a job for opening shared files. The OPNQRYF command is very effective in selecting and arranging records outside of your program. Then, your program can use the information supplied by the OPNQRYF command to process only the data it needs.

Using the Open Database File (OPNDBF) Command

Usually, when you use the OPNDBF command, you can use the defaults for the command parameter values. In some instances you may want to specify particular values, instead of using the default values, for the following parameters:

OPTION Parameter. Specify the *INP option if your application programs uses input-only processing (reading records without updating records). This allows the system to read records without trying to lock each one for possible update. Specify the *OUT option if your application programs uses output-only processing (writing records into a file but not reading or updating existing records).

Note: If your program does direct output operations to active records (updating by relative record number), *ALL must be specified instead of *OUT. If your program does direct output operations to deleted records only, *OUT must be specified.

MBR Parameter. If a member, other than the first member in the file, is to be opened, you must specify the name of the member to be opened or issue an Override with Database File (OVRDBF) command before the Open Database File (OPNDBF) command.

Note: You must specify a member name on the OVRDBF command to use a member (other than the first member) to open in subsequent programs.

OPNID Parameter. If an identifier other than the file name is to be used, you must specify it. The open identifier can be used in other CL commands to process the file. For example, the Close File (CLOF) command uses the identifier to specify which file is to be closed.

ACCPTH Parameter. If the file has a keyed sequence access path and either (1) the open option is *OUT, or (2) the open option is *INP or *ALL, but your program does not use the keyed access path, then you can specify ACCPTH(*ARRIVAL) on the OPNDBF parameter. Ignoring the keyed access path can improve your job's performance.

SEQONLY Parameter. Specify *YES if subsequent application programs process the file sequentially. This parameter can also be used to specify the number of records that should be transferred between the system data buffers and the program data buffers. SEQONLY(*YES) is not allowed unless OPTION(*INP) or OPTION(*OUT) is also specified on the Open Database File (OPNDBF) command.

COMMIT Parameter. Specify *YES if the application programs use commitment control. If you specify *YES you must be running in a commitment control environment (the Start Commitment Control [STRCMTCTL] command was processed) or the OPNDBF command will fail. Use the default of *NO if the application programs do not use commitment control.

TYPE Parameter. Specify what you wish to happen when unmonitored exceptions occur in your application program. If you specify *NORMAL, your program can issue a Reclaim Resources (RCLRSC) command to close the files, or the high-level language you are using can perform a close operation. Specify *PERM if you want to continue the application without opening the files again. Any error message received in your program when files are open causes those files to be closed if TYPE(*NORMAL) was specified. TYPE(*PERM) allows the files to remain open even if an error message is received.

DUPKEYCHK Parameter. Specify whether or not you want duplicate key feedback. If you specify *YES, duplicate key feedback is returned on I/O operations. If you specify *NO, duplicate key feedback is not returned on I/O operations. Use the default (*NO) if the application programs are not written in COBOL/400 or if your COBOL programs do not use the duplicate key feedback information returned.

Using the Open Query File (OPNQRYF) Command

The Open Query File (OPNQRYF) command is a CL command that allows you to perform many data processing functions on database files. Essentially, the OPNQRYF command acts as a filter between the processing program and the database records. The database file can be a physical or logical file.

Unlike a database file created with the Create Physical File (CRTPF) command or the Create Logical File (CRTLF) command, the OPNQRYF command creates only a temporary file for processing the data, it does not create a permanent file.

To understand the OPNQRYF command support, you should already be familiar with database concepts such as physical and logical files, key fields, record formats, and join logical files. These concepts are discussed in Part 1 and Part 2.

The OPNQRYF command has functions similar to those in DDS, and the CRTPF and CRTLF commands. DDS requires source statements and a separate step to create the file. OPNQRYF allows a dynamic definition without using DDS. The OPNQRYF command does not support all of the DDS functions, but it supports significant functions that go beyond the capabilities of DDS.

In addition, Query can be used to perform some of the function the OPNQRYF command performs. However, the OPNQRYF command is more useful as a programmer's tool.

The OPNQRYF command parameters also have many functions similar to the SQL/400 SELECT functions. For example, the FILE parameter is similar to the SQL/400 FROM statement, the QRYSLT parameter is similar to the SQL/400 WHERE statement, the GRPFLD parameter is similar to the SQL/400 GROUP BY statement, and the GRPSLT parameter is similar to the SQL/400 HAVING statement. For more information about SQL/400, see the *SQL/400 Programmer's Guide*.

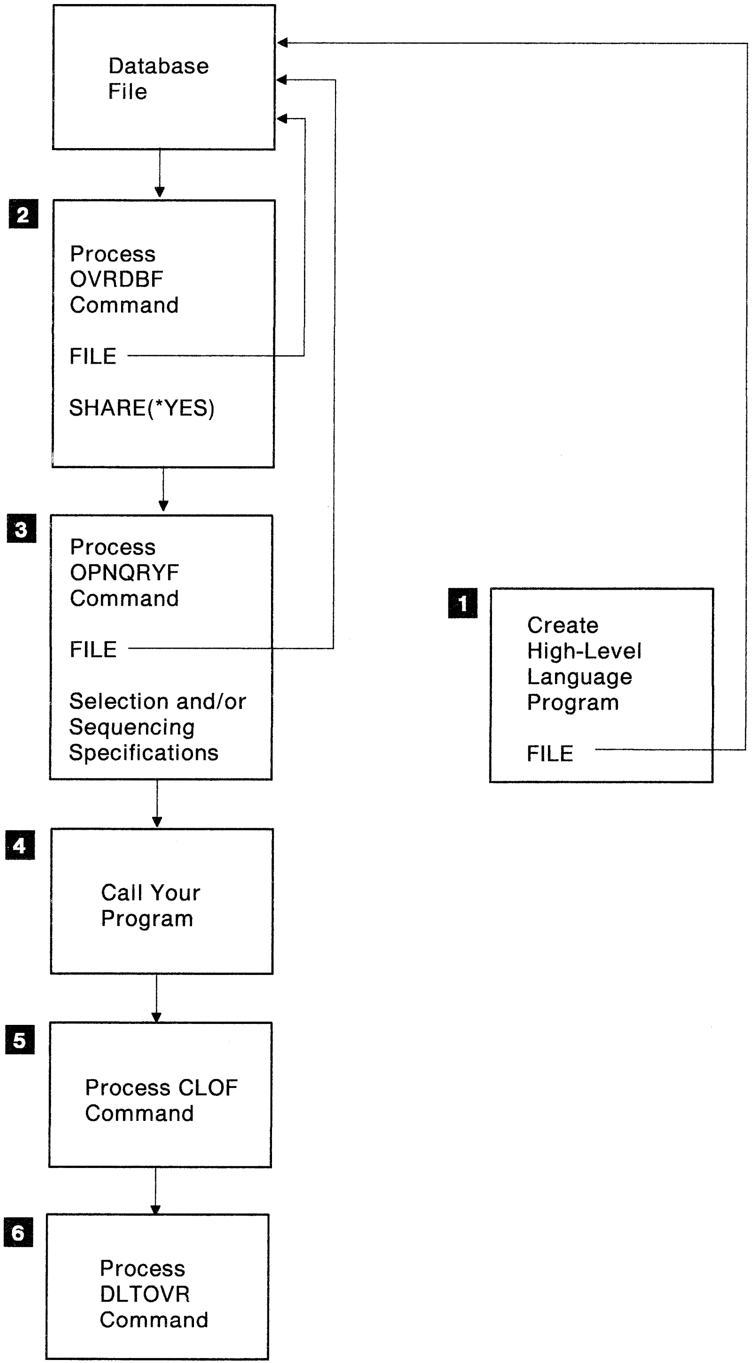
The following is a list of the major functions supplied by OPNQRYF. Each of these functions is described later in this section.

- Dynamic record selection
- Dynamic keyed access path
- Dynamic keyed access path over a join
- Dynamic join
- Handling missing records in secondary join files
- Unique-key processing
- Mapped field definitions
- Group processing
- Final total-only processing
- Improving performance

To understand the OPNQRYF command, you must be familiar with its two processing approaches: using a format in the file, and using a file with a different format. The typical use of the OPNQRYF command is to select, arrange, and format the data so it can be read sequentially by your high-level language program.

Using an Existing Record Format in the File

Assume you only want your program to process the records in which the *Code* field is equal to D. You create the program as if there were only records with a D in the *Code* field. That is, you do not code any selection operations in the program. You then run the OPNQRYF command, and specify that only the records with a D in the *Code* field are to be returned to the program. The OPNQRYF command does the record selection and your program processes only the records that meet the selection values. You can use this approach to select a set of records, return records in a different sequence than they are stored, or both. The following is an example of using the OPNQRYF command to select and sequence records:



RSLH298-3

1 Create the high-level language program to process the database file as you would any normal program using externally described data. Only one format can be used, and it must exist in the file.

2 Run the OVRDBF command specifying the file and member to be processed and SHARE(*YES). (If the member is permanently changed to SHARE(*YES) and the first or only member is the one you want to use, this step is not necessary.)

The OVRDBF command can be run after the OPNQRYF command, unless you want to override the file name specified in the OPNQRYF command. In this discussion and in the examples, the OVRDBF command is shown first.

Some restrictions are placed on using the OVRDBF command with the OPNQRYF command. For example, MBR(*ALL) causes an error message and the file is not opened. Refer to "Considerations for Files Shared in a Job" on page 9-37 for more information.

3 Run the OPNQRYF command, specifying the database file, member, and format names and any selection and/or sequencing options.

4 Call the high-level language program you created in step 1. Only a high-level language program can use the file created by the OPNQRYF command. CL commands (for example, the Copy File [CPYF] and the Display Physical File Member [DSPPFM] commands) and utilities (for example, Query) do not work on files created with the OPNQRYF command.

5 Close the file that you opened in step 3, unless you want the file to remain open. The Close File (CLOF) command can be used to close the file.

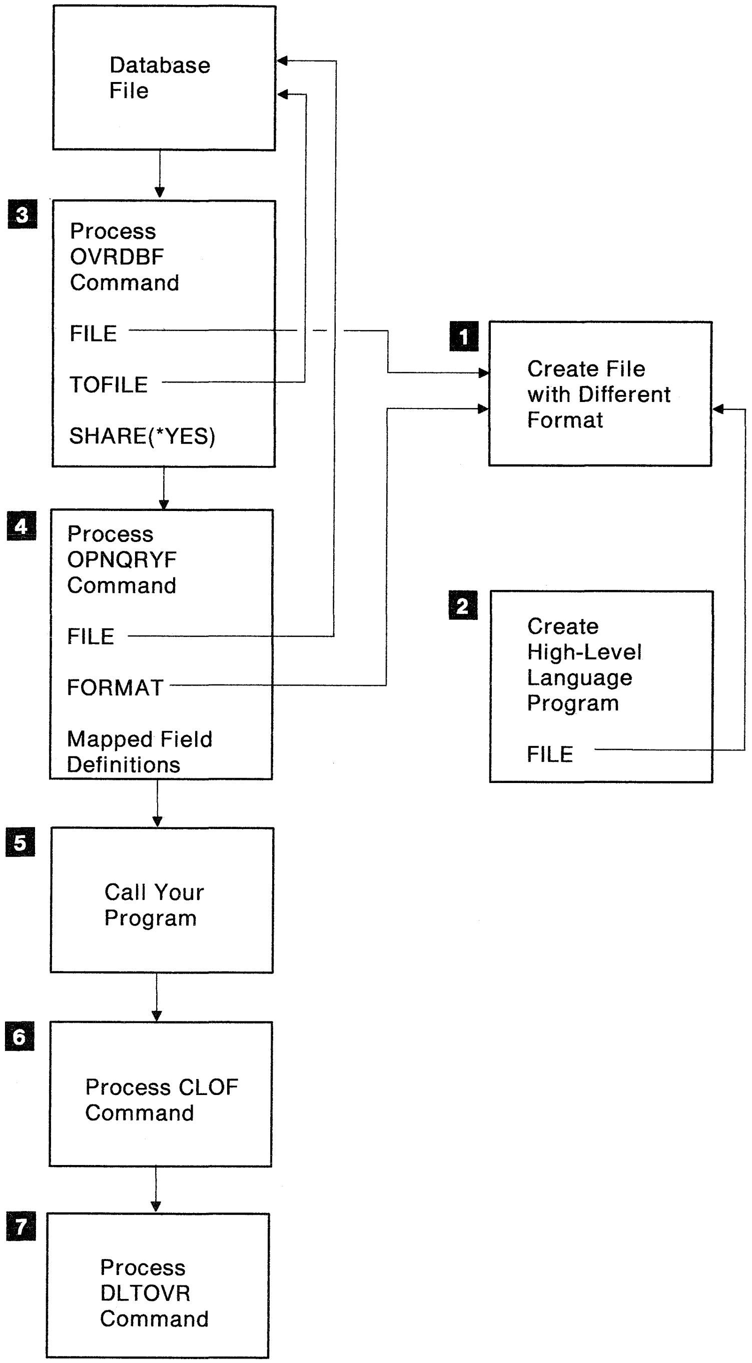
6 Delete the override specified in step 2 with the the Delete Override (DLTOVR) command. It may not always be necessary to delete the override, but the command is shown in all the examples for consistency.

Using a File with a Different Record Format

For more advanced functions of the Open Query File (OPNQRYF) command (such as dynamically joining records from different files), you must define a new file that contains a different record format. This new file is a separate file from the one you are going to process. This new file contains the fields that you want to create with the OPNQRYF command. This powerful capability also lets you define fields that do not currently exist in your database records, but can be derived from them.

When you code your high-level language program, specify the name of the file with the different format so the externally described field definitions of both existing and derived fields can be processed by the program.

Before calling your high-level language program, you must specify an Override with Database File (OVRDBF) command to direct your program file name to the open query file. On the OPNQRYF command, specify both the database file and the new file with the special format to be used by your high-level language program. If the file you are querying does not have SHARE(*YES) specified, you must specify SHARE(*YES) on the OVRDBF command.



RSLH299-4

1 Specify the DDS for the file with the different record format, and create the file. This file contains the fields that you want to process with your high-level language program. Normally, data is not contained in this file, and it does not require a member. You normally create this file as a physical file without keys. A field reference file can be used to describe the fields. The record format name can be different from the record format name in the database file that is specified. You can use any database or DDM file for this function. The file could be a logical file and it could be indexed. It could have one or more members, with or without data.

2 Create the high-level language program to process the file with the record format that you created in step 1. In this program, do not name the database file that contains the data.

3 Run the Override with Database File (OVRDBF) command. Specify the name of the file with the different (new) record format on the FILE parameter. Specify the name of the database file that you want to query on the TOFILE parameter. You can also specify a member name on the MBR parameter. If the database member you are querying does not have SHARE(*YES) specified, you must also specify SHARE(*YES) on the OVRDBF command.

4 Run the Open Query File (OPNQRYF) command. Specify the database file to be queried on the FILE parameter, and specify the name of the file with the different (new) format that was created in step 1 on the FORMAT parameter. Mapped field definitions can be required on the OPNQRYF command to describe how to map the data from the database file into the format that was created in step 1. You can also specify selection and sequencing options.

5 Call the high-level language program you created in step 2.

6 The first file named in step 4 for the FILE parameter was opened with OPNQRYF as SHARE(*YES) and is still open. The file must be closed. The Close File (CLOF) command can be used.

7 Delete the override that was specified in step 3.

The previous steps show the normal flow using externally described data. It is not necessary to create unique DDS and record formats for each OPNQRYF command. You can reuse an existing record format. However, all fields in the record format must be actual fields in the real database file or defined by mapped field definitions. If you use program-described data, you can create the program at any time.

You can use the file created in step 1 to hold the data created by the Open Query File (OPNQRYF) command. For example, you can replace step 5 with a high-level language processing program that copies data to the file with the different format (the CPYF command cannot be used). You can then follow step 5 with the CPYF command or Query.

Examples

The following sections describe how to specify both the OPNQRYF parameters for each of the major functions discussed earlier and how to use the Open Query File command with your high-level language program.

Notes:

1. If you are running the OPNQRYF command from a command entry line, any error messages occurring after the OPNQRYF command is successfully run will close the file unless TYPE(*PERM) was specified on the OPNQRYF command. The system automatically runs a Reclaim Resources (RCLRSC) command if an error message occurs, except message CPF0001 (Command analyzer found error in command) sent when the system detects and error in the command. You must run the OPNQRYF command again after the error is removed.

   Note: If the OPNQRYF command specified TYPE(*PERM), the file is not automatically closed by the system.

2. After running a program that uses the Open Query File command for sequential processing, the file position is normally at the end of the file. If you want to run the same program or a different program with the same files, you must position the file or close the file and open it with the same OPNQRYF command. You can

position the file with the Position Database File (POSDBF) command. In some cases, a high-level language program statement can be used.

Selecting Records without Using DDS

Dynamic record selection allows you to request a subset of the records in a file without using DDS. For example, you can select records that have a specific value or range of values (for example, all customer numbers between 1000 and 1050). The Open Query File (OPNQRYF) command allows you to combine these and other selection functions to produce powerful record selection capabilities.

Examples of Selecting Records Using the Open Query File (OPNQRYF) Command

In all of the following examples, it is assumed that a single-format database file (physical or logical) is being processed. (The FILE parameter on the OPNQRYF command allows you to specify a record format name if the file is a multiple format logical file.)

Example 1: Selecting records with a specific value

Assume you want to select all the records from FILEA where the value of the *Code* field is D. Your processing program is PGMB. PGMB only sees the records that meet the selection value (you do not have to test in your program).

Note: You can specify parameters easier by using the prompt function for the OPNQRYF command. For example, you can specify an expression for the QRYSLT parameter without the surrounding delimiters because the system will add the apostrophes.

Specify the following:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) QRYSLT('CODE *EQ "D" ')
CALL       PGM(PGMB)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

Notes:

1. The entire expression in the QRYSLT parameter must be enclosed in apostrophes.
2. When specifying field names in the OPNQRYF command, the names in the record are not enclosed in apostrophes.
3. Character literals must be enclosed by quotation marks or two apostrophes. (The quotation mark character is used in the examples.) It is important to place the character(s) between the quotation marks in either uppercase or lowercase to match the value you want to find in the database. (The examples are all shown in uppercase.)
4. To request a selection against a numeric constant, specify:

   ```
   OPNQRYF    FILE(FILEA) QRYSLT('AMT *GT 1000.00')
   ```

   Notice that numeric constants are *not* enclosed by two apostrophes (quotation marks).

When comparing two fields or constants, the data types must be compatible. The following table describes the valid comparisons.

Figure 9-1. Valid Data Type Comparisons for the OPNQRYF Command

	Any Numeric	**Character**
Any Numeric	Valid	Not Valid
Character	Not Valid	Valid

Note: For DBCS information, see Appendix B.

The performance of record selection can be greatly enhanced if some file on the system uses the field being selected in a keyed access path. This allows the system to quickly access only the records that meet the selection values. If no such access path exists, the system must read every record to determine if it meets the selection values.

Even if an access path exists on the field you want to select from, the system may not use the access path. For example, if it is faster for the system to process the data in arrival sequence, it will do so. See the discussion in "Performance Considerations" on page 9-34 for more details.

Example 2: Selecting records with a specific date value

Assume you want to process all records in which the *Date* field in the record is the same as the current date. Also assume the *Date* field is in the same format as the system date. In a CL program, you can specify:

```
DCL        VAR(&CURDAT)  TYPE(*CHAR)  LEN(6)
RTVSYSVAL  SYSVAL(QDATE) RTNVAR(&CURDAT)
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) QRYSLT('"' *CAT &CURDAT *CAT '" *EQ DATE')
CALL       PGM(PGMB)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

A CL variable is assigned with a leading ampersand (&) and is not enclosed in apostrophes. The quotation mark is enclosed in apostrophes as is the string *EQ DATE. The quotation mark is concatenated to the value of the CL variable, as is the remainder of the QRYSLT string.

It is important to know whether the data in the database is defined as character or numeric. In this example, the *Date* field is assumed to be character. If the field was defined as numeric and you were using a constant, the QRYSLT statement would be specified as follows:

```
QRYSLT('123187 *EQ DATE')
```

Note that double apostrophes are not used around the constant. If you replace the constant with a variable, the QRYSLT parameter would be written as follows (see the difference from the one specified above):

```
QRYSLT(&CURDAT *CAT ' *EQ DATE')
```

If a numeric field exists in the database and you want to compare it to a variable, only a character variable can be used. For example, to select all records where a packed *Date* field is greater than a variable, you must ensure the variable is in character form. Normally, this will mean that before the Open Query File (OPNQRYF) command, you use the Change Variable (CHGVAR) command to change the variable from a decimal field to a character field. The CHGVAR command would be specified as follows:

```
CHGVAR VAR(&CHARVAR)  VALUE('123188')
```

The QRYSLT parameter would be specified as follows:

```
QRYSLT(&CHARVAR  *CAT ' *GT DATE')
```

Example 3: Selecting records in a range of values

Assume you have a *Date* field specified in the character format YYMMDD, and you want to process all records for 1988. You can specify:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) QRYSLT('DATE *EQ %RANGE("880101" +
              "881231") ')
CALL       PGM(PGMC)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

If the ranges are variables, you would specify the QRYSLT parameter as follows:

```
QRYSLT('DATE *EQ %RANGE("' *CAT &LORNG *CAT'"' *BCAT '"'  +
   *CAT &HIRNG *CAT '")')
```

Example 4: Selecting records using the contains function

Assume you want to process all records in which the *Addr* field contains the street named BROADWAY. The contains (*CT) function determines if the characters appear anywhere in the named field. You can specify:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) QRYSLT('ADDR *CT "BROADWAY" ')
CALL       PGM(PGMC)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

In this example, assume that the data is in uppercase in the database record. If the data was in lowercase or mixed case, you could specify a translation function to translate the lowercase or mixed case data to uppercase before the comparison is made. The system-provided table QSYSTRNTBL translates the letters a through z to uppercase. (You could use any translation table to perform the translation.) Therefore, you can specify:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) QRYSLT('%XLATE(ADDR QSYSTRNTBL) *CT +
                  "BROADWAY" ')
CALL       PGM(PGMC)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

When the %XLATE function is used on the QRYSLT statement, the value of the field passed to the high-level language program appears as it is in the database. You can force the field to appear in uppercase using the %XLATE function on the MAPFLD parameter.

Example 5: Selecting records using multiple fields

Assume you want to process all records in which the *Amt* field is equal to zero or the *Lstdat* field (YYMMDD order in character format) is equal to or less than 881231. You can specify:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) QRYSLT('AMT *EQ 0 *OR LSTDAT +
                  *LE "881231" ')
CALL       PGM(PGMC)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

If variables are used, the QRYSLT parameter would be typed as follows:

```
QRYSLT('AMT *EQ ' *CAT &VARAMT *CAT ' *OR  +
   LSTDAT *LE "' *CAT &VARDAT *CAT '"')
```

Note that the &VARAMT variable must be defined as a character type. If the variable is passed to your CL program as a numeric type, you must convert it to a character type to allow concatenation. You can use the Change Variable (CHGVAR) command to do this conversion.

Example 6: Using the Open Query File (OPNQRYF) command many times in a program

You can use the OPNQRYF command more than once in a high-level language program. For example, assume you want to prompt the user for some selection values, then display one or more pages of records. At the end of the first request for records, the user may want to specify other selection values and display those records. This can be done by doing the following:

1. Before calling the high-level language program, use an Override with Database File (OVRDBF) command to specify SHARE(*YES).
2. In the high-level language program, prompt the user for the selection values.
3. Pass the selection values to a CL program that issues the OPNQRYF command (or run the command with a call to program QCMDEXC). The file must be closed before your program processes the OPNQRYF command. You normally use the Close File (CLOF) command and monitor for the file not being open.
4. Return to the high-level language program.
5. Open the file in the high-level language program.
6. Process the records.
7. Close the file in the program.
8. Return to step 2.

When the program completes, run the Close File (CLOF) command or the Reclaim Resources (RCLRSC) command to close the file, then delete the Override with Database File command specified in step 1.

Note: An override command in a called CL program does not affect the open in the main program. All overrides are implicitly deleted when the program is ended. (However, you can use a call to program QCMDEXC from your high-level language program to specify an override, if needed.)

Example 7: Mapping fields for packed numeric data fields

Assume you have a packed decimal *Date* field in the format MMDDYY and you want to select all the records for the year 1988. You cannot select records directly from a portion of a packed decimal field, but you can use the MAPFLD parameter on the OPNQRYF command to create a new field that you can then use for selecting part of the field.

The format of each mapped field definition is:

(result field 'expression' attributes)

where:

result field = The name of the result field.
expression = How the result field should be derived. The expression can include substring, other built-in functions, or mathematical statements.
attributes = The optional attributes of the result field. If no attributes are given (or the field is not defined in a file), the OPNQRYF command calculates a field attribute determined by the fields in the expression.

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) QRYSLT('YEAR *EQ "88" ') +
             MAPFLD((CHAR6 '%DIGITS(DATE)') +
               (YEAR '%SST(CHAR6 5 2)' *CHAR 2))
CALL       PGM(PGMC)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

The first mapped field definition specifies that the *Char6* field be created from the packed decimal *Date* field. The %DIGITS function converts from packed decimal to character and ignores any decimal definitions (that is, 1234.56 is converted to '123456'). Because no definition of the *Char6* field is specified, the system assigns a length of 6. The second mapped field defines the *Year* field as type *CHAR (character) and length 2. The expression uses the substring function to map the last 2 characters of the *Char6* field into the *Year* field.

Note that the mapped field definitions are processed in the order in which they are specified. In this example, the *Date* field was converted to character and assigned to the *Char6* field. Then, the last two digits of the *Char6* field (the year) were assigned to the *Year* field. Any changes to this order would have produced an incorrect result.

Note: Mapped field definitions are always processed before the QRYSLT parameter is evaluated.

You could accomplish the same result by specifying the substring on the QRYSLT parameter and dropping one of the mapped field definitions as follows:

```
OPNQRYF    FILE(FILEA) +
             QRYSLT('%SST(CHAR6 5 2) *EQ "88" ') +
             MAPFLD((CHAR6 '%DIGITS(DATE)'))
```

Example 8: Using the "wild card" function

Assume you have a packed decimal *Date* field in the format MMDDYY and you want to select the records for March 1988. To do this, you can specify:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) +
             QRYSLT('%DIGITS(DATE) *EQ %WLDCRD("03__88")')
CALL       PGM(PGMC)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

Note that the only time the MAPFLD parameter is needed to define a database field for the result of the %DIGITS function is when the result needs to be used with a function that only supports a simple field name (not a function or expression) as an argument. The %WLDCRD operation has no such restriction on the operand that appears before the *EQ operator.

Note that although the field in the database is in numeric form, double apostrophes surround the literal to make its definition the same as the *Char6* field.

The %WLDCRD function lets you select any records that match your selection values, in which the underline (_) will match any single character value. The two underline characters in Example 8 allow any day in the month of March to be selected. The %WLDCRD function also allows you to name the wild card character (underline is the default).

The wild card function supports two different forms:

- A fixed-position wild card as shown in the previous example in which the underline (or your designated character) matches any single character as in the following example:

  ```
  QRYSLT('FLDA *EQ %WLDCRD("A_C")')
  ```

  This compares successfully to ABC, ACC, ADC, AxC, and so on. In this example, the field being analyzed only compares correctly if it is exactly 3 characters in length. If the field is longer than 3 characters, you also need the second form of wild card support.
- A variable-position wild card will match any zero or more characters. The Open Query File (OPNQRYF) command uses an asterisk (*) for this type of wild card variable character or you can specify your own character. An asterisk is used in the following example:

  ```
  QRYSLT('FLDB *EQ %WLDCRD("A*C*") ')
  ```

  This compares successfully to AC, ABC, AxC, ABCD, AxxxxxxxC, and so on. The asterisk causes the command to ignore any intervening characters if they exist. Notice that in this example the asterisk is specified both before and after the character or characters that can appear later in the field. If the asterisk were omitted from the end of the search argument, it causes a selection only if the field ends with the character C.

  You must specify an asterisk at the start of the wild card string if you want to select records where the remainder of the pattern starts anywhere in the field. Similarly, the pattern string must end with an asterisk if you want to select records where the remainder of the pattern ends anywhere in the field.

For example, you can specify:

```
QRYSLT('FLDB *EQ %WLDCRD("*ABC*DEF*") ')
```

You get a match on ABCDEF, ABCxDEF, ABCxDEFx, ABCxxxxxxDEF, ABCxxxDEFxxx, xABCDEF, xABCxDEFx, and so on.

You can combine the two wild card functions as in the following example:

```
QRYSLT('FLDB *EQ %WLDCRD("ABC_*DEF*") ')
```

You get a match on ABCxDEF, ABCxxxxxxDEF, ABCxxxDEFxxx, and so on. The underline forces at least one character to appear between the ABC and DEF (for example, ABCDEF would not match).

Assume you have a *Name* field that contains:

JOHNS
JOHNS SMITH
JOHNSON
JOHNSTON

If you specify the following you will only get the first record:

```
QRYSLT('NAME *EQ "JOHNS"')
```

You would not select the other records because a comparison is made with blanks added to the value you specified. The way to select all four names is to specify:

```
QRYSLT('NAME *EQ %WLDCRD("JOHNS*")')
```

Note: For information about using the %WLDCRD function for DBCS, see Appendix B.

Example 9: Using complex selection statements

Complex selection statements can also be specified. For example, you can specify:

```
QRYSLT('DATE *EQ "880101" *AND AMT *GT 5000.00')

QRYSLT('DATE *EQ "880101" *OR AMT *GT 5000.00')
```

You can also specify:

```
QRYSLT('CODE *EQ "A" *AND TYPE *EQ "X" *OR CODE *EQ "B")
```

The rules governing the priority of processing the operators are described in the *CL Reference*. Some of the rules are:

- The *AND operations are processed first; therefore, the record would be selected if:

  The *Code* field = ″A″ and The *Type* field = ″X″
  or
  The *Code* field = ″B″

- Parentheses can be used to control how the expression is handled, as in the following example:

  ```
  QRYSLT('(CODE *EQ "A" *OR CODE *EQ "B") *AND TYPE *EQ "X" +
          *OR CODE *EQ "C"')
  ```

  The *Code* field = ″A″ and The *Type* field= ″X″
  or
  The *Code* field = ″B″ and The *Type* field = ″X″
  or
  The *Code* field = ″C″

You can also use the symbols described in the *CL Reference* instead of the abbreviated form (for example, you can use = instead of *EQ) as in the following example:

```
QRYSLT('CODE = "A" & TYPE = "X" | AMT > 5000.00')
```

This command selects all records in which:

The *Code* field = ″A″ and The *Type* field = ″X″
or
The *Amt* field > 5000.00

A complex selection statement can also be written, as in the following example:

```
QRYSLT('CUSNBR = %RANGE("60000" "69999") & TYPE = "B" +
      & SALES>0 & ACCRCV / SALES>.3')
```

This command selects all records in which:

The *Cusnbr* field is in the range 60000-69999 and
The *Type* field = ″B″ and
The *Sales* fields are greater than 0 and
Accrcv divided by *Sales* is greater than 30 percent

Specifying a Keyed Access Path without Using DDS

The dynamic access path function allows you to specify a keyed access path for the data to be processed. If an access path already exists that can be shared, the system can share it. If a new access path is required, it is built before any records are passed to the program.

Example 1: Arranging records using one key field

Assume you want to process the records in FILEA arranged by the value in the *Cust* field with program PGMD. You can specify:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) KEYFLD(CUST)
CALL       PGM(PGMD)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

Note: The FORMAT parameter on the Open Query File (OPNQRYF) command is not needed because PGMD is created by specifying FILEA as the processed file. FILEA can be an arrival sequence or a keyed sequence file. If FILEA is keyed, its key field can be the *Cust* field or a totally different field.

Example 2: Arranging records using multiple key fields

If you want the records to be processed by *Cust* sequence and then by *Date* in *Cust*, specify:

```
OPNQRYF    FILE(FILEA) KEYFLD(CUST DATE)
```

If you want the *Date* to appear in descending sequence, specify:

```
OPNQRYF    FILE(FILEA) KEYFLD((CUST) (DATE *DESCEND))
```

In these two examples, the FORMAT parameter is not used. (If a different format is defined, all key fields must exist in the format.)

Specifying Key Fields from Different Files

A dynamic keyed access path over a join logical file allows you to specify a processing sequence in which the keys can be in different physical files (DDS restricts the keys to the primary file).

The specification is identical to the previous method. The access path is specified using whatever key fields are required. There is no restriction on which physical file the key fields are in. However, if a key field exists in other than the primary file of a join specification, the system must make a temporary copy of the joined records. The system must also build a keyed access path over the copied records before the query file is opened. The key fields must exist in the format identified on the FORMAT parameter.

Example 1: Using a field in a secondary file as a key field

Assume you already have a join logical file named JOINLF. FILEX is specified as the primary file and is joined to FILEY. You want to process the records in JOINLF by the *Descrp* field which is in FILEY.

Assume the file record formats contain the following fields:

FILEX	FILEY	JOINLF
Item	Item	Item
Qty	Descrp	Qty
		Descrp

You can specify:

```
OVRDBF    FILE(JOINLF) SHARE(*YES)
OPNQRYF   FILE(JOINLF) KEYFLD(DESCRP)
CALL      PGM(PGMC)
CLOF      OPNID(JOINLF)
DLTOVR    FILE(JOINLF)
```

If you want to arrange the records by *Qty* in *Descrp* (*Descrp* is the primary key field and *Qty* is a secondary key field) you can specify:

```
OPNQRYF   FILE(JOINLF) KEYFLD(DESCRP QTY)
```

Dynamically Joining Database Files without DDS

The dynamic join function allows you to join files without having to first specify DDS and create a join logical file. You must use the FORMAT parameter on the Open Query File (OPNQRYF) command to specify the record format for the join. You can join any physical or logical file including a join logical file and a view (DDS does not allow you to join logical files). You can specify either a keyed or arrival sequence access path. If keys are specified, they can be from any of the files included in the join (DDS restricts keys to just the primary file).

In the following examples, it is assumed that the file specified on the FORMAT parameter was created. You will normally want to create the file before you create the processing program so you can use the externally described data definitions.

The default for the join order (JORDER) keyword is used in all of the following examples. The default for the JORDER keyword is *ANY, which tells the system that it can determine the order in which to join the files. That is, the system determines which file to use as the primary file and which as the secondary files. This allows the system to try to improve the performance of the join function.

Example 1: Dynamically joining files

Assume you want to join FILEA and FILEB. Assume the files contain the following fields:

FILEA	FILEB	JOINAB
Cust	Cust	Cust
Name	Amt	Name
Addr		Amt

The join field is *Cust* which exists in both files. Any record format name can be specified in the Open Query File (OPNQRYF) command for the join file. The file does not need a member. The records are not required to be in keyed sequence.

You can specify:

```
OVRDBF    FILE(JOINAB) TOFILE(FILEA) SHARE(*YES)
OPNQRYF   FILE(FILEA FILEB) FORMAT(JOINAB) +
            JFLD((FILEA/CUST FILEB/CUST)) +
            MAPFLD((CUST 'FILEA/CUST'))
CALL      PGM(PGME) /* Created using file JOINAB as input */
CLOF      OPNID(FILEA)
DLTOVR    FILE(JOINAB)
```

File JOINAB is a physical file with no data. This is the file that contains the record format to be specified on the FORMAT keyword in the Open Query File (OPNQRYF) command.

Notice that the TOFILE parameter on the Override with Database File (OVRDBF) command specifies the name of the primary file for the join operation (the first file specified for the FILE parameter on the OPNQRYF command). In this example, the FILE parameter on the Open Query File (OPNQRYF) command identifies the files in the sequence they are to be joined (A to B). The format for the file is in the file JOINAB.

The JFLD parameter identifies the *Cust* field in FILEA to join to the *Cust* field in FILEB. Because the *Cust* field is not unique across all of the joined record formats, it must be qualified on the JFLD parameter. The system attempts to determine, in some cases, the most efficient values even if you do not specify the JFLD parameter on the Open Query File (OPNQRYF) command. For example, using the previous example, if you specified:

```
OPNQRYF    FILE(FILEA FILEB) FORMAT(JOINAB) +
              QRYSLT('FILEA/CUST *EQ FILEB/CUST') +
              MAPFLD((CUST 'FILEA/CUST'))
```

The system joins FILEA and FILEB using the *Cust* field because of the values specified for the QRYSLT parameter. Notice that in this example the JFLD parameter is not specified on the command. However, if either JDFTVAL(*ONLYDFT) or JDFTVAL(*YES) is specified on the OPNQRYF command, the JFLD parameter must be specified.

The MAPFLD parameter is needed on the Open Query File (OPNQRYF) command to describe which file should be used for the data for the *Cust* field in the record format for file JOINAB. If a field is defined on the MAPFLD parameter, its unqualified name (the *Cust* field in this case without the file name identification) can be used anywhere else in the OPNQRYF command. Because the *Cust* field is defined on the MAPFLD parameter, the first value of the JFLD parameter need not be qualified. For example, the same result could be achieved by specifying:

```
JFLD((CUST FILEB/CUST)) +
MAPFLD((CUST 'FILEA/CUST'))
```

Any other uses of the same field name in the Open Query File (OPNQRYF) command to indicate a field from a file other than the file defined by the MAPFLD parameter must be qualified with a file name.

Because no KEYFLD parameter is specified, the records appear in any sequence depending on how the Open Query File (OPNQRYF) command selects the records. You can force the system to arrange the records the same as the primary file. To do this, specify *FILE on the KEYFLD parameter. You can specify this even if the primary file is in arrival sequence.

The JDFTVAL parameter (similar to the JDFTVAL keyword in DDS) can also be specified on the Open Query File (OPNQRYF) command to describe what the system should do if one of the records is missing from the secondary file. In this example, the JDFTVAL parameter was not specified, so only the records that exist in both files are selected.

If you tell the system to improve the results of the query (through parameters on the OPNQRYF command), it will generally try to use the file with the smallest number of records as the primary file. However, the system will also try to avoid building a temporary file. Depending on the selection and sequence values you specify, the file with the largest number of records could be the primary file.

You can force the system to follow the sequence of the join as you have specified it in the JFILE parameter on the Open Query File (OPNQRYF) command by specifying JORDER(*FILE). If JDFTVAL(*YES) or JDFTVAL(*ONLYDFT) is specified, the system will never change the join file sequence because a different sequence could cause different results.

Example 2: Reading only those records with secondary file records

Assume you want to join files FILEAB, FILECD, and FILEEF to select only those records with matching records in secondary files. Define a file JOINF and describe the format that should be used. Assume the record formats for the files contain the following fields:

FILEAB	FILECD	FILEEF	JOINF
Abitm	Cditm	Efitm	Abitm
Abord	Cddscp	Efcolr	Abord
Abdat	Cdcolr	Efqty	Cddscp
			Cdcolr
			Efqty

In this case, all field names in the files that make up the join file begin with a 2-character prefix (identical for all fields in the file) and end with a suffix that is identical across all the files (for example, *xxitm*). This makes all field names unique and avoids having to qualify them.

The *xxitm* field allows the join from FILEAB to FILECD. The two fields *xxitm* and *xxcolr* allow the join from FILECD to FILEEF. A keyed access path does not have to exist for these files. However, if a keyed access path does exist, performance may improve significantly because the system will attempt to use the existing access path to arrange and select records, where it can. If access paths do not exist, the system automatically creates and maintains them as long as the file is open.

```
OVRDBF     FILE(JOINF) TOFILE(FILEAB) SHARE(*YES)
OPNQRYF    FILE(FILEAB FILECD FILEEF) +
              FORMAT(JOINF) +
              JFLD((ABITM CDITM)(CDITM EFITM) +
              (CDCOLR EFCOLR))
CALL       PGM(PGME) /* Created using file JOINF as input */
CLOF       OPNID(FILEAB)
DLTOVR     FILE(JOINF)
```

The join field pairs do not have to be specified in the order shown above. For example, the same result is achieved with a JFLD parameter value of:

```
JFLD((CDCOLR EFCOLR)(ABITM CDITM) (CDITM EFITM))
```

The attributes of each pair of join fields do not have to be identical. Normal padding of character fields and decimal alignment for numeric fields occurs automatically.

The JDFTVAL keyword is not specified so *NO is assumed and no default values are used to construct join records. If you specified JDFTVAL(*YES) and there is no record in file FILECD that has the same join field value as a record in file FILEAB, defaults are used for the *Cddscp* and *Cdcolr* fields to join to file FILEEF. Using these defaults, a matching record can be found in file FILEEF (depending on if the default value matches a record in the secondary file). If not, a default value appears for these files and for the *Efqty* field.

Example 3: Using mapped fields as join fields

You can use fields defined on the MAPFLD parameter for either one of the join field pairs. This is useful when the key in the secondary file is defined as a single field (for example, a 6-character date field) and there are separate fields for the same information (for example, month, day, and year) in the primary file. Assume FILEA has character fields *Year, Month*, and *Day* and needs to be joined to FILEB which has the *Date* field in YYMMDD format. Assume you have defined file JOINAB with the desired format. You can specify:

```
OVRDBF     FILE(JOINAB) TOFILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA FILEB) FORMAT(JOINAB) +
              JFLD((YYMMDD FILEB/DATE)) +
              MAPFLD((YYMMDD 'YEAR *CAT MONTH *CAT DAY'))
CALL       PGM(PGME) /* Created using file JOINAB as input */
CLOF       OPNID(FILEA)
DLTOVR     FILE(JOINAB)
```

The MAPFLD parameter defines the *YYMMDD* field as the concatenation of several fields from FILEA. You do not need to specify field attributes (for example, length or type) for the *YYMMDD* field on the MAPFLD parameter because the system calculates the attributes from the expression.

Handling Missing Records in Secondary Join Files

The system allows you to control whether to allow defaults for missing records in secondary files (similar to the JDFTVAL DDS keyword for a join logical file). You can also specify that only records with defaults be processed. This allows you to select only those records in which there is a missing record in the secondary file.

Example 1: Reading records from the primary file that do not have a record in the secondary file

In Example 1 under “Dynamically Joining Database Files without DDS” on page 9-17, the JDFTVAL parameter is not specified, so the only records read are the result of a successful join from FILEA to FILEB. If you want a list of the records in FILEA that do not have a match in FILEB, you can specify *ONLYDFT on the JDFTVAL parameter as shown in the following example:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA FILEB) FORMAT(FILEA) +
              JFLD((CUST FILEB/CUST)) +
              MAPFLD((CUST 'FILEA/CUST')) +
              JDFTVAL(*ONLYDFT)
CALL       PGM(PGME) /* Created using file FILEA as input */
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

JDFTVAL(*ONLYDFT) causes a record to be returned to the program only when there is no equivalent record in the secondary file (FILEB).

Because any values returned by the join operation for the fields in FILEB are defaults, it is normal to use only the format for FILEA. The records that appear are those that do not have a match in FILEB. The FORMAT parameter is required whenever the FILE parameter describes more than a single file, but the file name specified can be one of the files specified on the FILE parameter. The program is created using FILEA.

Conversely, you can also get a list of all the records where there is a record in FILEB that does not have a match in FILEA. You can do this by making the secondary file the primary file in all the specifications. You would specify:

```
OVRDBF     FILE(FILEB) SHARE(*YES)
OPNQRYF    FILE(FILEB FILEA) FORMAT(FILEB) JFLD((CUST FILEA/CUST)) +
             MAPFLD((CUST 'FILEB/CUST')) JDFTVAL(*ONLYDFT)
CALL       PGM(PGMF) /* Created using file FILEB as input */
CLOF       OPNID(FILEB)
DLTOVR     FILE(FILEB)
```

Note: The Override with Database File (OVRDBF) command in this example uses FILE(FILEB) because it must specify the first file on the OPNQRYF FILE parameter. The Close File (CLOF) command also names FILEB. The JFLD and MAPFLD parameters also changed. The program is created using FILEB.

Unique-Key Processing

Unique-key processing allows you to process only the first record of a group. The group is defined by one or more records with the same set of key values. Processing the first record implies that the records you receive will have unique keys.

When you use unique-key processing, you can only read the file sequentially.

Example 1: Reading only unique-key records

Assume you want to process FILEA, which has records with duplicate keys for the *Cust* field. You want only the first record for each unique value of the *Cust* field to be processed by program PGMF. You can specify:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) KEYFLD(CUST) UNIQUEKEY(*ALL)
CALL       PGM(PGMF)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

Example 2: Reading records using only some of the key fields

Assume you want to process the same file with the sequence: *Slsman, Cust, Date*, but you want only one record per *Slsman* and *Cust.* Assume the records in the file are:

Slsman	**Cust**	**Date**	**Record #**
01	5000	880109	1
01	5000	880115	2
01	4025	880103	3
01	4025	880101	4
02	3000	880101	5

You specify the number of key fields that are unique, starting with the first key field.

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) KEYFLD(SLSMAN CUST DATE) UNIQUEKEY(2)
CALL       PGM(PGMD)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

The following records are retrieved by the program:

Slsman	Cust	Date	Record #
01	4025	880101	4
01	5000	880109	1
02	3000	880101	5

Defining Fields Derived from Existing Field Definitions

Mapped field definitions:

- Allow you to create internal fields that specify selection values (see Example 7 under “Selecting Records without Using DDS” on page 9-8 for more information).
- Allow you to avoid confusion when the same field name occurs in multiple files (see Example 1 under “Dynamically Joining Database Files without DDS” on page 9-17 for more information).
- Allow you to create fields that exist only in the format to be processed, but not in the database itself. This allows you to perform translate, substring, concatenation, and complex mathematical operations. The following examples describe this function.

Example 1: Using derived fields

Assume you have the *Price* and *Qty* fields in the record format. You can multiply one field by the other by using the Open Query File (OPNQRYF) command to create the derived *Exten* field. You want FILEA to be processed, and you have already created FILEAA. Assume the record formats for the files contain the following fields:

FILEA	FILEAA
Order	Order
Item	Item
Qty	Exten
Price	Brfdsc
Descrp	

The *Exten* field is a mapped field. Its value is determined by multiplying *Qty* times *Price*. It is not necessary to have either the *Qty* or *Price* field in the new format, but they can exist in that format, too if you wish. The *Brfdsc* field is a brief description of the *Descrp* field (it uses the first 10 characters).

Assume you have specified PGMF to process the new format. To create this program, use FILEAA as the file to read. You can specify:

```
OVRDBF     FILE(FILEAA) TOFILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) FORMAT(FILEAA) +
              MAPFLD((EXTEN 'PRICE * QTY') +
              (BRFDSC 'DESCRP'))
CALL       PGM(PGMF) /* Created using file FILEAA as input */
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEAA)
```

Notice that the attributes of the *Exten* field are those defined in the record format for FILEAA. If the value calculated for the field is too large, an exception is sent to the program.

It is not necessary to use the substring function to map to the *Brfdsc* field if you only want the characters from the beginning of the field. The length of the *Brfdsc* field is defined in the FILEAA record format.

All fields in the format specified on the FORMAT parameter must be described on the OPNQRYF command. That is, all fields in the output format must either exist in one of the record formats for the files specified on the FILE parameter or be defined on the MAPFLD parameter. If you have fields in the format on the FORMAT parameter that your program does not use, you can use the MAPFLD parameter to place zeros or blanks in the fields. Assume the *Fldc* field is a character field and the *Fldn* field is a numeric field in the output format, and you are using neither value in your program. You can avoid an error on the OPNQRYF command by specifying:

```
MAPFLD((FLDC ' " " ')(FLDN 0))
```

Notice quotation marks enclose a blank value. By using a constant for the definition of an unused field, you avoid having to create a unique format for each use of the OPNQRYF command.

Example 2: Using built-in functions

Assume you want to calculate a mathematical function that is the sine of the *Fldm* field in FILEA. First create a file (assume it is called FILEAA) with a record format containing the following fields:

FILEA	FILEAA
Code	Code
Fldm	Fldm
	Sinm

You can then create a program (assume PGMF) using FILEAA as input and specify:

```
OVRDBF     FILE(FILEAA) TOFILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) FORMAT(FILEAA) +
               MAPFLD((SINM '%SIN(FLDM)'))
CALL       PGM(PGMF) /* Created using file FILEAA as input */
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEAA)
```

The built-in function %SIN calculates the sine of the field specified as its argument. Because the *Sinm* field is defined in the format specified on the FORMAT parameter, the OPNQRYF command converts its internal definition of the sine value (in floating point) to the definition of the *Sinm* field. This technique can be used to avoid certain high-level language restrictions regarding the use of floating-point fields. For example, if you defined the *Sinm* field as a packed decimal field, PGMF could be written using any high-level language, even though the value was built using a floating-point field.

There are many other functions besides sine that can be used. Refer to the OPNQRYF command in the *CL Reference* for a complete list of built-in functions.

Example 3: Using derived fields and built-in functions

Assume, in the previous example, that a field called *Fldx* also exists in FILEA, and the *Fldx* field has appropriate attributes used to hold the sine of the *Fldm* field. Also assume that you are not using the contents of the *Fldx* field. You can use the MAPFLD parameter to change the contents of a field before passing it to your high-level language program. For example, you can specify:

```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) MAPFLD((FLDX '%SIN(FLDM)'))
CALL       PGM(PGMF) /* Created using file FILEA as input */
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

In this case, you do not need to specify a different record format on the FORMAT parameter. (The default uses the format of the first file on the FILE parameter.) Therefore, the program is created by using FILEA. When using this technique, you must ensure that the field you redefine has attributes that allow the calculated value to process correctly. The least complicated approach is to create a separate file with the specific fields you want to process for each query.

You can also use this technique with a mapped field definition and the %XLATE function to translate a field so that it appears to the program in a different manner than what exists in the database. For example, you can translate a lowercase field so the program only sees uppercase.

The example described uses FILEA as an input file. You can also update data using the OPNQRYF command. However, if you use a mapped field definition to change a field, updates to the field are ignored.

Handling Divide by Zero

Dividing by zero is considered an error by the Open Query File (OPNQRYF) command.

Record selection is normally done before field mapping errors occur (for example, where field mapping would cause a division error). Therefore, a record can be omitted (based on the QRYSLT parameter values and valid data in the record) that would have caused a divide-by-zero error. In such an instance, the record would be omitted and processing by the OPNQRYF command would continue.

If you want a zero answer, the following describes a solution that is practical for typical commercial data.

Assume you want to divide A by B giving C (stated as A / B = C). Assume the following definitions where B can be zero.

Field	Digits	Dec
A	6	2
B	3	0
C	6	2

The following algorithm can be used:

```
(A * B) / %MAX((B * B) .nnnnl)
```

The %MAX function returns the maximum value of either B * B or a small value. The small value must have enough leading zeros so that it is less than any value calculated by B * B unless B is zero. In this example, B has zero decimal positions so .1 could be used. The number of leading zeros should be 2 times the number of decimals in B. For example, if B had 2 decimal positions, then .00001 should be used.

Specify the following MAPFLD definition:

```
MAPFLD((C '(A * B) / %MAX((B * B) .1)'))
```

The intent of the first multiplication is to produce a zero dividend if B is zero. This will ensure a zero result when the division occurs. Dividing by zero does not occur if B is zero because the .1 value will be the value used as the divisor.

Summarizing Data from Database File Records (Grouping)

The group processing function allows you to summarize data from existing database records. You can specify:

- The grouping fields
- Selection values both before and after grouping
- A keyed access path over the new records
- Mapped field definitions that allow you to do such functions as sum, average, standard deviation, and variance, as well as counting the records in each group

You normally start by creating a file with a record format containing only the following types of fields:

- Grouping fields. Specified on the GRPFLD parameter that define groups. Each group contains a constant set of values for all grouping fields. The grouping fields do not need to appear in the record format identified on the FORMAT parameter.
- Aggregate fields. Defined by using the MAPFLD parameter with one or more of the following built-in functions:

%COUNT	Counts the records in a group
%SUM	A sum of the values of a field over the group
%AVG	Arithmetic average (mean) of a field, over the group
%MAX	Maximum value in the group for the field
%MIN	Minimum value in the group for the field
%STDDEV	Standard deviation of a field, over the group
%VAR	Variance of a field, over the group

- Constant fields. Allow constants to be placed in field values. The restriction that the Open Query File (OPNQRYF) command must know all fields in the output format is also true for the grouping function.

When you use group processing, you can only read the file sequentially.

Example 1: Using group processing

Assume you want to group the data by customer number and analyze the amount field. Your database file is FILEA and you create a file named FILEAA containing a record format with the following fields:

FILEA	FILEAA
Cust	Cust
Type	Count (count of records per customer)
Amt	Amtsum (summation of the amount field)
	Amtavg (average of the amount field)
	Amtmax (maximum value of the amount field)

When you define the fields in the new file, you must ensure that they are large enough to hold the results. For example, if the *Amt* field is defined as 5 digits, you may want to define the *Amtsum* field as 7 digits. Any arithmetic overflow causes your program to end abnormally.

Assume the records in FILEA have the following values:

Cust	Type	Amt
001	A	500.00
001	B	700.00
004	A	100.00
002	A	1200.00
003	B	900.00
001	A	300.00
004	A	300.00
003	B	600.00

You then create a program (PGMG) using FILEAA as input to print the records.

```
OVRDBF     FILE(FILEAA) TOFILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) FORMAT(FILEAA) KEYFLD(CUST) +
              GRPFLD(CUST) MAPFLD((COUNT '%COUNT') +
                (AMTSUM '%SUM(AMT)') +
                (AMTAVG '%AVG(AMT)') +
                (AMTMAX '%MAX(AMT)'))
CALL       PGM(PGMG) /* Created using file FILEAA as input */
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEAA)
```

The records retrieved by the program appear as:

Cust	Count	Amtsum	Amtavg	Amtmax
001	3	1500.00	500.00	700.00
002	1	1200.00	1200.00	1200.00
003	2	1500.00	750.00	900.00
004	2	400.00	200.00	300.00

Note: If you specify the GRPFLD parameter, the groups may not appear in ascending sequence. To ensure a specific sequence, you should specify the KEYFLD parameter.

Assume you want to print only the summary records in this example in which the *Amtsum* value is greater than 700.00. Because the *Amtsum* field is an aggregate field for a given customer, use the GRPSLT parameter to specify selection after grouping. Add the GRPSLT parameter:

```
GRPSLT('AMTSUM *GT 700.00') +
```

The records retrieved by your program are:

Cust	Count	Amtsum	Amtavg	Amtmax
001	3	1500.00	500.00	700.00
002	1	1200.00	1200.00	1200.00
003	2	1500.00	750.00	900.00

The Open Query File (OPNQRYF) command supports selection both before grouping (QRYSLT parameter) and after grouping (GRPSLT parameter).

Assume you want to select additional customer records in which the *Type* field is equal to A. Because *Type* is a field in the record format for file FILEA and not an aggregate field, you add the QRYSLT statement to select before grouping as follows:

```
QRYSLT('TYPE *EQ "A" ')
```

Note that fields used for selection do not have to appear in the format processed by the program.

The records retrieved by your program are:

Cust	Count	Amtsum	Amtavg	Amtmax
001	2	800.00	400.00	500.00
002	1	1200.00	1200.00	1200.00

Notice the values for CUST 001 changed because the selection took place before the grouping took place.

Assume you want to arrange the output by the *Amtavg* field in descending sequence, in addition to the previous QRYSLT parameter value. You can do this by changing the KEYFLD parameter on the OPNQRYF command as:

```
KEYFLD((AMTAVG *DESCEND))
```

The records retrieved by your program are:

Cust	Count	Amtsum	Amtavg	Amtmax
002	1	1200.00	1200.00	1200.00
001	2	800.00	400.00	500.00

Final Total-Only Processing

Final-total-only processing is a special form of grouping in which you do not specify grouping fields. Only one record is output. All of the special built-in functions for grouping can be specified. You can also specify the selection of records that make up the final total.

Example 1: Simple total processing

Assume you have a database file FILEA and decide to create file FINTOT for your final total record as follows:

FILEA	FINTOT
Code	Count (count of all the selected records)
Amt	Totamt (total of the amount field)
	Maxamt (maximum value in the amount field)

The FINTOT file is created specifically to hold the single record which is created with the final totals. You would specify:

```
OVRDBF     FILE(FINTOT) TOFILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) FORMAT(FINTOT) +
              MAPFLD((COUNT '%COUNT') +
                (TOTAMT '%SUM(AMT)') (MAXAMT '%MAX(AMT)'))
CALL       PGM(PGMG) /* Created using file FINTOT as input */
CLOF       OPNID(FILEA)
DLTOVR     FILE(FINTOT)
```

Example 2: Total-only processing with record selection

Assume you want to change the previous example so that only the records where the *Code* field is equal to B are in the final total. You can add the QRYSLT parameter as follows:

```
OVRDBF     FILE(FINTOT) TOFILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) FORMAT(FINTOT) +
              QRYSLT('CODE *EQ "B" ') MAPFLD((COUNT '%COUNT') +
                (TOTAMT '%SUM(AMT)') (MAXAMT '%MAX(AMT)'))
CALL       PGM(PGMG) /* Created using file FINTOT as input */
CLOF       OPNID(FILEA)
DLTOVR     FILE(FINTOT)
```

You can use the GRPSLT keyword with the final total function. The GRPSLT selection values you specify determines if you receive the final total record.

Example 3: Total-only processing using a new record format

Assume you want to process the new file/format with a CL program. You want to read the file and send a message with the final totals. You can specify:

```
DCLF       FILE(FINTOT)
DCL        &COUNTA *CHAR LEN(7)
DCL        &TOTAMA *CHAR LEN(9)
OVRDBF     FILE(FINTOT) TOFILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) FORMAT(FINTOT) MAPFLD((COUNT '%COUNT') +
                (TOTAMT '%SUM(AMT)'))
RCVF
CLOF       OPNID(FILEA)
CHGVAR     &COUNTA &COUNT
CHGVAR     &TOTAMTA &TOTAMT
SNDPGMMSG  MSG('COUNT=' *CAT &COUNTA *CAT +
                ' Total amount=' *CAT &TOTAMTA)
DLTOVR     FILE(FINTOT)
```

You must convert the numeric fields to character fields to include them in an impromptu message.

Controlling How the System Runs the Open Query File Command

The optimization function allows you to specify how you are going to use the results of the query.

When you use the Open Query File (OPNQRYF) command there are two steps where performance considerations exist. The first step is during the actual processing of the OPNQRYF command itself. This step decides if OPNQRYF is going to use an existing access path or build a new one for this query request. The second step when performance considerations play a role is when the application program is using the results of the OPNQRYF to process the data.

For most batch type functions, you are usually only interested in the total time of both steps mentioned above. Therefore, the default for OPNQRYF is OPTIMIZE(*ALLIO). This means that OPNQRYF will consider the total time it takes for both steps.

If you use OPNQRYF in an interactive environment, you may not be interested in processing the entire file. You may want the first screen full of records to be displayed as quickly as possible. For this reason, you would want the first step to avoid building an access path, if possible. You might specify OPTIMIZE(*FIRSTIO) in such a situation.

If you want to process the same results of OPNQRYF with multiple programs, you would want the first step to make an efficient open data path (ODP). That is, you would try to minimize the number of records that must be read by the processing program in the second step by specifying OPTIMIZE(*MINWAIT) on the OPNQRYF command.

If the KEYFLD or GRPFLD parameters on the OPNQRYF command require that an access path be built when there is no access path to share, the access path is built entirely regardless of the OPTIMIZE entry. Optimization mainly affects selection processing.

Example 1: Optimizing for the first set of records

Assume that you have an interactive job in which the operator requests all records where the *Code* field is equal to B. Your program's subfile contains 15 records per screen. You want to get the first screen of results to the operator as quickly as possible. You can specify:

```
OVRDBF    FILE(FILEA) SHARE(*YES)
OPNQRYF   FILE(FILEA) QRYSLT('CODE = "B" ') +
            SEQONLY(*YES 15) OPTIMIZE(*FIRSTIO)
CALL      PGM(PGMA)
CLOF      OPNID(FILEA)
DLTOVR    FILE(FILEA)
```

The system optimizes handling the query and fills the first buffer with records before completing the entire query regardless of whether an access path already exists over the *Code* field.

Example 2: Optimizing to minimize the number of records read

Assume that you have multiple programs that will access the same file which is built by the Open Query File (OPNQRYF) command. In this case, you will want to optimize the performance so that the application programs read only the data they are interested in. This means that you want OPNQRYF to perform the selection as efficiently as possible. You could specify:

```
OVRDBF   FILE(FILEA) SHARE(*YES)
OPNQRYF  FILE(FILEA) QRYSLT('CODE *EQ "B"') +
         KEYFLD(CUST) OPTIMIZE(*MINWAIT)
CALL     PGM(PGMA)
POSDBF   OPNID(FILEA) POSITION(*START)
CALL     PGM(PGMB)
CLOF     OPNID(FILEA)
DLTOVR   FILE(FILEA)
```

Considerations for Creating a File and Using the FORMAT Parameter

You must specify a record format name on the FORMAT parameter when you request join processing by specifying multiple entries on the FILE parameter (that is, you cannot specify FORMAT(*FILE)). Also, a record format name is normally specified with the grouping function or when you specify a complex expression on the MAPFLD parameter to define a derived field. Consider the following:

- The record format name is any name you select. It can differ from the format name in the database file you want to query.
- The field names are any names you select. If the field names are unique in the database files you are querying, the system implicitly maps the values for any fields with the same name in a queried file record format (FILE parameter) and in the query result format (FORMAT parameter). See Example 1 under "Dynamically Joining Database Files without DDS" on page 9-17 for more information.
- If the field names are unique, but the attributes differ between the file specified on the FILE parameter and the file specified on the FORMAT parameter, the data is implicitly mapped.

- The correct field attributes must be used when using the MAPFLD parameter to define derived fields. For example, if you are using the grouping %SUM function, you must define a field that is large enough to contain the total. If not, an arithmetic overflow occurs and an exception is sent to the program.
- Decimal alignment occurs for all field values mapped to the record format identified on the FORMAT parameter. Assume you have a field in the query result record format with 5 digits with 0 decimals, and the value that was calculated or must be mapped to that field is 0.12345. You will receive a result of 0 in your field because digits to the right of the decimal point are truncated.

Considerations for Arranging Records

The default processing for the Open Query File (OPNQRYF) command provides records in any order that improves performance and does not conflict with the order specified on the KEYFLD parameter. Therefore, unless you specify the KEYFLD parameter to either name specific key fields or specify KEYFLD(*FILE), the sequence of the records returned to your program can vary each time you run the same Open Query File (OPNQRYF) command.

Considerations for DDM Files

The Open Query File (OPNQRYF) command can process DDM files. All files identified on the FILE parameter must exist on the same IBM AS/400 system or System/38 target system. An OPNQRYF which specifies group processing and uses a DDM file requires that both the source and target system be the same type (either both System/38 or both AS/400 systems).

Considerations for Writing a High-Level Language Program

For the method described under "Using an Existing Record Format in the File" on page 9-4 (where the FORMAT parameter is omitted), your high-level language program is coded as if you are directly accessing the database file. Selection or sequencing occurs external to your program, and the program receives the selected records in the order you specified. The program does not receive records that are omitted by your selection values. This same function occurs if you process through a logical file with select/omit values.

If you use the FORMAT parameter, your program specifies the same file name used on the FORMAT parameter. The program is written as if this file contains actual data.

If you read the file sequentially, your high-level language can automatically specify that the key fields are ignored. Normally you write the program as if it is reading records in arrival sequence. If the KEYFLD parameter is used on the Open Query File (OPNQRYF) command, you receive a diagnostic message, which can be ignored.

If you process the file randomly by keys, your high-level language probably requires a key specification. If you have selection values, it can prevent your program from accessing a record that exists in the database. A `Record not found` condition can occur on a random read whether the OPNQRYF command was used or whether a logical file created using DDS select/omit logic was used.

In some cases, you can monitor exceptions caused by mapping errors such as arithmetic overflow, but it is better to define the attributes of all fields to correctly handle the results.

Messages Sent When the Open Query File (OPNQRYF) Command Is Run

When the OPNQRYF command is run, messages are sent informing the interactive user of the status of the OPNQRYF request. For example, a message would be sent to the user if a keyed access path was built by the OPNQRYF to satisfy the request. The following messages might be sent during a run of the OPNQRYF command:

Message Identifier	Description
CPI4301	`Query running.`
CPI4302	`Query running.  Building access path...`
CPI4303	`Query running.  Creating copy of file...`
CPI4304	`Query running.  Selection complete...`
CPI4305	`Query running.  Sorting copy of file...`
CPI4306	`Query running.  Building access path from file...`
CPI4011	`Query running.  Number of records processed...`

To stop these status messages from appearing, see the discussion about message handling in the *CL Programmer's Guide*.

Using the Open Query File (OPNQRYF) Command for More Than Just Input

The OPNQRYF command supports the OPTION parameter to determine the type of processing. The default is OPTION(*INP), so the file is opened for input only. You can also use other OPTION values on the OPNQRYF command and a high-level language program to add, update, or delete records through the open query file. However, if you specify the UNIQUEKEY, GRPFLD, or GRPSLT parameters, use one of the aggregate functions, or specify multiple files on the FILE parameter, your use of the file is restricted to input only.

A join logical file is limited to input-only processing. A view is limited to input-only processing, if group, join, union, or distinct processing is specified in the definition of the view.

If you want to change a field value from the current value to a different value in some of the records in a file, you can use a combination of the OPNQRYF command and a specific high-level language program. For example, assume you want to change all the records where the *Flda* field is equal to ABC so that the *Flda* field is equal to XYZ. You can specify:

```
OVRDBF    FILE(FILEA) SHARE(*YES)
OPNQRYF   FILE(FILEA) OPTION(*ALL) QRYSLT('FLDA *EQ "ABC" ')
CALL      PGM(PGMA)
CLOF      OPNID(FILEA)
DLTOVR    FILE(FILEA)
```

Program PGMA processes all records it can read, but the query selection restricts these to records where the *Flda* field is equal to ABC. The program changes the field value in each record to XYZ and updates the record.

You can also delete records in a database file using the OPNQRYF command. For example, assume you have a field in your record that, if equal to X, means the record should be deleted. Your program can be written to delete any records it reads and use the OPNQRYF command to select those to be deleted such as:

```
OVRDBF    FILE(FILEA) SHARE(*YES)
OPNQRYF   FILE(FILEA) OPTION(*ALL) QRYSLT('DLTCOD *EQ "X" ')
CALL      PGM(PGMB)
CLOF      OPNID(FILEA)
DLTOVR    FILE(FILEA)
```

You can also add records by using the OPNQRYF command. However, if the query specifications include selection values, your program can be prevented from reading the added records because of the selection values.

Using the Open Query File (OPNQRYF) Command for Random Processing

Most of the previous examples show the OPNQRYF command using sequential processing. Random processing operations (for example, the RPG/400 operation CHAIN or the COBOL/400 operation READ) can be used in most cases. However, if you are using the group or unique-key functions, you cannot process the file randomly.

Performance Considerations

The best performance can occur when the Open Query File (OPNQRYF) command uses an existing keyed access path. For example, if you want to select all the records where the *Code* field is equal to B and an access path exists over the *Code* field, the system can use the access path to perform the selection rather than read the records and select at run time (dynamic selection).

Part of the OPNQRYF processing is to determine what is the fastest approach to satisfying your request. If the file you are using is large and most of the records have the *Code* field equal to B, it is faster to use arrival sequence processing than to use an existing keyed access path. Your program will still see the same records. OPNQRYF can only make this type of decision if an access path exists on the *Code* field. In general, if your request will include approximately 20% or more of the number of records in the file, OPNQRYF will tend to ignore the existing access paths and read the file in arrival sequence. You can force the use of an existing access path in most cases by specifying OPTIMIZE(*FIRSTIO) on the OPNQRYF command.

If no access path exists over the *Code* field, the program reads all of the records in the file and passes only the selected records to your program. That is, the file is processed in arrival sequence.

The system can perform selection faster than your application program. If no appropriate keyed access path exists, either your program or the system makes the selection of the records you want to process. Allowing the system to perform the selection process is considerably faster than passing all the records to your application program.

This is especially true if you are opening a file for update operations because individual records must be passed to your program, and locks are placed on every record read (in case your program needs to update the record). By letting the system perform the record selection, the only records passed to your program and locked are those that meet your selection values.

If you use the KEYFLD parameter to request a specific sequence for reading records, the fastest performance results if an access path already exists that uses the same key specification or if a keyed access path exists that is similar to your specifications (such as a key that contains all the fields you specified plus some additional fields on the end of the key). This is also true for the GRPFLD parameter and on the to-fields of the JFLD parameter. If no such access path exists, the system builds an access path and maintains it as long as the file is open in your job.

Processing all of the records in a file by an access path that does not already exist is generally not as efficient as using a full record sort, if the number of records to be arranged (not necessarily the total number of records in the file) exceeds 1000. While it is generally faster to build the keyed access path than to do the sort, faster processing allowed by the use of arrival sequence processing normally favors sorting the data when looking at the total job time. If a usable access path already exists, using the access path can be faster than sorting the data.

If you use the grouping function, faster performance is achieved if you specify selection before grouping (QRYSLT parameter) instead of selection after grouping (GRPSLT parameter). Only use the GRPSLT parameter for comparisons involving aggregate functions.

For most uses of the OPNQRYF command, new or existing access paths are used to access the data and present it to your program. In some cases, the system must create a temporary file and then build an access path over the temporary structure. The rules for this situation are complex, but the following are typical cases in which this occurs:

- When you specify both the GRPFLD and KEYFLD parameters, but they are not the same
- When you specify a dynamic join, and the KEYFLD parameter describes key fields from different physical files

When a dynamic join occurs, OPNQRYF will attempt to improve performance by joining the file with the smallest number of records with the file with the largest number of records. OPNQRYF tries to avoid building a temporary file. Consequently, if your sequence specifications cause a temporary result, OPNQRYF can join the file with the large number of records to the file with the small number of records thus causing slower performance. You can force the join sequence by specifying JORDER(*FILE).

Performance Comparisons with Other Database Functions

The Open Query File (OPNQRYF) command uses the same database support as logical files and join logical files. Therefore, the performance of functions like building a keyed access path or doing a join operation will be the same.

The selection functions done by the OPNQRYF command (for the QRYSLT and GRPSLT parameters) are similar to logical file select/omit. The main difference is that for the OPNQRYF command, the system decides whether to use access path selection or dynamic selection (similar to omitting or specifying the DYNSLT keyword in the DDS for a logical file), as a result of the access paths available on the system and what value was specified on the OPTIMIZE parameter.

Considerations for Field Use

When the grouping function is used, all fields in the record format for the open query file (FORMAT parameter) and all key fields (KEYFLD parameter) must either be grouping fields (specified on the GRPFLD parameter) or mapped fields (specified on the MAPFLD parameter) that are defined using only grouping fields, constants, and aggregate functions. The aggregate functions are: %AVG, %COUNT, %MAX (using only one operand), %MIN (using only one operand), %STDDEV, %SUM, and %VAR. Group processing is required in the following cases:

- When you specify grouping field names on the GRPFLD parameter
- When you specify group selection values on the GRPSLT parameter
- When a mapped field that you specified on the MAPFLD parameter uses an aggregate function in its definition

Fields contained in a record format, identified on the FILE parameter, and defined (in the DDS used to create the file) with a usage value of N (neither input nor output) cannot be specified on any parameter of the OPNQRYF Only fields defined as either I (input-only) or B (both input and output) usage can be specified. Any fields with usage defined as N in the record format identified on the FORMAT parameter are ignored by the OPNQRYF command.

Fields in the open query file records normally have the same usage attribute (input-only or both input and output) as the fields in the record format identified on the FORMAT parameter, with the exceptions noted below. If the file is opened for any option (OPTION parameter) that includes output or update and any usage, and if any B (both input and output) field in the record format identified on the FORMAT parameter is changed to I (input only) in the open query file record format, then an information message is sent by the OPNQRYF command.

If you request join processing or group processing, or if you specify UNIQUEKEY processing, all fields in the query records are given input-only use. Any mapping from an input-only field from the file being processed (identified on the FILE parameter) is given input-only use in the open query file record format. Fields defined using the MAPFLD parameter are normally given input-only use in the open query file. A field defined on the MAPFLD parameter is given a value that matches the use of its constituent field if all of the following are true:

- Input-only is not required because of any of the conditions previously described in this section.
- The field-definition expression specified on the MAPFLD parameter is a field name (no operators or built-in functions).
- The field used in the field-definition expression exists in one of the file, member, or record formats specified on the FILE parameter (not in another field defined using the MAPFLD parameter).
- The base field and the mapped field are compatible field types (the mapping does not mix numeric and character field types, unless the mapping is between zoned and character fields of the same length).
- If the base field is binary with nonzero decimal precision, the mapped field must also be binary and have the same precision.

Considerations for Files Shared in a Job

In order for your application program to use the open data path built by the Open Query File (OPNQRYF) command, your program must share the query file. If your program does not open the query file as shared, then it actually does a full open of the file it was originally compiled to use (not the query open data path built by the OPNQRYF command). Your program will share the query open data path, depending on the following conditions:

- Your application program must open the file as shared. Your program meets this condition when the first or only member queried (as specified on the FILE parameter) has an attribute of SHARE(*YES). If the first or only member has an attribute of SHARE(*NO), then you must specify SHARE(*YES) in an Override with Database File (OVRDBF) command before calling your program.
- The file opened by your application program must have the same name as the file opened by the OPNQRYF command. Your program meets this condition when the file specified in your program has the same file and member name as the first or only member queried (as specified on the FILE parameter). If the first or only member has a different name, then you must specify an Override with Database File (OVRDBF) command of the name of the file your program was compiled against to the name of the first or only member queried.

The OPNQRYF command never shares an existing open data path in the job. If an open data path that can be shared already exists in the job with the same file, library, and member name as the query open data path, an error message is sent to report the error and the query file is not opened.

Subsequent shared opens adhere to the same open options (such as SEQONLY) that were in effect when the OPNQRYF command was run.

See "Sharing Database Files in the Same Job" on page 8-7 for more information about sharing files in a job.

Considerations for Checking If the Record Format Description Changed

If record format level checking is indicated, the format level number of the open query file record format (identified on the FORMAT parameter) is checked against the record format your program was compiled against. This occurs when your program shares the previously opened query file. Your program's shared open is checked for record format level if the following conditions are met:

- The first or only file queried (as specified on the FILE parameter) must have the LVLCHK(*YES) attribute.
- There must not be an override of the first or only file queried to LVLCHK(*NO).

Other Run Time Considerations

Overrides can change the name of the file, library, and member that should be processed by the open query file. (However, any parameter values other than TOFILE, MBR, LVLCHK, INHWRT, or SEQONLY specified on an Override with Database File (OVRDBF) command are ignored by the OPNQRYF command.) If a name change override applies to the first or only member queried, any additional overrides must be against the new name, not the name specified for the FILE parameter on the OPNQRYF command.

Typical Errors When Using the Open Query File (OPNQRYF) Command

Several functions must be correctly specified for the OPNQRYF command and your program to get the correct results. The Display Job (DSPJOB) command is your most useful tool if problems occur. This command supports both the open files option and the file overrides option. You should look at both of these if you are having problems.

These are the most common problems and how to correct them:

- Shared open data path (ODP). The OPNQRYF command operates through a shared ODP. In order for the file to process correctly, the member must be opened for a shared ODP. If you are having problems, use the open files option on the DSPJOB command to determine if the member is opened and has a shared ODP.

  There are normally two reasons that the file is not open:
  - The member to be processed must be SHARE(*YES). Either use an Override with Database File (OVRDBF) command or permanently change the member.
  - The file is closed. If you operate from a command entry line and successfully run the OPNQRYF command, then run a command that fails and sends an error message, the command entry function internally runs a Reclaim Resources (RCLRSC) command. This closes all open files, including the open query file, unless TYPE(*PERM) was specified on the OPNQRYF command. You must run the OPNQRYF command again.
- Level check. Level checking is normally used because it ensures that your program is running against the same record format that the program was compiled with. If you are experiencing level check problems, it is normally because of one of the following:
  - The record format was changed since the program was created. Creating the program again should correct the problem.
  - An override is directing the program to an incorrect file. Use the file overrides option on the DSPJOB command to ensure that the overrides are correctly specified.
  - The FORMAT parameter is needed but is either not specified or incorrectly specified. When a file is processed with the FORMAT parameter, you must ensure:
    - The Override with Database File (OVRDBF) command, used with the TOFILE parameter, describes the first file on the FILE parameter of the Open Query File (OPNQRYF) command.
    - The FORMAT parameter identifies the file that contains the format used to create the program.
  - The FORMAT parameter is used to process a format from a different file (for example, for group processing), but SHARE(*YES) was not requested on the OVRDBF command.
- The file to be processed is at end of file. The normal use of the OPNQRYF command is to process a file sequentially where you can only process the file once. At that point, the position of the file is at the end of the file and you will not receive any records if you attempt to process it again. To process the file again from the start, you must either run the OPNQRYF command again or reposition the file before processing. You can reposition the file by using the Position Database File (POSDBF) command, or through a high-level language program statement.
- No records exist. This can be caused when you use the FORMAT keyword, but do not specify the OVRDBF command.

- Syntax errors. The system found an error in the specification of the OPNQRYF command.
- Operation not valid. The definition of the query does not include the KEYFLD parameter, but the high-level language program attempts to read the query file using a key field.
- Get option not valid. The high-level language program attempted to read a record or set a record position before the current record position, and the query file used either the group by option, the unique key option, or the distinct option on the SQL/400 statement.

Chapter 10. Basic Database File Operations

The basic database file operations that can be performed in a program are discussed in this chapter. The operations include: setting a position in the database file, reading records from the file, updating records in the file, adding records to the file, and deleting records from the file.

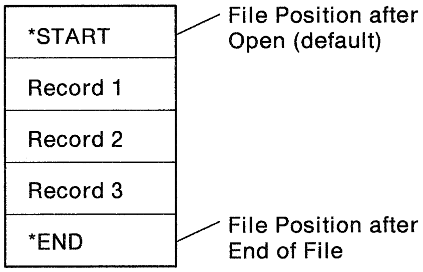
Setting a Position in the File

After a file is opened by a job, the system maintains a position in the file for that job. The file position is used in processing the file. For example, if a program does a read operation requesting the next sequential record, the system uses the file position to determine which record to return to the program. The system will then set the file position to the record just read, so that another read operation requesting the next sequential record will return the correct record. The system keeps track of all file positions for each job. In addition, each job can have multiple positions in the same file.

The file position is first set to the position specified in the POSITION parameter on the Override with Database File (OVRDBF) command. If you do not use an OVRDBF command, or if you take the default for the POSITION parameter, the file position is set just before the first record in the member's access path.

A program can change the current file position by using the appropriate high-level language program file positioning operation (for example, SETLL in RPG/400 or START in COBOL/400). A program can also change the file position by using the CL Position Database File (POSDBF) command.

At end of file, after the last read, the file member is positioned to *START or *END file positions, depending on whether the program was reading forward or backward through the file. The following diagram shows *START and *END file positions.


*START
File Position after Open (default)
Record 1
Record 2
Record 3
*END
File Position after End of File

RSLH296-1

Only a read operation, force-end-of-data operation, high-level language positioning operation, or specific CL command to change the file position can change the file position. Add, update, and delete operations do not change the file position. After a read operation, the file is positioned to the new record. This record is then returned to your program. After the read operation is completed, the file is positioned at the record just returned to your program. If the member is open for input, a force-end-of-data operation positions the file after the last record in the file (*END) and sends the end-of-file message to your program.

For sequential read operations, the current file position is used to locate the next or previous record on the access path. For read-by-key or read-by-relative-record-number operations, the file position is not used. If POSITION(*NONE) is specified at open time, no starting file position is set. In this case, you must establish a file position in your program, if you are going to read sequentially.

If end-of-file delay was specified for the file on an Override With Database File (OVRDBF) command, the file is not positioned to *START or *END when the program reads the last record. The file remains positioned at the last record read. A file with end-of-file delay processing specified is positioned to *START or *END only when a force-end-of-data (FEOD) occurs or a controlled job end occurs. For more information about end-of-file delay, see "Waiting for More Records When End of File Is Reached" on page 10-4.

You can also use the Position Database File (POSDBF) command to set or change the current position in your file for files opened using either the Open Database File (OPNDBF) command or the Open Query File (OPNQRYF) command.

Reading Database Records

The AS/400 system provides a number of ways to read database records. The next sections describe those ways in detail. (Some high-level languages do not support all of the read operations available on the system. See your high-level language guide for more information about reading database records.)

Reading Database Records Using an Arrival Sequence Access Path

The system performs the following read operations based on the operations you specify using your high-level language. These operations are allowed if the file was defined with an arrival sequence access path; or if the file was defined with a keyed sequence access path with the ignore-keyed-sequence-access-path option specified in the program, on the Open Database File (OPNDBF) command, or on the Open Query File (OPNQRYF) command. See "Ignoring the Keyed Sequence Access Path" on page 8-3 for more details about the option to ignore a keyed sequence access path.

Note: Your high-level language may not allow all of the following read operations. Refer to your high-level language guide to determine which operations are allowed by the language.

Read Next: Positions the file to and gets the next record that is not deleted in the arrival sequence access path. Deleted records between the current position in the file and the next active record are skipped. (The READ statement in RPG/400 and the READ NEXT statement in COBOL/400 are examples of this operation.)

Read Previous: Positions the file to and gets the previous active record in the arrival sequence access path. Deleted records between the current file position and the previous active record are skipped. (The READP statement in RPG/400 and the READ PRIOR statement in COBOL/400 are examples of this operation.)

Read First: Positions the file to and gets the first active record in the arrival sequence access path.

Read Last: Positions the file to and gets the last active record in the arrival sequence access path.

Read Same: Gets the record that is identified by the current position in the file. The file position is not changed.

Read by Relative Record Number: Positions the file to and gets the record in the arrival sequence access path that is identified by the relative record number. The relative record number must identify an active record and must be less than or equal to the largest active relative record number in the member. This operation also reads the record in the arrival sequence access path identified by the current file position plus or minus some number of records. (The CHAIN statement in RPG/400 and the READ statement in COBOL/400 are examples of this operation.)

Reading Database Records Using a Keyed Sequence Access Path

The system performs the following read operations based on the statements you specify using your high-level language. These operations can be used with a keyed sequence access path to get database records.

When a keyed sequence access path is used, a read operation cannot position to the storage occupied by a deleted record.

Note: Your high-level language may not allow all of the following operations. Refer to your high-level language guide to determine which operations are allowed by the language.

Read Next: Gets the next record on the keyed access path. If a record format name is specified, this operation gets the next record in the keyed access path that matches the record format. The current position in the file is used to locate the next record. (The READ statement in RPG/400 and the READ NEXT statement in COBOL/400 are examples of this operation.)

Read Previous: Gets the previous record on the keyed access path. If a record format name is specified, this operation gets the previous record on the keyed access path that matches the record format. The current position in the file is used to locate the previous record. (The READP statement in RPG/400 and the READ PRIOR statement in COBOL/400 are examples of this operation.)

Read First: Gets the first record on the keyed access path. If a record format name is specified, this operation gets the first record on the access path with the specified format name.

Read Last: Gets the last record on the keyed access path. If a record format name is specified, this operation gets the last record on the access path with the specified format name.

Read Same: Gets the record that is identified by the current file position. The position in the file is not changed.

Read by Key: Gets the record identified by the key value. Key operations of equal, equal or after, equal or before, read previous key equal, read next key equal, after, or before can be specified. If a format name is specified, the system searches for a record of the specified key value and record format name. If a format name is not specified, the entire keyed access path is searched for the specified key value. If the key definition for the file includes multiple key fields, a partial key can be specified (you can specify either the number of key fields or the key length to be used). This allows you to do generic key searches. If the program does not specify a number of key fields, the system assumes a default number of key fields. This default varies depending on if a record format name is passed by the program. If a record format name is passed, the default number of key fields is the total number of key fields defined for that format. If a record format name is not passed, the default number of key fields is the maximum number of key fields that are common across all record formats in the access path. The program must supply enough key data to match the number of key fields assumed by the system. (The CHAIN statement in RPG/400 and the READ statement in COBOL/400 are examples of this operation.)

Read by Relative Record Number: For a keyed access path, the relative record number can be used. This is the relative record number in the arrival sequence, even though the member opened has a keyed access path. If the member contains multiple record formats, a record format name must be specified. In this case, you are requesting a record in the associated physical file member that matches the record format specified. If the member opened contains select/omit statements and the record identified by the relative record number is omitted from the keyed access path, an error message is sent to your program and the operation is not allowed. After the operation is completed, the file is positioned to the key value in the keyed access path that is contained in the physical record, which was identified by the relative record number. This operation also gets the record in the keyed sequence access path identified by the current file position plus or minus some number of records. (The CHAIN statement in RPG/400 and the READ statement in COBOL/400 are examples of this operation.)

Waiting for More Records When End of File Is Reached

End-of-file delay is a method of continuing to read sequentially from a database file (logical or physical) after an end-of-file condition occurs. When an end-of-file condition occurs on a file being read sequentially (for example, next/previous record) and you have specified an end-of-file delay time (EOFDLY parameter on the Override with Database File [OVRDBF] command), the system waits for the time you specified. At the end of the delay time, another read is done to determine if any new records were added to the file. If records were added, normal record processing is done until an end-of-file condition occurs again. If records were not added to the file, the system waits again for the time specified. Special consideration should be taken when using end-of-file delay on a logical file with select/omit specifications, opened so that the keyed sequence access path is not used. In this case, once end-of-file is reached, the system retrieves only those records added to a based-on physical file that meet the select/omit specifications of the logical file.

Also, special consideration should be taken when using end-of-file delay on a file with a keyed sequence access path, opened so that the keyed sequence access path is used. In this case, once end-of-file is reached, the system retrieves only those records added to the file or those records updated in the file that meet the specification of the read operation using the keyed sequence access path.

For example, end-of-file delay is used on a keyed file that has a numeric key field in ascending order. An application program reads the records in the file using the keyed sequence access path. The application program performs a read next operation and gets a record that has a key value of 99. The application program performs another read next and no more records are found in the file, so the system attempts to read the file again after the specified end-of-file delay time. If a record is added to the file or a record is updated, and the record has a key value less than 99, the system does not retrieve the record. If a record is added to the file or a record is updated and the record has a key value greater than or equal to 99, the system retrieves the record.

For end-of-file delay times equal to or greater than 10 seconds, the job is eligible to be removed from main storage during the wait time. If you do not want the job eligible to be moved from main storage, specify PURGE(*NO) on the Create Class (CRTCLS) command for the CLASS the job is using.

To indicate which jobs have an end-of-file delay in progress, the status field of the Work with Active Jobs (WRKACTJOB) display shows an end-of-file wait or end-of-file activity level for jobs that are waiting for a record.

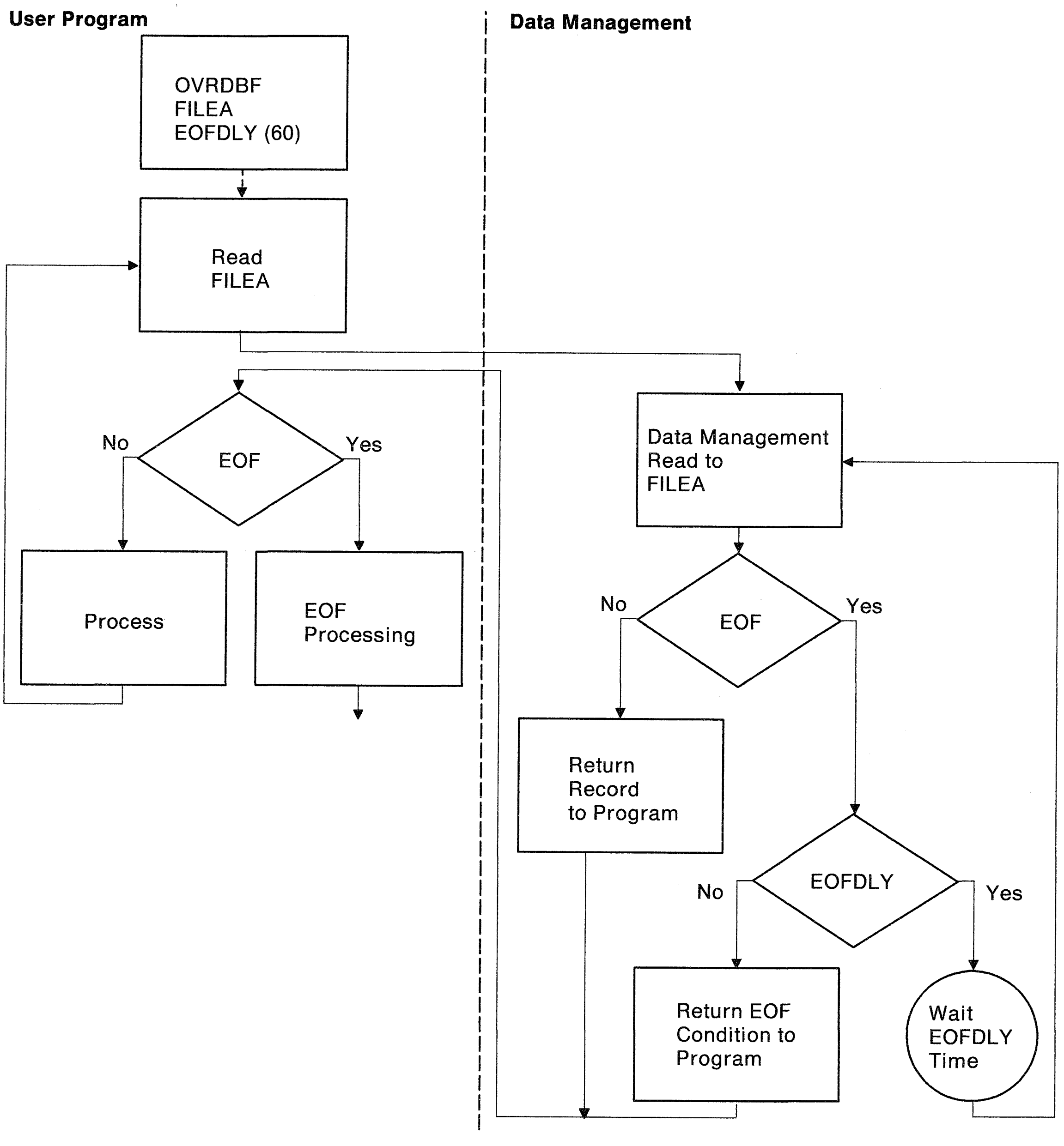
If a job uses end-of-file-delay and commitment control, it can hold its record locks for a longer period of time. This increases the chances that some other job can try to access those same records and be locked out. For that reason, be careful when using end-of-file-delay and commitment control in the same job.

If a file is shared, the Override with Database File (OVRDBF) command specifying an end-of-file delay must be requested before the *first* open of the file because overrides are ignored that are specified after the shared file is opened.

There are several ways to end a job that is waiting for more records because of an end-of-file-delay specified on the Override with Database File (OVRDBF) command:

- Write a record to the file with the end-of-file-delay that will be recognized by the application program as a last record. The application program can then specify a force-end-of-data (FEOD) operation. An FEOD operation allows the program to complete normal end-of-file processing.
- Do a controlled end of a job by specifying OPTION(*CNTRLD) on the End Job (ENDJOB) command, with a DELAY parameter value time greater than the EOFDLY time. The DELAY parameter time specified must allow time for the EOFDLY time to run out, time to process any new records that have been put in the file, and any end-of-file processing required in your application. After new records are processed, the system signals end of file, and a normal end-of-file condition occurs.
- Specify OPTION(*IMMED) on the End Job (ENDJOB) command. No end-of-file processing is done.
- If the job is interactive, press the System Request key to end the last request.

The following is an example of end-of-file delay operation:



RSLH300-1

The actual processing of the EOFDLY parameter is more complex than shown because it is possible to force a true end-of-file if OPTION(*CNTRLD) on the End Job (ENDJOB) command is used with a long delay time.

The job does not become active whenever a new record is added to the file. The job becomes active after the specified end-of-file delay time ends. When the job becomes active, the system checks for any new records. If new records were added, the application program gets control and processes all new records, then waits again. Because of this, the job takes on the characteristic of a batch job when it is processing. For example, it normally processes a batch of requests. When the batch is completed, the job becomes inactive. If the delay is small, you can cause excessive system overhead because of the internal processing required to start the job and check for new records. Normally, only a small amount of overhead is used for a job waiting during end-of-file delay.

Note: When the job is inactive (waiting) it is in a long-wait status, which means it was released from an activity level. After the long-wait status is satisfied, the system reschedules the job in an activity level. (See the *Work Management Guide* for more information about activity levels.)

Releasing Locked Records

The system automatically releases a locked record when the record is updated, deleted, or when you read another record in the file. However, you may want to release a locked record without performing these operations. Some high-level languages support an operation to release a locked record. See your high-level language guide for more information about releasing record locks.

Note: The rules for locking are different if your job is running under commitment control.

Updating Database Records

The update operation allows you to change an existing database record in a logical or physical file. (The UPDAT statement in RPG/400 and the REWRITE statement in COBOL/400 are examples of this type operation.) Before you update a database record, the record must first be read and locked. The lock is obtained by specifying the update option on any of the read operations listed under the "Reading Database Records Using an Arrival Sequence Access Path" on page 10-2 and "Reading Database Records Using a Keyed Sequence Access Path" on page 10-3.

If you issue several read operations with the update option specified, each read operation releases the lock on the previous record before attempting to locate and lock the new record. When you do the update operation, the system assumes that you are updating the currently locked record. Therefore, you do not have to identify the record to be updated on the update operation. After the update operation is done, the system releases the lock.

Note: The rules for locking are different if your job is running under commitment control.

If the update operation changes a key field in an access path for which immediate maintenance is specified, the access path is updated if the high-level language allows it. (Some high-level languages do not allow changes to the key field in an update operation.)

If you request a read operation on a record that is already locked for update, you must wait until the record is released or the time specified by the WAITRCD parameter that was specified on the create file or override commands has been exceeded. If the WAITRCD time is exceeded without the lock being released, an exception is returned to your program and a message is sent to your job stating the file, member, relative record number, and the job which has the lock.

Adding Database Records

The write operation is used to add a new record to a physical database file member. (The WRITE statement in RPG/400 and the WRITE statement in COBOL/400 are examples of this operation.) New records can be added to a physical file member or to a logical file member that is based on the physical file member. When using a multiple format logical file, a record format name must be supplied to tell the system which physical file member to add the record to.

The new record is normally added at the end of the physical file member. The next available relative record number (including deleted records) is assigned to the new record. Some high-level languages allow you to write a new record over a deleted record position (for example, the WRITE statement in COBOL/400 when the file organization is defined as RELATIVE). For more information about writing records over deleted record positions, see your high-level language guide.

If you are adding new records to a file member that has a keyed access path, the new record appears in the keyed access path immediately at the location defined by the record key. If you are adding records to a logical member that contains select/omit values, the omit values can prevent the new record from appearing in the member's access path.

The SIZE parameter on the Create Physical File (CRTPF) and Create Source Physical File (CRTSRCPF) commands determines how many records can be added to a physical file member.

Identifying Which Record Format to Add in a File with Multiple Formats

If your application uses a file name instead of a record format name for records to be added to the database, and if the file used is a logical file with more than one record format, you need to write a format selector program to determine where a record should be placed in the database. A format selector can be a CL program or a high-level language program.

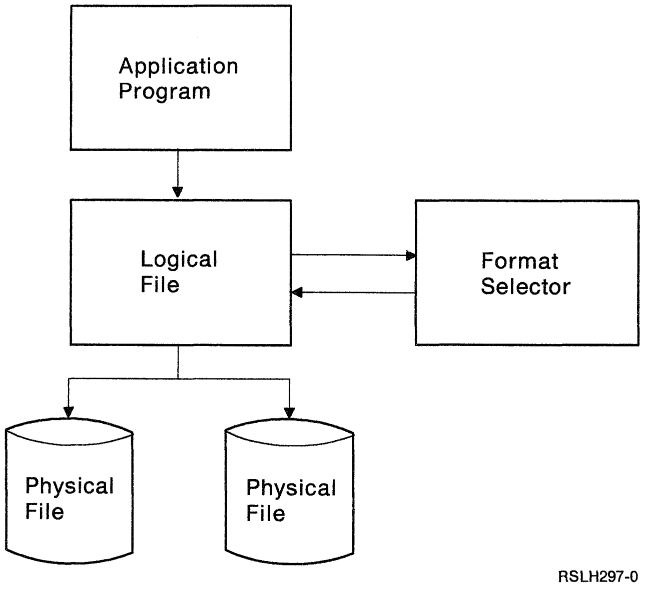
A format selector program must be used if all of the following are true:

- The logical file is not a join and not a view logical file.
- The logical file is based on multiple physical files.
- The program uses a file name instead of a record format name on the add operation.

If you do not write a format selector program for this situation, your program ends with an error when it tries to add a record to the database.

Note: A format selector program cannot be used to select a member if a file has multiple members; it can only select a record format.

When an application program wants to add a record to the database file, the system calls the format selector program. The format selector program examines the record and specifies the record format to be used. The system then adds the record to the database file using the specified record format name.



The following example shows the programming statements for a format selector program written in RPG/400:

```
*ENTRY  PLIST
        PARM                 RECORD 80 (length of longest
                                        record expected)
        PARM                 FORMAT 10
        MOVELRECORD          BYTE 1
BYTE    IFEQ 'A'
        MOVEL'HDR'           FORMAT
        ELSE
        MOVEL'DTL'           FORMAT
        END
        RETRN
```

The format selector receives the record in the first parameter; therefore, this field must be declared to be the length of the longest record expected by the format selector. The format selector can access any portion of the record to determine the record format name. In this example, the format selector checks the first character in the record for the character A. If the first character is A, the format selector moves the record format name HDR into the second parameter (FORMAT). If the character is not A, the format selector moves the record format name DTL into the second parameter.

The format selector uses the second parameter, which is a 10-character field, to pass the record format name to the system. When the system knows the name of the record format, it adds the record to the database.

You do not need a format selector if:

- You are doing updates only. For updates, your program already retrieved the record, and the system knows which physical file the record came from.
- Your application program specifies the record format name instead of a file name for an add or delete operation.
- All the records used by your application program are contained in one physical file.

To create the format selector, you use the create program command for the language in which you wrote the program. You cannot specify USRPRF(*OWNER) on the create command. The format selector must run under the user's user profile not the owner's user profile.

In addition, for security and integrity and because performance would be severely affected, you must not have any calls or input/output operations within the format selector.

The name of the format selector is specified on the FMTSLR parameter of the Create Logical File (CRTLF), Change Logical File (CHGLF), or Override with Database File (OVRDBF) command. The format selector program does not have to exist when the file is created, but it must exist when the application program is run.

Using the Force-End-Of-Data Operation

The force-end-of-data (FEOD) operation allows you to force all changes to a file made by your program to auxiliary storage. Normally, the system determines when to force changes to auxiliary storage. However, you can use the FEOD operation to ensure that all changes are forced to auxiliary storage. Some high-level languages do not support the FEOD operation. See your high-level language guide for more information about the FEOD operation.

Deleting Database Records

The delete operation allows you to delete an existing database record. (The DELET statement in RPG/400 and the DELETE statement in COBOL/400 are examples of this operation.) To delete a database record, the record must first be read and locked. The record is locked by specifying the update option on any of the read operations listed under "Reading Database Records Using an Arrival Sequence Access Path" on page 10-2. The rules for locking records for deletion and identifying which record to delete are the same as for update operations.

Note: Some high-level languages do *not* require that you read the record first. These languages allow you to simply specify which record you want deleted on the delete statement. For example, RPG/400 allows you to delete a record without first reading it.

When a database record is deleted, the physical record is marked as deleted. This is true even if the delete operation is done through a logical file. A deleted record *cannot* be read. The record is removed from all keyed access paths that contain the record. The relative record number of the deleted record remains the same. All other relative record numbers within the physical file member do not change.

The space used by the deleted record remains in the file, but it is not reused until:

- The Reorganize Physical File Member (RGZPFM) command is run to compress and free these spaces in the file member. See "Reorganizing Data in Physical File Members" on page 13-3 for more information about this command.
- Your program writes a record to the file by relative record number and the relative record number used is the same as that of the deleted record.

The system does not allow you to retrieve the data for a deleted record. You can, however, write a new record to the position (relative record number) associated with a deleted record. The write operation replaces the deleted record with a new record. See your high-level language guide for more details about how to write a record to a specific position (relative record number) in the file.

To write a record to the relative record number of a deleted record, that relative record number must exist in the physical file member. You can delete a record in the file using the delete operation in your high-level language. You can also delete records in your file using the the Initialize Physical File Member (INZPFM) command. The INZPFM command can initialize the entire physical file member to deleted records. For more information about the INZPFM command, see "Initializing Data in a Physical File Member" on page 13-2.

Chapter 11. Closing a Database File

When your program completes processing a database file member, it should close the file. Closing a database file disconnects your program from the file. The close operation releases all record locks and releases all file member locks, forces all changes made through the open data path (ODP) to auxiliary storage, then destroys the ODP. (When a shared file is closed but the ODP remains open, the functions differ. For more information about shared files, see "Sharing Database Files in the Same Job" on page 8-7.)

The ways you can close a database file in a program include:

- High-level language close statements
- Close File (CLOF) command
- Reclaim Resources (RCLRSC) command

Most high-level languages allow you to specify that you want to close your database files. For more information about how to close a database file in a high-level language program, see your high-level language guide.

You can use the Close File (CLOF) command to close database files that were opened using either the Open Database File (OPNDBF) or Open Query File (OPNQRYF) commands.

You can also close database files by running the Reclaim Resources (RCLRSC) command. The RCLRSC command releases all locks (except, under commitment control, locks on records that were changed but not yet committed), forces all changes to auxiliary storage, then destroys the open data path for that file. You can use the RCLRSC command to allow a calling program to close a called program's files. (For example, if the called program returns to the calling program without closing its files, the calling program can then close the called program's files.) However, the normal way of closing files in a program is with the high-level language close operation or through the Close File (CLOF) command.

If a job ends normally (for example, a user signs off) and all the files associated with that job were not closed, the system automatically closes all the remaining open files associated with that job, forces all changes to auxiliary storage, and releases all record locks for those files. If a job ends abnormally, the system also closes all files associated with that job, releases all record locks for those files, and forces all changes to auxiliary storage.

Chapter 12. Handling Database File Errors in a Program

Error conditions detected during processing of a database file cause messages to be sent to the program message queue for the program processing the file or cause an inquiry message to be sent to the system operator message queue. In addition, file errors and diagnostic information generally appear to your program as return codes and status information in the file feedback area. (For example, COBOL/400 sets a return code in the file status field, if it is defined in the program.) For more information about handling file errors in your program, see your high-level language guide.

If your programming language allows you to monitor for error messages, you can choose which ones you wish to monitor for. The following messages are some of the error messages you can monitor (see your high-level language guide and the *CL Reference* for a complete list of errors and messages you can monitor):

Message Identifier	**Description**
CPF5001	End of file reached
CPF5006	Record not found
CPF5007	Record deleted
CPF5018	Maximum file size reached
CPF5025	Read attempted past *START or *END
CPF5026	Duplicate key
CPF5027	Record in use by another job
CPF5028	Record key changed
CPF5029	Data mapping error
CPF5030	Partial damage on member
CPF5031	Maximum number of record locks exceeded
CPF5032	Record already allocated to job
CPF5033	Select/omit error
CPF5034	Duplicate key in another member's access path
CPF5040	Omitted record not retrieved
CPF5072	Join value in member changed
CPF5079	Commitment control resource limit exceeded
CPF5084	Duplicate key for uncommitted key
CPF5085	Duplicate key for uncommitted key in another access path
CPF5090	Invalid unique access path prevented add or update
CPF5097	Key mapping error

Note: To display the full description of these messages, use the Display Message Description (DSPMSGD) command.

If you do not monitor for any of these messages, the system handles the error. The system also sets the appropriate error return code in the program. Depending on the error, the system can end the job or send a message to the operator requesting further action.

If a message is sent to your program while processing a data base file member, the position in the file is not lost. It remains at the record it was positioned to before the message was sent, except:

- After an end-of-file condition is reached and a message is sent to the program, the file is positioned at *START or *END.
- After a conversion mapping message on a read operation, the file is positioned to the record containing the data that caused the message.

Part 4. Managing Database Files

The chapters in this part include information on managing database files. This includes: adding new members, changing attributes of existing members, renaming members, or removing them from a database file. Information on member operations unique to physical files including initializing data, clearing data, reorganizing data, and displaying data in a physical file member is also included.

This section includes information on changing database file descriptions and attributes (including the effects of changing fields in file descriptions), changing physical file descriptions and attributes, changing logical file descriptions and attributes, and using database attributes and cross reference information is included.

Another topic covered is displaying information about the database files files such as attributes, descriptions of fields, relationships between the fields, files used by programs, system cross reference files, and information on how to write a command output directly to a database file.

Also included is information to help you plan for recovery of your database files in the event of a system failure. This includes saving and restoring, journaling, using auxiliary storage, and commitment control. This section also includes information on access path recovery which includes rebuilding and journaling access paths.

A section on source files discusses source file concepts and reasons you would use a source file. Information on how to set up a source file, how to enter data into a source file, and ways to use a source file to create another object on the system is included.

Chapter 13. Managing Database Members

Before you perform any input or output operations on a file, the file must have at least one member. As a general rule, database files have only one member, the one created when the file is created. The name of this member is the same as the file name, unless you give it a different name. Because most operations on database files assume that the member being used is the first member in the file, and because most files only have one member, you do not normally have to be concerned with, or specify, member names.

If a file contains more than one member, each member serves as a subset of the data in the file. This allows you to classify data easier. For example, you define an accounts receivable file. You decide that you want to keep data for a year in that file, but you frequently want to process data just one month at a time. For example, you create a physical file with 12 members, one named for each month. Then, you process each month's data separately (by individual member). You can also process several or all members together.

Member Operations Common to All Database Files

The system supplies a way for you to:

- Add new members to an existing file.
- Change some attributes for an existing member (for example, the text describing the member) without having to recreate the member.
- Rename a member.
- Remove the member from the file.

The following section discusses these operations.

Adding Members to Files

You can add members to files in any of these ways:

- Automatically. When a file is created using the Create Physical File (CRTPF) or Create Logical File (CRTLF) commands, the default is to automatically add a member (with the same name as the file) to the newly created file. (The default for the Create Source Physical File [CRTSRCPF] command is *not* to add a member to the newly created file.) You can specify a different member name using the MBR parameter on the create database file commands. If you do not want a member added when the file is created, specify *NONE on the MBR parameter.
- Specifically. After the file is created, you can add a member using the Add Physical File Member (ADDPFM) or Add Logical File Member (ADDLFM) commands.
- Copy File command. Another way to add members to a database file is using the Copy File (CPYF) command. If the member you are copying does not exist in the file being copied to, then the member is added to the file by the CPYF command.

Changing Member Attributes

You can use the Change Physical File Member (CHGPFM) or Change Logical File Member (CHGLFM) command to change certain attributes of a physical or a logical file member. For a physical file member, you can change the following parameters: EXPDATE (the member's expiration date), SHARE (whether the member can be shared within a job), and TEXT (the text description of the member). For a logical file member you can change the SHARE and TEXT parameters.

Note: You can use the Change Physical File (CHGPF) and Change Logical File (CHGLF) commands to change many other file attributes. For example, to change the maximum size allowed for each member in the file, you would use the SIZE parameter on the CHGPF command.

Renaming Members

The Rename Member (RNMM) command changes the name of an existing member in a physical or logical file. The file name is not changed.

Removing Members from Files

The Remove Member (RMVM) command is used to remove the member and its contents. Both the member data and the member itself are removed. After the member is removed, it can no longer be used by the system. This is different from just clearing or deleting the data from the member. If the member still exists, programs can continue to use (for example, add data to) the member.

Physical File Member Operations

The following section describes member operations that are unique to physical file members. Those operations include initializing data, clearing data, reorganizing data, and displaying data in a physical file member.

Initializing Data in a Physical File Member

To use relative record processing in a program, the database file must contain a number of record positions equal to the highest relative record number used in the program. Programs using relative-record-number processing sometimes require that these records be initialized.

You can use the Initialize Physical File Member (INZPFM) command to initialize members with one of two types of records:

- Default records
- Deleted records

You specify which type of record you want using the RECORDS parameter on the Initialize Physical File Member (INZPFM) command.

If you initialize records using default records, the fields in each new record are initialized to the default field values defined when the file was created. If no default field value was defined, then numeric fields are filled with zeros and character fields are filled with blanks. Program-described files have a default value of all blanks.

Note: You cannot initialize a default record if the UNIQUE keyword is specified in DDS for the physical file member or any associated logical file members. Otherwise, you would create a series of duplicate key records.

If the records are initialized to the default records, you can read a record by relative record number and change the data.

If the records were initialized to deleted records, you can change the data by adding a record using a relative record number of one of the deleted records. (You cannot add a record using a relative record number that was not deleted.)

Deleted records cannot be read; they only hold a place in the member. A deleted record can be changed by writing a new record over the deleted record. Refer to "Deleting Database Records" on page 10-10 for more information about processing deleted records.

Clearing Data from Physical File Members

The Clear Physical File Member (CLRPFM) command is used to remove the data from a physical file member. After the clear operation is complete, the member description remains, but the data is gone.

Reorganizing Data in Physical File Members

You can use the Reorganize Physical File Member (RGZPFM) command to:

- Remove deleted records to make the space occupied by them available for more records.
- Reorganize the records of a file in the order in which you normally access them sequentially, thereby minimizing the time required to retrieve records. This is done using the KEYFILE parameter. This may be advantageous for files that are primarily accessed in an order other than arrival sequence. A member can be reorganized using either of the following:
  - Key fields of the physical file
  - Key fields of a logical file based on the physical file
- Reorganize a source file member, insert new source sequence numbers, and reset the source date fields (using the SRCOPT and SRCSEQ parameters on the Reorganize Physical File Member command).

For example, the following Reorganize Physical File Member (RGZPFM) command reorganizes the first member of a physical file using an access path from a logical file:

```
RGZPFM  FILE(DSTPRODLB/ORDHDRP)
        KEYFILE(DSTPRODLB/ORDFILL  ORDFILL)
```

The physical file ORDHDRP has an arrival sequence access path. It was reorganized using the access path in the logical file ORDFILL. Assume the key field is the *Order* field. The following illustrates how the records were arranged.

The following is an example of the original ORDHDRP file. Note that record 3 was deleted before the RGZPFM command was run:

Relative Record Number	**Cust**	**Order**	**Ordate. . .**
1	41394	41882	072480. . .
2	28674	32133	060280. . .
3	deleted	record	
4	56325	38694	062780. . .

The following example shows the ORDHDRP file reorganized using the *Order* field as the key field in ascending sequence:

Relative Record Number	**Cust**	**Order**	**Ordate. . .**
1	28674	32133	060280. . .
2	56325	38694	062780. . .
3	41394	41882	072480. . .

Notes:

1. If a file with an arrival sequence access path is reorganized using a keyed access path, the arrival sequence access path is changed. That is, the records in the file are physically placed in the order of the keyed sequence access path used. By reorganizing the data into a physical sequence that closely matches the keyed access path you are using, you can improve the performance of processing the data sequentially.
2. Reorganizing a file compresses deleted records, which changes subsequent relative record numbers.
3. Because access paths with either the FIFO or LIFO DDS keyword specified depend on the physical sequence of records in the physical file, the sequence of the records with duplicate key fields may change after reorganizing a physical file using a keyed sequence access path.

   Also, because access paths with the FCFO DDS keyword specified are ordered as FIFO, when a reorganization is done, the sequence of the records with duplicate key fields may also change.

If one of the following conditions occur and the Reorganize Physical File Member (RGZPFM) command is running, the records may not be reorganized:

- The system ends abnormally.
- The job containing the RGZPFM command is ended with an *IMMED option.
- The subsystem in which the RGZPFM command is running ends with an *IMMED option.
- The system stops with an *IMMED option.

The status of the member being reorganized depends on how much the system was able to do before the reorganization was ended and what you specified in the SRCOPT parameter. If the SRCOPT parameter was not specified, the member is either completely reorganized or not reorganized at all. You should display the contents of the file, using the Display Physical File Member (DSPPFM) command, to determine if it was reorganized. If the member was not reorganized, issue the Reorganize Physical File Member (RGZPFM) command again.

If the SRCOPT parameter was specified, any of the following could have happened to the member:

- It was completely reorganized. A completion message is sent to your job log indicating the reorganize operation was completely successful.
- It was not reorganized at all. A message is sent to your job log indicating that the reorganize operation was not successful. If this occurs, issue the Reorganize Physical File Member (RGZPFM) command again.
- It was reorganized, but only some of the sequence numbers were changed. A completion message is sent to your job log indicating that the member was reorganized, but all the sequence numbers were not changed. If this occurs, issue the RGZPFM command again with KEYFILE(*NONE) specified.

Displaying Records in a Physical File Member

The Display Physical File Member (DSPPFM) command can be used to display the data in the physical database file members by arrival sequence. The command can be used for:

- Problem analysis
- Debugging
- Record inquiry

You can display source files or data files, regardless if they are keyed or arrival sequence. Records are displayed in arrival sequence, even if the file is a keyed file. You can page through the file, locate a particular record by record number, or shift the display to the right or left to see other parts of the records. You can also press a function key to show either character data or hexadecimal data on the display.

If you have Query installed, you can use the Start Query (STRQRY) command to select and display records, too.

If you have the SQL/400 installed, you can use the Start SQL/400 (STRSQL) command to interactively select and display records.

Chapter 14. Changing Database File Descriptions and Attributes

This chapter describes the things to consider when planning to change the description or attributes of a database file.

Effect of Changing Fields in a File Description

When a program that uses externally described data is compiled, the compiler copies the file descriptions of the files into the compiled program. When you run the program, the system can verify that the record formats the program was compiled with are the same as the record formats currently defined for the file. The default is to do level checking.

The system assigns a unique level identifier for each record format when the file it is associated with is created. The system uses the information in the record format description to determine the level identifier. This information includes the total length of the record format, the record format name, the number and order of fields defined, the data type, the size of the fields, the field names, and the number of decimal positions in the field. Changes to this information in a record format cause the level identifier to change.

The following DDS information has no effect on the level identifier and, therefore, can be changed without recompiling the program that uses the file:

- TEXT keyword
- COLHDG keyword
- CHECK keyword
- EDTCDE keyword
- EDTWRD keyword
- REF keyword
- REFFLD keyword
- CMP, RANGE, and VALUES keywords
- TRNTBL keyword
- REFSHIFT keyword
- DFT keyword
- Join specifications and join keywords
- Key fields
- Access path keywords
- Select/omit fields

Keep in mind that even though changing key fields or select/omit fields will not cause a level check, the change may cause unexpected results in programs using the new access path. For example, changing the key field from the customer number to the customer name changes the order in which the records are retrieved, and may cause unexpected problems in the programs processing the file.

If level checking is specified (or defaulted to), the level identifier of the file to be used is compared to the level identifier of the file in your program when the file is opened. If the identifiers differ, a message is sent to the program to identify the changed condition and the changes may affect your program. You can simply compile your program again so that the changes are included.

An alternative is to display the file description to determine if the changes affect your program. You can use the Display File Field Description (DSPFFD) command to display the description or, if you have SEU, you can display the source file containing the DDS for the file.

The format level identifier defined in the file can be displayed by the Display File Description (DSPFD) command. When you are displaying the level identifier, remember that the record format identifier is compared, rather than the file identifier.

Not every change in a file necessarily affects your program. For example, if you add a field to the end of a file and your program does not use the new field, you do not have to recompile your program. If the changes do not affect your program, you can use the Change Physical File (CHGPF) or the Change Logical File (CHGLF) commands with LVLCHK(*NO) specified to turn off level checking for the file, or you can enter an Override with Database File (OVRDBF) command with LVLCHK(*NO) specified so that you can run your program without level checking.

Keep in mind that level checking is the preferred method of operating. The use of LVLCHK(*YES) is a good database integrity practice. The results produced by LVLCHK(*NO) cannot always be predicted.

Changing a Physical File Description and Attributes

Sometimes, when you make a change to a physical file description and then re-create the file, the level identifier can change. For example, the level identifier will change if you add a field to the file description, or change the length of an existing field. If the level identifier changes, you can compile the programs again that use the physical file. After the programs are recompiled, they will use the new level check identifier.

You can avoid compiling again by creating a logical file that presents data to your programs in the original record format of the physical file. Using this approach, the logical file has the same level check identifier as the physical file before the change.

For example, you decide to add a field to a physical file record format. You can avoid compiling your program again by doing the following:

1. Change the DDS and create a new physical file (FILEB in LIBA) to include the new field:

   ```
   CRTPF FILE(LIBA/FILEB) MBR(*NONE)...
   ```

   FILEB does not have a member. (The old file FILEA is in library LIBA and has one member MBRA.)
2. Copy the member of the old physical file to the new physical file:

   ```
   CPYF    FROMFILE(LIBA/FILEA) TOFILE(LIBA/FILEB)
           FROMMBR(*ALL) TOMBR(*FROMMBR)
           MBROPT(*ADD) FMTOPT(*MAP)
   ```

   The member in the new physical file is automatically named the same as the member in the old physical file because FROMMBR(*ALL) and TOMBR(*FROMMBR) are specified. The FMTOPT parameter specifies to copy (*MAP) the data in the fields by field name.
3. Describe a new logical file (FILEC) that looks like the original physical file (the logical file record format does *not* include the new physical file field). Specify FILEB for the PFILE keyword. (When a level check is done, the level identifier in

the logical file and the level identifier in the program match because FILEA and FILEC have the same format.)

4. Create the new logical file:

   ```
   CRTLF FILE(LIBA/FILEC)...
   ```

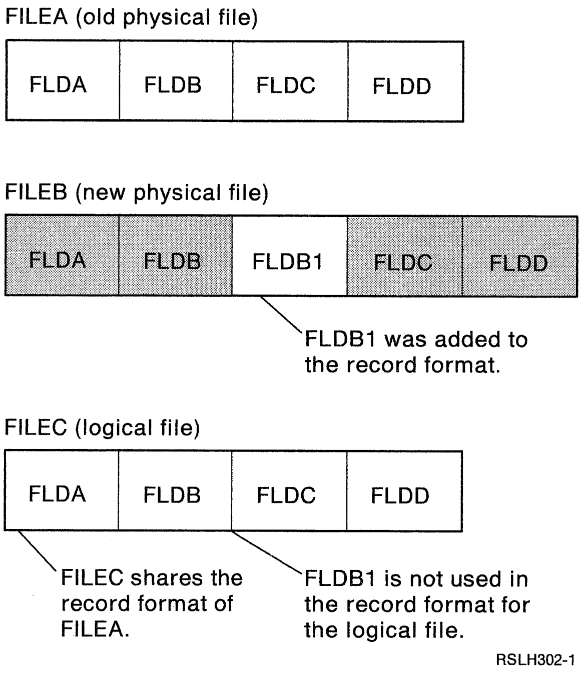
5. You can now do one of the following:
   a. Use an Override with Database File (OVRDBF) command in the appropriate jobs to override the old physical file referred to in the program with the logical file (the OVRDBF command parameters are described in more detail in Chapter 8).

      ```
      OVRDBF FILE(FILEA) TOFILE(LIBA/FILEC)
      ```

   b. Delete the old physical file and rename the logical file to the name of the old physical file so the file name in the program does not have to be overridden.

      ```
      DLTF FILE(LIBA/FILEA)
      RNMOBJ   OBJ(LIBA/FILEC) OBJTYPE(*FILE)
               NEWOBJ(FILEA)
      ```

The following illustrates the relationship of the record formats used in the three files:


FILEA (old physical file)
FLDA
FLDB
FLDC
FLDD
FILEB (new physical file)
FLDA
FLDB
FLDB1
FLDC
FLDD
FLDB1 was added to the record format.
FILEC (logical file)
FLDA
FLDB
FLDC
FLDD
FILEC shares the record format of FILEA.
FLDB1 is not used in the record format for the logical file.
RSLH302-1

When you make changes to a physical file that cause you to create the file again, all logical files referring to it must first be deleted before you can delete and create the new physical file. After the physical file is re-created, you can re-create or restore the logical files referring to it. The following examples show two ways to do this:

Example 1: Create a new physical file with the same name in a different library

1. Create a new physical file with a different record format in a library different from the library the old physical file is in. The name of new file should be the same as the name of the old file. (The old physical file FILEPFC is in library LIBB and has two members, MBRC1 and MBRC2.)

   ```
   CRTPF FILE(NEWLIB/FILEPFC) MAXMBRS(2)...
   ```

2. Copy the members of the old physical file to the new physical file. The members in the new physical file are automatically named the same as the members in the old physical file because TOMBR(*FROMMBR) and FROMMBR(*ALL) are specified.

   ```
   CPYF     FROMFILE(LIBB/FILEPFC) TOFILE(NEWLIB/FILEPFC)
            FROMMBR(*ALL) TOMBR(*FROMMBR)
            FMTOPT(*MAP *DROP) MBROPT(*ADD)
   ```

3. Describe and create a new logical file in a library different from the library the old logical file is in. The name of the new logical file should be the same as the old logical file name. (You can use the FORMAT keyword to use the same record formats as are in the current logical file, if no changes need be made to the record formats.) The old logical file FILELFC is in library LIBB.

   ```
   CRTLF FILE(NEWLIB/FILELFC)
   ```

4. Delete the old logical and physical files.

   ```
   DLTF FILE(LIBB/FILELFC)
   DLTF FILE(LIBB/FILEPFC)
   ```

5. Move the newly created files to the original library by using the following commands:

   ```
   MOVOBJ OBJ(NEWLIB/FILELFC) OBJTYPE(*FILE) TOLIB(LIBB)
   MOVOBJ OBJ(NEWLIB/FILEPFC) OBJTYPE(*FILE) TOLIB(LIBB)
   ```

Example 2: Creating new versions of files in the same libraries

1. Create a new physical file with a different record format in the same library the old physical file is in. The names of the files should be different. (The old physical file FILEPFA is in library LIBA and has two members MBRA1 and MBRA2.)

   ```
   CRTPF FILE(LIBA/FILEPFB) MAXMBRS(2)...
   ```

2. Copy the members of the old physical file to the new physical file.

   ```
   CPYF     FROMFILE(LIBA/FILEPFA) TOFILE(LIBA/FILEPFB)
            FROMMBR(*ALL) TOMBR(*FROMMBR)
            FMTOPT(*MAP *DROP) MBROPT(*REPLACE)
   ```

3. Create a new logical file in the same library as the old logical file is in. The names of the old and new files should be different. (You can use the FORMAT keyword to use the same record formats as are in the current logical file if no changes need be made to the record formats.) The PFILE keyword must refer to the new physical file created in step 1. The old logical file FILELFA is in library LIBA.

   ```
   CRTLF FILE(LIBA/FILELFB)
   ```

4. Delete the old logical and physical files.

   ```
   DLTF FILE(LIBA/FILELFA)
   DLTF FILE(LIBA/FILEPFA)
   ```

5. Rename the new logical file to the name of the old logical file. (If you also decide to rename the physical file, be sure to change the DDS for logical file so that the PFILE keyword refers to the new physical file name.)

   ```
   RNMOBJ(LIBA/FILELFB) OBJTYPE(*FILE) NEWOBJ(FILELFA)
   ```

6. If the logical file member should be renamed, and assuming the default was used on the Create Logical File (CRTLF) command, issue the following command:

   ```
   RNMM FILE(LIBA/FILELFA) MBR(FILELFB) NEWMBR(FILELFA)
   ```

You can use the Change Physical File (CHGPF) command to change some of the attributes of a physical file and its members. For information on these parameters, see the Change Physical File (CHGPF) command in the *CL Reference*.

Changing a Logical File Description and Attributes

As a general rule, when you make changes to a logical file that will cause a change to the level identifier (for example, adding a new field, deleting a field, or changing the length of a field), it is *strongly* recommended that you recompile the programs that use the logical file. Sometimes you can make changes to a file that change the level identifier and which do not require you to recompile your program (for example, adding a field that will not be used by your program to the end of the file). However, in those situations you will be forced to turn off level checking to run your program using the changed file. That is not the preferred method of operating. It increases the chances of incorrect data in the future.

To avoid recompiling, you can keep the current logical file (unchanged) and create a new logical file with the added field. Your program refers to the old file, which still exists.

You can use the Change Logical File (CHGLF) command to change most of the attributes of a logical file and its members that were specified on the Create Logical File (CRTLF) command.

Chapter 15. Using Database Attribute and Cross-Reference Information

The AS/400 integrated database provides file attribute and cross-reference information. Some of the cross-reference information includes:

- The files used in a program
- The files that depend on other files for data or access paths
- File attributes
- The fields defined for a file

Each of the commands described in the following sections can present information on a display, a printout, or write the cross-reference information to a database file that, in turn, can be used by a program or utility (for example, Query) for analysis.

For more information about writing the output to a database file, see "Writing the Output from a Command Directly to a Database File" on page 15-4.

You can retrieve information about a member of a database file for use in your applications with the Retrieve Member Description (RTVMBRD) command. See the section on "Retrieving Member Description Information" in the *CL Programmer's Guide* for an example of how the RTVMBRD command is used in a CL program to retrieve the description of a specific member.

Displaying Information about Database Files

Displaying Attributes for a File

You can use the Display File Description (DSPFD) command to display the file attributes for database files and device files. The information can be displayed, printed, or written to a database output file (OUTFILE). The information supplied by this command includes (parameter values given in parentheses):

- Basic attributes (*BASATR)
- File attributes (*ATR)
- Access path specifications (*ACCPTH, logical and physical files only)
- Select/omit specifications (*SELECT, logical files only)
- Join logical file specifications (*JOIN, join logical files only)
- Alternative collating sequence specifications (*SEQ, physical and logical files only)
- Record format specifications (*RCDFMT)
- Member attributes (*MBR, physical and logical files only)
- Spooling attributes (*SPOOL, printer and diskette files only)
- Member lists (*MBRLIST, physical and logical files only)

Displaying the Descriptions of the Fields in a File

You can use the Display File Field Description (DSPFFD) command to display field information for both database and device files. The information can be displayed, printed, or written to a database output file (OUTFILE).

Displaying the Relationships between Files on the System

You can use the Display Database Relations (DSPDBR) command to display the following information about the organization of your database:

- A list of database files (physical and logical) that use a specific record format.
- A list of database files (physical and logical) that depend on the specified file for data sharing
- A list of members (physical and logical) that depend on the specified member for sharing data or sharing an access path.

This information can be displayed, printed, or written to a database output file (OUTFILE).

For example, to display a list of all database files associated with physical file ORDHDRP, with the record format ORDHDR, type the following DSPDBR command:

```
DSPDBR FILE(DSTPRODLB/ORDHDRP) RCDFMT(ORDHDR)
```

Note: See the DSPDBR command description in the *CL Reference* for details of this display.

This display presents header information when a record format name is specified on the RCDFMT parameter, and presents information about which files are using the specified record format.

If a member name is specified on the MBR parameter of the DSPDBR command, the dependent members are shown.

If the Display Database Relations (DSPDBR) command is specified with the default MBR(*NONE) parameter value, the dependent data files are shown. To display the shared access paths, you must specify a member name.

The Display Database Relations (DSPDBR) command output identifies the type of sharing involved. If the results of the command are displayed, the name of the type of sharing is displayed. If the results of the command are written to a database file, the code for the type of sharing (shown below) is placed in the *WHTYPE* field in the records of the output file.

Type	Code	Description
Data	D	The file or member is dependent on the data in a member of another file.
Access path	I	(Access path sharing) The file member is sharing an access path.
Acc path owner	O	(Access path owner) If an access path is shared, one of the file members is considered the owner. The owner of the access path is charged with the storage used for the access path. If the member displayed is designated the owner, one or more file members are designated with an I for access path sharing.
SQL View	S	The SQL view or member is dependent upon another SQL view.

Displaying the Files Used by Programs

You can use the Display Program Reference (DSPPGMREF) command to determine which files, data areas, and other programs are used by a program. This information is available for compiled programs only.

The information can be displayed, printed, or written to a database output file (OUTFILE).

When a program is created, the information about certain objects used in the program is stored. This information is then available for use with the Display Program References (DSPPGMREF) command.

The following chart shows the objects for which the high-level languages and utilities save information:

Language or Utility	**Files**	**Programs**	**Data Areas**	**See Notes**
BASIC	Yes	Yes	No	1
C	No	No	N/A	
CL	Yes	Yes	Yes	2
COBOL/400	Yes	Yes	No	3
CSP	Yes	Yes	No	4
DFU	Yes	N/A	N/A	
Pascal	No	No	N/A	
PL/I	Yes	Yes	N/A	3
RPG/400	Yes	No	Yes	5
SQL/400	Yes	N/A	N/A	

Notes:

1. Externally described file references, programs, and data areas are stored.
2. All system commands that refer to files, programs, or data areas specify in the command definition that the information should be stored when the command is compiled in a CL program. If a variable is used, the name of the variable is used as the object name (for example, &FILE). If an expression is used, the name of the object is stored as *EXPR. User-defined commands can also store the information for files, programs, or data areas specified on the command. See the description of the FILE, PGM, and DTAARA parameters on the PARM or ELEM command statements in the *CL Programmer's Guide*.
3. The program name is stored only when a literal is used for the program name (this is a static call, for example, CALL 'PGM1'), not when a COBOL/400 identifier is used for the program name (this is a dynamic call, for example, CALL PGM1).
4. CSP programs also save information for an object of type *MSGF, *CSPMAP, and *CSPTBL.
5. The use of the local data area is not stored.

The stored file information contains an entry (a number) for the type of use. In the database file output of the Display Program References (DSPPGMREF) command (built when using the OUTFILE parameter), this is specified as:

Code	**Meaning**
1	Input
2	Output
3	Input and Output
4	Update
8	Unspecified

Combinations of codes are also used. For example, a file coded as a 7 would be used for input, output, and update.

Displaying the System Cross-Reference Files

The system manages two database files that contain basic file attribute information (QSYS/QADBXREF) and cross-reference information (QSYS/QADBFDEP) about all the database files on the system (except those database files that are in the QTEMP library). You can use these files to determine basic attribute and file requirements. To display the fields contained in these files, use the Display File Field Description (DSPFFD) command.

Note: Normally, the authority to use these files is restricted to the security officer.

Writing the Output from a Command Directly to a Database File

You can store the output from many CL commands in an output physical file by specifying the OUTFILE parameter on the command.

You can use the output files in programs or utilities (for example, Query) for data analysis. For example, you can send the output of the Display Program References (DSPPGMREF) command to a physical file, then query that file to determine which programs use a specific file.

The physical files are created for you when you specify the OUTFILE parameter on the commands. Initially, the files are created with private authority; only the owner (the person who ran the command) can use it. However, the owner can authorize other users to these files as you would for any other database file.

The system supplies model files that identify the record format for each command that can specify the OUTFILE parameter. If you specify a file name on the OUTFILE parameter for a file that does not already exist, the system creates the file using the same record format as the model files. If you specify a file name for an existing output file, the system checks to see if the record format is the same record format as the model file. If the record formats do not match, the system sends a message to the job and the command does not complete.

Note: You must use your own files for output files, rather than specifying the system-supplied model files on the OUTFILE parameter.

The following is a list of commands that allow output files, and the model files supplied for those commands:

Command	Description	Model File	Format Name
DSPAUTHLR	Display Authority Holder	QADSHLR	QSYDSHLR
DSPAUTL	Display Authority List	QAOBJAUT	QSYDSAUTH
DSPAUTLOBJ	Display Authority List Objects	QADALO	QSYDALO
DSPDBR	Display Database Relations	QADSPDBR	QWHDRDBR
DSPDIR[1]	Display Directory		
DSPDSTL[1]	Display Distribution List		
DSPFD[2]	Display File Description		
DSPFFD	Display File Field Description	QADSPFFD	QWHDRFFD
DSPFLR[1]	Display Folder		
DSPJRN[3]	Display Journal	QADSPJRN	QJORDJE
DSPOBJAUT	Display Object Authority	QAOBJAUT	QSYDSAUTH
DSPOBJD	Display Object Description	QADSPOBJ	QLIDOBJD
DSPPGMADP	Display Programs That Adopt	QADPGMMAD	QADPGMAD
DSPPGMREF	Display Program References	QADSPPGM	QWHDRPPR
DSPPRB[5]	Display Problem		
DSPPTF	Display PTF	QADSPPTF	QSCPTF
PRTDOC[1]	Print Document		
PRTERRLOG	Print Error Log	QAPRTELG	QSCELOG
QRYDOCLIB[1]	Query Document Library		
QRYDST[1]	Query Distribution		
RCVDST[1]	Receive Distribution		
RTVDOC[1]	Retrieve Document		
RUNQRY[4]	Run Query		
STRCPYSCN	Start Screen Copy	QASCCPY	OSCCPY1
TRCJOB	Trace Job	QATRCJOB	QSCTJTRC
WRKNETF	Work with Network Files	QANFDNTF	QNFDNTF

[1] See *Programming: Office Services Concepts and Programmer's Guide*, SC21-9758 for more details.

[2] See "Output File for the Display File Description Command" on page 15-6.

[3] Additional model files are supplied for specific types of journal entries. See "Output Files for the Display Journal Command" on page 15-7 for more information.

[4] No model file is supplied.

[5] See "Output Files for the Display Problem Command" on page 15-8 for more information.

Note: All system-supplied model files are located in the QSYS library.

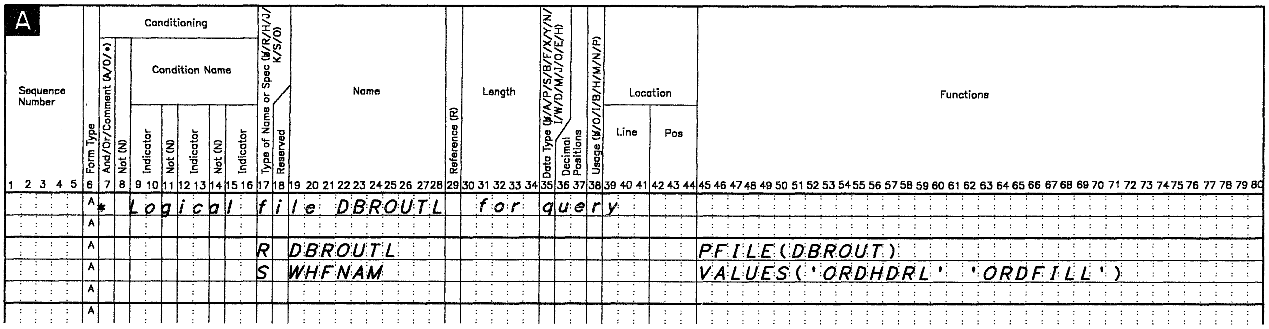
You can display the fields contained in the record formats of the system-supplied model files using the Display File Field Descriptions (DSPFFD) command.

Example of Using a Command Output File

The following example uses the Display Program References (DSPPGMREF) command to collect information for all compiled programs in all libraries, and place the output in a database file named DBROUT:

```
DSPPGMREF  PGM(*ALL/*ALL)  OUTPUT(*OUTFILE)  OUTFILE(DSTPRODLB/DBROUT)
```

You can use Query to process the output file. Another way to process the output file is to create a logical file to select information from the file. The following is the DDS for such a logical file. Records are selected based on the file name.


A
Conditioning
Condition Name
Sequence Number
Form Type
And/Or/Comment (A/O/*)
Not (N)
Indicator
Type of Name or Spec (b/R/H/J/K/S/O)
Reserved
Name
Reference (R)
Length
Data Type (b/A/P/S/B/F/X/Y/N/I/W/D/M/J/O/E/H)
Decimal Positions
Usage (b/O/I/B/H/M/N/P)
Location
Line
Pos
Functions

```
     A*  Logical file DBROUTL for query
     A
     A          R DBROUTL                   PFILE(DBROUT)
     A          S WHFNAM                    VALUES('ORDHDRL' 'ORDFILL')
     A
     A
```

RSLH301-1

Output File for the Display File Description Command

The Display File Description (DSPFD) command provides unique output files, depending on the parameters specified. The model files for the DSPFD command are as follows:

Type	File Attribute	Model File Name	Record Format Name
*ACCPTH		QAFDACCP	QWHFDACP
*BASATR		QAFDBASI	QWHFDBAS
*SEQ		QAFDCSEQ	QWHFDSEQ
*JOIN		QAFDJOIN	QWHFDJN
*MBR		QAFDMBR	QWHFDMBR
*MBRLIST		QAFDMBRL	QWHFDML
*RCDFMT		QAFDRFMT	QWHFDFMT
*SELECT		QAFDSELO	QWHFDSO
*SPOOL		QAFDSPOL	QWHFDSPL
*ATR	*BSCF	QAFDBSC	QWHFDBSC
*ATR	*CMNF	QAFDCMN	QWHFDCMN
*ATR	*DDMF	QAFDDDM	QWHFDDDM
*ATR	*DKTF	QAFDDKT	QWHFDDKT
*ATR	*DSPF	QAFDDSP	QWHFDDSP
*ATR	*ICFF	QAFDICF	QWHFDICF
*ATR	*LF	QAFDLGL	QWHFDLGL
*ATR	*PF	QAFDPHY	QWHFDPHY
*ATR	*PRTF	QAFDPRT	QWHFDPRT
*ATR	*SAVF	QAFDSAV	QWHFDSAV
*ATR	*TAPF	QAFDTAP	QWHFDTAP

Note: All system-supplied model files are in the QSYS library.

To collect access path information about all files in the LIBA library, you could specify:

```
DSPFD   FILE(LIBA/*ALL)  TYPE(*ACCPTH)  OUTPUT(*OUTFILE) +
        OUTFILE(LIBB/ABC)
```

The file ABC is created in library LIBB and is externally described with the same field descriptions as in the system-supplied file QSYS/QAFDACCP. The ABC file then contains a record for each key field in each file found in library LIBA that has an access path.

If the Display File Description (DSPFD) command is coded as:

```
DSPFD   FILE(LIBX/*ALL)  TYPE(*ATR)  OUTPUT(*OUTFILE) +
        FILEATR(*PF)  OUTFILE(LIBB/DEF)
```

the file DEF is created in library LIBB and is externally described with the same field descriptions as exist in QSYS/QAFDPHY. The DEF file then contains a record for each physical file found in library LIBX.

You can display the field names of each model file supplied by IBM using the DSPFFD command. For example, to display the field description for the access path model file (*ACCPTH specified on the TYPE parameter), specify the following:

```
DSPFFD   QSYS/QAFDACCP
```

Output Files for the Display Journal Command

The following is a list of Display Journal (DSPJRN) command model output files supplied on the system.

Description	Model File	Record Format
Standard model file	QADSPJRN	QJORDJE
Job accounting	QAJBACG	QWTJAJBE
Print accounting	QAPTACG	QSPJAPTE
SNADS configuration	QAZDCFLG	QZDCFLOG
SNADS error log	QAZDERLG	QZDERLOG
SNADS logging	QAZDJRNL	QZDLGLOG
SNADS routing	QAZDRTLG	QZDRTLOG
SNADS alert routing	QAZDALLG	QZALLLOG
DSNX logging	QADXJRNL	QDXLGLOG
DSNX error logging	QADXERLG	QDXERLOG

Communications Trace Service Function

You can access the Communications Trace service function using CL commands. Therefore, you do not need to access System Service Tools (SST) to debug your program. Output file support is provided via the following CL commands:

- Start Communications Trace (STRCMNTRC) starts the trace to run continuously until explicitly ended.
- End Communications Trace (ENDCMNTRC) ends a continuously running trace.
- Print Communications Trace (PRTCMNTRC) prints the trace data at a later time than the time the trace was ended. The trace data can be sent to an output file also.

Output Files for the Display Problem Command

The following is a list of Display Problem (DSPPRB) command model output files supplied on the system. The command provides unique output files depending on the type specified below.

Type	Model File	Record Format	Description
*BASIC	QASXPBOF	QSXPBOF	Basic problem data record. This includes problem type, status machine type/model/serial number, product ID, contact information, and tracking data.
*CAUSE	QASXCAOF	QSXCAOF	Point of failure, isolation, or answer FRU records. Answer FRUs will be used if available, if not isolation FRUs will be used if available, if not point of failure FRUs will be used.
*FIX	QASXFXOF	QSXFXOF	PTF fix records.
*USRTXT	QASXTXOF	QSXTXOF	User entered text (note records).

The records in all four output files have a problem identifier so that the cause, fix, and user text information can be correlated with the basic problem data. Only one type of data may be written to a particular output file. The cause, fix, and user text output files may have multiple records for a particular problem. See the *CL Reference* for more information on the DSPPRB command.

Chapter 16. Planning Database Recovery

This chapter discusses several things to consider when you are planning for database recovery. For detailed information about the AS/400 recovery functions, see the *Backup and Recovery Guide*.

Database Save and Restore Considerations

It is very important that you periodically save your files (and other objects). Saving your files (and other objects) is fundamental to protecting your data. This section describes the AS/400 save and restore functions as they apply to database files.

Save and Restore Commands

The AS/400 system has a full set of commands to save and restore your database files. The save commands save database files on tape or diskette. In addition, database files can be saved in a disk-resident save file, then later written to tape or diskette. The save and restore commands used to save and restore files include:

- The Save Library (SAVLIB) command to save one or more libraries
- The Save Object (SAVOBJ) command to save one or more objects (including database files and members)
- The Save Changed Object (SAVCHGOBJ) command to save any objects that have changed since either the last time the library was saved or from a specified date
- The Save Save File Data (SAVSAVFDTA) command to save the contents of a save file
- The Restore Library (RSTLIB) command to restore a library
- The Restore Object (RSTOBJ) command to restore one or more objects (including database files and members)

Saving and Restoring Database File Members

The save and restore commands can be used to save and restore your database files. You also have an option on the SAVOBJ and RSTOBJ commands to save and restore one or more database file members instead of the complete file.

Saving and Restoring Access Paths

Restoring an access path can be faster than rebuilding it. For example, assume a logical file is built over a physical file containing 500,000 records and you have determined (through the Display Object Description [DSPOBJD] command) that the size of the logical file is about 15 megabytes. In this example, assume it would take about 50 minutes to rebuild the access path for the logical file compared to about 1 minute to restore the same access path from a tape. (This assumes that the system can build approximately 10,000 index entries per minute.)

After restoring the access path, the file may need to be brought up-to-date by applying the latest journal changes (depending on whether journaling is active). Normally, the system can apply approximately 80,000 to 100,000 journal entries per hour. (This assumes that each of the physical files to which entries are being applied has only one access path built over it.) Even with this additional recovery time, you will usually find it is faster to restore access paths rather than to rebuild them.

Saving and Restoring Data Using a Disk-Resident Save File

Save files are disk resident files that can be the target of a save operation or the source of a restore operation. Save files allow unattended save operations. That is, an operator does not need to load tapes or diskettes when saving to a save file. However, it is still important to use the SAVSAVFDTA command to periodically save the save file data on tape or diskette. The tapes or diskettes should periodically be removed from the site. Storing a copy of your save tapes or diskettes off the site is important to help recover from a site disaster.

Save files can be used when you run the following commands:

- The Save Library (SAVLIB) command (except when saving multiple libraries)
- The Save Object (SAVOBJ) command
- The Save Changed Object (SAVCHGOBJ) command
- The Restore Library (RSTLIB) command
- The Restore Object (RSTOBJ) command

Database Data Recovery

The AS/400 system has integrated recovery functions and commands to help recover data in a database file. One function discussed in this section is journal management. Journal management allows you to record all the data changes occurring to one or more database files. You can then use the journal for recovery.

Journal Management

You should seriously consider using journal management. If a database file is destroyed or becomes unusable and you are using journaling, you can reconstruct most of the activity for the file. Optionally, the journal allows you to remove changes made to the file.

Journaling can be started or ended very quickly. It requires no additional programming or changes to existing programs.

When a change is made to a file and you are using journaling, the system records the change in a journal receiver and writes the receiver to auxiliary storage before it is recorded in the file. Therefore, the journal receiver always has the latest database information. Activity for a file is recorded regardless of the type of program, user, or job that made the change, or the logical file through which the change was made.

Journal receiver entries record activity for a specific record (record added, updated or deleted), or for the file as a whole (file opened, file member saved, and so on). Each entry includes additional bytes of control information identifying the source of the activity (including user, job, program, time, and date). For changes that affect a single record, record images are included following the control information. The image of the record after a change is made is always included. Optionally, the record image before the change is made can also be included. You control whether to record both before and after record images or just after record images by specifying the IMAGES parameter on the Start Journaling Physical File (STRJRNPF) command.

All journaled database files are automatically synchronized with the journal when the system is started (IPL time). If the system ended abnormally, some database changes may be in the journal, but not yet reflected in the database itself. If that is the case, the system automatically updates the database from the journal to bring the database files up to date.

Journaling can make saving database files easier and faster. For example, instead of saving an entire file everyday, you can simply save the journal receiver that contains the changes to that file. You might still save the entire file on a weekly basis. This method can reduce the amount of time it takes to perform your daily save operations.

The Display Journal (DSPJRN) command, can be used to convert journal receiver entries to a database file. Such a file can be used for activity reports, audit trails, security, and program debugging.

Because the journal supplies many useful functions, not the least of which is recovering data, journal management ought to be considered a key part of your recovery strategy.

Writing Data to Auxiliary Storage

The force-write ratio (FRCRATIO) parameter on the create file and override database file commands can be used to force data to be physically written to auxiliary storage. A force-write ratio of one causes every add, update, and delete request to be immediately written to auxiliary storage for the file in question. However, choosing this option can reduce system performance. Therefore, saving your files and journaling your files should be considered the primary methods for protecting database files.

Transaction Recovery

The AS/400 system has an integrated transaction recovery function called commitment control. Commitment control is an extension of the journal function on the system. It provides you additional assistance in recovering data and restarting your programs. Commitment control can ensure that complex application transactions are logically synchronized even if the job or system ends.

Transactions can be classified as follows:

- Inquiries in which no file changes occur.
- Simple transactions in which one file is changed each time you press the Enter key.
- Complex transactions in which two or more files are changed each time you press the Enter key.
- Complex transactions in which one or more files are changed each time you press the Enter key, but these changes represent only part of a logical group of transactions.

If the system or job ends abnormally, journaling alone can ensure that, at most, only the very last record change is lost. However, if the system or job ends abnormally during a complex transaction (where more than one file may be changed), the files can reflect an incomplete logical transaction. For example, the job may have updated a record in file A, but before it had a chance to update a corresponding record in file B the job ended abnormally. In this case, the logical transaction consisted of two updates, but only one update completed before the job ended abnormally.

Recovering a complex application requires detailed application knowledge. Programs cannot simply be restarted. Adjustments may have to be made with an application program or data file utility to reverse the files to just before the last complex transaction began. This task becomes more complex if multiple users were accessing the files at the same time.

Commitment control helps solve these problems. Under commitment control, the records used during a complex transaction are locked from other users. This ensures that other users do not use the records until the transaction is complete. At the end of the transaction, the program issues the commit operation, freeing the records. However, should the system or job end abnormally before the commit operation is performed, all record changes for that job since the last time a commit operation occurred are rolled back. Any affected records that are still locked are then unlocked. In other words, database changes are rolled back to a clean transaction boundary.

The rollback operation can also occur under your control. Assume that in an order entry application, the application program runs the commit operation at the end of each order. In the middle of an order, the operator can signal the program to do a rollback operation. All file changes will be rolled back to the beginning of the order.

The commit and roll back operations are available in several AS/400 programming languages including RPG/400, COBOL/400, PL/I, SQL/400, and the system control language.

An optional feature of commitment control is the use of a notify object. The notify object is a file, data area, or message queue. When a rollback operation occurs, information specified by the program is automatically sent to the notify object. This information can be used by an operator or application programs to start the application from the last successful transaction boundary.

Commitment control can also be used in a batch environment. Just as it provides assistance in interactive transaction recovery, commitment control can help in batch job recovery.

Access Path Recovery

The system ensures the integrity of an access path before you can use it. If the system determines that the access path is unusable, the system attempts to recover it. You can control when an access path will be recovered. See "Controlling When Access Paths Are Rebuilt" on page 16-7 for more information.

Access path recovery can be time consuming, especially if you have large access paths or many access paths to be rebuilt. The system can assist you in reducing this recovery time.

Rebuilding Access Paths

Rebuilding a database access path takes approximately one minute for every 10,000 records.

Note: This estimate should be used until actual times for your system can be calculated.

The following factors affect this time estimate (listed in general order of significance):

- Storage pool size. The size of the storage pool used to rebuild the access path is a very important factor. You can improve the rebuild time by running the job in a larger storage pool.
- The system model. The speed of the processing unit is a key factor in the time needed to rebuild an access path.
- Key length. A large key length will slow rebuilding the access path because more key information must be constructed and stored in the access path.
- Select/omit values. Select/omit processing will slow the rebuilding of an access path because each record must be compared to see if it meets the select/omit values.
- Record length. A large record length will slow the rebuilding of an access path because more data is looked at.
- Storage device containing the data. The relative speed of the storage device containing the actual data and the device where the access path is stored has an effect on the time needed to rebuild an access path.
- The order of the records in the file. The system tries to rebuild an access path so that it can find information quickly when using that access path. The order of the records in a file has a small impact on how fast the system can build the access path while trying to maintain an efficient access path.

All of the preceding factors must be considered when estimating the amount of time to rebuild an access path.

Journaling Access Paths

Journaling access paths can significantly reduce recovery time by reducing the number of access paths that need to be rebuilt after an abnormal system end.

When you journal database files, images of changes to the records in the file are recorded in the journal. These record images are used to recover the file should the system end abnormally. However, after an abnormal end, the system may find that access paths built over the file are not synchronized with the data in the file. If an access path and its data are not synchronized, the system must rebuild the access path to ensure that the two are synchronized and usable.

When access paths are journaled, the system records images of the access path in the journal to provide known synchronization points between the access path and its data. By having that information in the journal, the system can recover both the data files and the access paths, and ensure that the two are synchronized. In such cases, the lengthy time to rebuild the access paths can be avoided.

In addition, journaling access paths works with other recovery functions on the system. For example, the system has a number of options to help reduce the time required to recover from the failure and replacement of a disk unit. These options include user auxiliary storage pools and checksum protection. While these options reduce the chances that the entire system must be reloaded because of the disk failure, they do not change the fact that access paths may need to be rebuilt when the system is started following replacement of the failed disk. By using access path journaling and some of the recovery options discussed previously, you can reduce your chances of having to reload the entire system and having to rebuild access paths.

Journaling access paths is easy to start. The Start Journal Access Path (STRJRNAP) command is used to start journaling the access path for the specified file. You can journal access paths that have a maintenance attribute of immediate (*IMMED) or delayed (*DLY). Once journaling is started, the system continues to protect the access path until the access path is deleted or you run the End Journaling Access Path (ENDJRNAP) command for that access path.

Before journaling an access path, you must start journaling for the physical files associated with the access path. In addition, you must use the same journal for the access path and its associated physical files.

Access path journaling is designed to minimize additional output operations. For example, the system will write the journal data for the changed record and the changed access path in the same output operation. However, you should seriously consider isolating your journal receivers in user auxiliary storage pools when you start journaling your access paths. Placing journal receivers in their own user auxiliary storage pool will provide the best journaling performance, while helping to protect them from a disk failure.

Other Methods to Avoid Rebuilding Access Paths

If you do not journal your access paths, then you might consider some other system functions that can help you reduce the chances of having to rebuild access paths.

The method used by the system to determine if an access path needs to be rebuilt is a file synchronization indicator. Normally the synchronization indicator is on, indicating that the access path and its associated data are synchronized. When a job changes a file that affects an access path, the system turns off the synchronization indicator in the file. If the system ends abnormally, it must rebuild any access path whose file has its synchronization indicator off.

To reduce the number of access paths that must be rebuilt, you need a way to periodically synchronize the data with its access path. There are several methods to synchronize a file with its access path:

- Full file close. The last full (that is, not shared) system-wide close performed against a file will synchronize the access path and the data.
- Force access path. The force-access-path (FRCACCPTH) parameter can be specified on the create or change file commands.
- Force write ratio of 2 or greater. The force-write-ratio (FRCRATIO) parameter can be specified on the create, change, or override database file commands.
- Force end of data. The file's data and its access path can be synchronized by running the force-end-of-data operation in your program. (Some high-level languages do not have a force-end-of-data operation. See your high-level language guide for further details.)

Keep in mind that while the data and its access path are synchronized after performing one of the methods mentioned previously, the next change to the data in the file can cause the synchronization indicator to be turned off again. It is also important to note that each of the methods can be costly in terms of performance; therefore, they should be used with caution. Consider journaling access paths, along with saving access paths, as the primary means of protecting access paths.

Controlling When Access Paths Are Rebuilt

If the system ends abnormally, during the next IPL the system automatically lists those files requiring access path recovery. You can decide whether to rebuild the access path:

- During the IPL
- After the IPL
- When the file is first used

The access path recovery value for a file is determined by the value you specified on the RECOVER parameter on the create and change file commands. You can override this value on the IPL Access Path Recovery display.

For those files that do not have to be rebuilt during IPL time, specify that the file can be rebuilt at a later time. This should help reduce the number of files that need to be rebuilt at IPL, allowing the system to complete its IPL much faster.

For example, you can specify that all files that must have their access path rebuilt should rebuild the access path when the file is first used. In this case, no access paths are rebuilt at IPL. You can control the order in which the access paths are rebuilt by running only those programs that use the files you want rebuilt first. This method will shorten the IPL time (because there are no access paths to rebuild during the IPL) and get the first several applications available faster. Note, however, that the overall time to rebuild all the access paths will probably be longer (because there may be other work running when the access path is being rebuilt and there may be less main storage available to rebuild the access path).

Designing Files to Reduce Access Path Rebuilding Time

File design can also help reduce access path recovery time. For example, you might divide a large master file into a history file and a transaction file. The transaction file would be used for adding new data, the history file would be used for inquiry only. On a daily basis, you might merge the transaction data into the history file, then clear the transaction file for the next day's data. With this design, the time to rebuild access paths could be shortened. That is, if the system abnormally ended during the day the access path to the smaller transaction file might need to be rebuilt. However, the access path to the large history file, being read-only for most of the day, would rarely be out of synchronization with its data, thereby significantly reducing the chances that it would have to be rebuilt.

Consider the trade-off between using a file design to reduce access path rebuilding time and using system-supplied functions like access path journaling. The file design described above may require a more complex application design. After evaluating your situation, you may decide to use system-supplied functions like access path journaling rather than design more complex applications.

Database Recovery after an Abnormal System End

After an abnormal system end, the system proceeds through several automatic recovery steps. This includes such things as: rebuilding the system directory and synchronizing the journal to the files being journaled. The system performs recovery operations during IPL and after IPL.

Database File Recovery during the IPL

During IPL, nothing but the recovery function is active on the system. During IPL, database file recovery consists of the following:

- The following functions that were in progress when the system ended are completed:

  Delete file
  Remove member
  Rename member
  Move object
  Rename object
  Change object owner
  Change file
  Change member
  Grant authority
  Revoke authority
  Start journaling physical file
  Start journaling access path
  End journaling physical file
  End journaling access path
  Recover SQL/400 views
- The following functions that were in progress when the system ended are backed out (you must run them again):

  Create file
  Add member
- If the operator is doing the IPL, the Override Access Path Recovery display appears on the operator's display. The display gives the operator the ability to override the RECOVER option for the files with immediate or delayed maintenance which were in use. If no files were in use, or the IPL is unattended, no displays appear.
- Access paths that have immediate or delayed maintenance, that were in use, and that are specified for recovery during IPL (from the RECOVER option or overridden from the Override Access Path Recovery display) are rebuilt and a message is sent to the system operator. Files with journaled access paths that were in use are not displayed on the Override Access Path Recovery display. They are automatically recovered and a message is sent to the system operator.
- For unattended IPLs, if the system value QDBRCVYWT is 1 (wait), files that were in use that are specified for recovery after IPL are treated as files specified for recovery during IPL. See the *Work Management Guide* for more information on the system value QDBRCVYWT.

- Messages about the following information are sent to the history log:
  - The success or failure of the previously listed items
  - The physical file members that were open when the system ended abnormally and the last active relative record number in each member
  - The files that were open but whose access paths were not rebuilt because RECOVER(*NO) on the create or change file commands was specified for the files
  - The physical file members that could not be synchronized with the journal
  - That IPL database recovery has completed
- The files whose access paths are to be rebuilt after IPL are allocated (see Figure 16-1 on page 16-10).

Database File Recovery after the IPL

This recovery step is run after the IPL is complete. Interactive users may be active and batch jobs may be running with this step of database recovery. However, a file requiring recovery during this step is locked and not available to other jobs until recovery is completed on the access path.

Recovery after the IPL consists of the following:

- The access paths for immediate or delayed maintenance files which specify recovery after IPL, are rebuilt (see Figure 16-1 on page 16-10). A message is sent to the system operator indicating the number of access paths that are to be rebuilt. As each access path is rebuilt, its associated file is unlocked. (If a program tries to use a file that is being rebuilt, the program must wait until the rebuilding is complete.)
- Messages are sent to the system history log indicating the success or failure of the rebuild operation and that database recovery is complete.

Note: If you are not using journaling for a file and you were adding records to a member when the system ended abnormally, the records are in sequence after recovery. That is, if you added nine records and record 9 is there after recovery, records 1 through 8 are also there. However, if you were updating or deleting records when the system ended abnormally and you were not journaling, not all records you were updating are necessarily updated after recovery and not all records you were deleting are necessarily deleted after recovery.

Database File Recovery Options Table

The table below summarizes the file recovery options:

Figure 16-1. Relationship of Access Path, Maintenance, and Recovery

	RECOVER Parameter Specified		
Access Path/ Maintenance	***NO**	***AFTIPL**	***IPL**
Keyed sequence access path/ immediate or delayed maintenance	• No database recovery at IPL • File available immediately • Access path rebuilt first time file opened	• Access path rebuilt after IPL • File allocated • For logical files, the underlying physical data spaces are locked for shared read and are available immediately	• Access path rebuilt during IPL
Keyed sequence access path rebuild maintenance	• No database recovery at IPL • File available immediately • Access path rebuilt first time file opened	• Not applicable; no recovery is done for rebuild maintenance	• Not applicable; no recovery is done for rebuild maintenance
Arrival sequence access path	• No database recovery at IPL • File available immediately	• Not applicable; no recovery is done for an arrival sequence access path	• Not applicable; no recovery is done for an arrival sequence access path

Chapter 17. Using Source Files

This chapter describes source files. Source file concepts are discussed, along with why you would use a source file. In addition, this chapter describes how to set up a source file, how to enter data into a source file, and how to use that source file to create another object (for example, a file or program) on the system.

Source File Concepts

A source file is used when a command alone cannot supply sufficient information for creating an object. It contains input (source) data needed to create some types of objects. For example, to create a control language (CL) program, you must use a source file containing source statements, which are in the form of commands. To create a logical file, you must use a source file containing DDS.

To create the following objects, source files are required:

- High-level language programs
- Control language programs
- Logical files
- Intersystem communications function (ICF) files
- Commands
- Translate tables

To create the following objects, source files can be used, but are *not* required:

- Physical files
- Display files
- Printer files

A source file can be a database file, diskette file, tape file, or inline data file. (An inline data file is included as part of a job.) A source database file is simply another type of database file. You can use a source database file like you would any other database file on the system.

Creating a Source File

To create a source file, you can use the Create Source Physical File (CRTSRCPF), Create Physical File (CRTPF), or Create Logical File (CRTLF) command. Normally, you will use the CRTSRCPF command to create a source file, because many of the parameters default to values that you usually want for a source file. (If you want to use DDS to create a source file, then you would use the CRTPF or CRTLF command.)

The CRTSRCPF command creates a physical file, but with attributes appropriate for source physical files. For example, the default record length for a source file is 92 (80 for the source data field, 6 for the source sequence number field, and 6 for the source date field).

The following example shows how to create a source file using the CRTSRCPF command and using the command defaults:

```
CRTSRCPF FILE(QGPL/FRSOURCE) TEXT('Source file')
```

IBM-Supplied Source Files

For your convenience, OS/400 and other licensed programs provide a database source file for each type of source. These source files are:

File Name	Library Name	Used to Create
QBASSRC	QGPL	BASIC programs
QCLSRC	QGPL	CL programs
QCMDSRC	QGPL	Command definition statements
QDDSSRC	QGPL	Files
QFMTSRC	QGPL	Sort source
QLBLSRC	QGPL	COBOL/400 programs
QCBLSRC	QGPL	System/38 compatible COBOL programs
QS36SRC	#LIBRARY	System/36 compatible COBOL programs
QAPLISRC	QPLI	PL/I programs
QPLISRC	QGPL	PL/I programs
QRPGSRC	QRPG	RPG/400 programs
QARPGSRC	QRPG38	System/38 environment RPG
QRPG2SRC	#RPGLIB	System/36 compatible RPG II
QS36SRC	#LIBRARY	System/36 compatible RPG II (after install)
QPASSRC	QPAS	Pascal programs
QTBLSRC	QGPL	Translation tables
QTXTSRC	QPDA	Text

You can either add your source members to these files or create your own source files. Normally, you will want to create your own source files using the same names as the IBM-supplied files, but in different libraries (IBM-supplied files may get overlaid when a new release of the system is installed). The IBM-supplied source files are created with the file names used for the corresponding create command (for example, the CRTCLPGM command uses the QCLSRC file name as the default). Additionally, the IBM-supplied programmer menu uses the same default names. If you create your own source files, do not place them in the same library as the IBM-supplied source files. (If you use the same file names as the IBM-supplied names, you should ensure that the library containing your source files precedes the library containing the IBM-supplied source files in the library list.)

Source File Attributes

Source files usually have the following attributes:

- A record length of 92 characters (this includes a 6-byte sequence number, a 6-byte date, and 80 bytes of source).
- Keys (sequence numbers) that are unique even though the access path does not specify unique keys. You are not required to specify a key for a source file. Default source files are created without keys (arrival sequence access path). A source file created with an arrival sequence access path requires less storage space and reduces save/restore time in comparison to a source file for which a keyed sequence access path is specified.
- More than one member.
- Member names that are the same as the names of the objects that are created using them.
- The same record format for all records.
- Relatively few records in each member compared to most data files.

Some restrictions are:

- The source sequence number must be used as a key, if a key is specified.
- The key, if one is specified, must be in ascending sequence.
- The access path cannot specify unique keys.
- The ALTSEQ keyword is not allowed in DDS for source files.
- The first field must be a 6-digit sequence number field containing zoned decimal data and two decimal digits.
- The second field must be a 6-digit date field containing zoned decimal data and zero decimal digits.
- All fields following the second field must be zoned decimal or character.

Creating Source File without DDS

When you create a physical database source file without using DDS, but by specifying the record length (RCDLEN parameter), the source created contains three fields: *SRCSEQ*, *SRCDAT*, and *SRCDTA*. (The record length must include 12 characters for sequence number and date-of-last-change fields so that the length of the data portion of the record equals the record length minus 12.) The data portion of the record can be defined to contain more than one field (each of which must be character or zoned decimal). If you want to define the data portion of the record as containing more than one field, you must define the fields using DDS.

A record format consisting of the following three fields is automatically used for a source physical file created using the Create Source Physical File (CRTSRCPF) command:

Field	Name	Data Type and Length	Description
1	SRCSEQ	Zoned decimal, 6 digits, 2 decimal positions	Sequence number for record
2	SRCDAT	Zoned decimal, 6 digits, no decimal positions	Date of last update of record
3	SRCDTA	Character, any length	Data portion of the record (text)

Note: For all IBM-supplied database source files, the length of the data portion is 80 bytes. For IBM-supplied device source files, the length of the data portion is the maximum record length for the associated device.

Creating Source Files with DDS

If you want to create a source file for which you need to define the record format, use the Create Physical File (CRTPF) or Create Logical File (CRTLF) command. If you create a source logical file, the logical file member should only refer to one physical file member to avoid duplicate keys.

Working with Source Files

The following section describes how to enter and maintain data in your source files.

Using the Source Entry Utility

You can use SEU, to enter and change source in a source file. If you use SEU to enter source in a database file, SEU adds the sequence number and date fields to each source record. (See the *SEU User's Guide/Reference* for more information about SEU.)

If you use SEU to update a source file, you can add records between existing records. For example, if you add a record between records 0003.00 and 0004.00, the sequence number of the added record could be 0003.01. SEU will automatically arrange the newly added statements in this way.

When records are first placed in a source file, the date field is all zoned decimal zeros (unless DDS is used with the DFT keyword specified). If you use SEU, the date field changes in a record when you change the record.

Using Device Source Files

Tape and diskette unit files can be created as source files. When device files are used as source files, the record length must include the sequence number and date fields. Any maximum record length restrictions must consider these additional 12 characters. For example, the maximum record length for a tape record is 32,766. If data is to be processed as source input, the actual tape data record has a maximum length of 32,754 (which is 32,766 minus 12).

If you open source device files for input, the system adds the sequence number and date fields, but there are zeros in the date fields.

If you open a device file for output and the file is defined as a source file, the system deletes the sequence number and date before writing the data to the device.

Copying Source File Data

The Copy Source File (CPYSRCF) and Copy File (CPYF) commands can be used to write data to and from source file members.

Using the Copy Source File (CPYSRCF) Command for Copying to and from Source Files

The CPYSRCF command is designed to operate with database source files. Although it is similar, in function, to the Copy File (CPYF) command, the CPYSRCF command provides defaults that are normally used when copying a source file. For example, it has a default that assumes the TOMBR parameter is the same as the FROMMBR parameter and that any TOMBR records will always be replaced. The CPYSRCF command also supports a unique printing format when TOFILE(*PRINT) is specified. Therefore, when you are copying database source files, you will probably want to use the CPYSRCF command.

Using the Copy File (CPYF) Command for Copying to and from Files

The CPYF command provides additional functions over the CPYSRCF command such as:

- Copying from database source files to device files
- Copying from device files to database files
- Copying between nonsource database files and source database files
- Printing a source member in hexadecimal format
- Copying source with selection values

Source Sequence Numbers Used in Copies

When you copy to a database source file, you can use the SRCOPT parameter to update sequence numbers and initialize dates to zeros. By default, the system assigns a sequence number of 1.00 to the first record and increases the sequence numbers by 1.00 for the remaining records. You can use the SRCSEQ parameter to set a fractional increased value and to specify the sequence number at which the renumbering is to start. For example, if you specify in the SRCSEQ parameter that the increased value is .10 and is to start at sequence number 100.00, the copied records have the sequence numbers 100.00, 100.10, 100.20, and so on.

If a starting value of .01 and an increased value of .01 are specified, the maximum number of records that can have unique sequence numbers is 999,999. When the maximum sequence number (9999.99) is reached, any remaining records will have a sequence number of 9999.99.

The following is an example of copying source from one member to another in the same file. If MBRB does not exist, it is added; if it does exist, all records are replaced.

```
CPYSRCF FROMFILE(QCLSRC) TOFILE(QCLSRC) FROMMBR(MBRA) +
        TOMBR(MBRB)
```

The following is an example of copying a generic member name from one file to another. All members starting with PAY are copied. If the corresponding members do not exist, they are added; if they do exist, all records are replaced.

```
CPYSRCF FROMFILE(LIB1/QCLSRC) TOFILE(LIB2/QCLSRC) +
        FROMMBR(PAY*)
```

The following is an example of copying the member PAY1 to the printer file QSYSPRT (the default for *PRINT). A format similar to the one used by SEU is used to print the source statements.

```
CPYSRCF FROMFILE(QCLSRC) TOFILE(*PRINT) FROMMBR(PAY1)
```

When you copy from a device source file to a database source file, sequence numbers are added and dates are initialized to zeros. Sequence numbers start at 1.00 and are increased by 1.00. If the file being copied has more than 9999 records, then the sequence number is wrapped back to 1.00 and continues to be increased unless the SRCOPT and SRCSEQ parameters are specified.

When you are copying from a database source file to a device source file, the date and sequence number fields are removed.

Using Source Files in a Program

You can process a source file in your program. You can use the external definition of the source file and do any input/output operations for that file, just as you would for any other database file.

Source files are externally described database files. As such, when you name a source file in your program and compile it, the source file description is automatically included in your program printout. For example, assume you wanted to read and update records for a member called FILEA in the source file QDDSSRC. When you write the program to process this file, the system will include the *SRCSEQ*, *SRCDAT*, and *SRCDTA* fields from the source file.

Note: You can display the fields defined in a file by using the Display File Field Description command (DSPFFD). For more information about this command, see "Displaying the Descriptions of the Fields in a File" on page 15-1.

The program processing the FILEA member of the QDDSSRC file could:

- Open the file member (just like any other database file member).
- Read and update records from the source file (probably changing the *SRCDTA* field where the actual source data is stored).
- Close the source file member (just like any other database file member).

Creating an Object Using a Source File

You can use a create command to create an object using a source file. If you create an object using a source file, you can specify the name of the source file on the create command.

For example, to create a CL program, you use the Create Control Language Program (CRTCLPGM) command. A create command specifies through a SRCFILE parameter where the source is stored.

The create commands are designed so that you do not have to specify source file name and member name if you do the following:

1. Use the default source file name for the type of object you are creating. (To find the default source file name for the command you are using, see "IBM-Supplied Source Files" on page 17-2.)
2. Give the source member the same name as the object to be created.

For example, to create the CL program PGMA using the command defaults, you would simply type:

```
CRTCLPGM PGM(PGMA)
```

The system would expect the source for PGMA to be in the PGMA member in the QCLSRC source file. The library containing the QCLSRC file would be determined by the library list.

As another example, the following Create Physical File (CRTPF) command creates the file DSTREF using the database source file FRSOURCE. The source member is named DSTREF. Because the SRCMBR parameter is not specified, the system assumes that the member name, DSTREF, is the same as the name of the object being created.

```
CRTPF FILE(QGPL/DSTREF) SRCFILE(QGPL/FRSOURCE)
```

Creating an Object from Source Statements in a Batch Job

If your create command is contained in a batch job, you can use an inline data file as the source file for the command. However, inline data files used as a source file should not exceed 10,000 records. The inline data file can be either named or unnamed. Named inline data files have a unique file name that is specified on the //DATA command. (For more information about inline data files, see the *Data Management Guide*.

Unnamed inline data files are files without unique file names; they are all named QINLINE. The following is an example of an inline data file used as a source file:

```
//BCHJOB
CRTPF FILE(DSTPRODLB/ORD199) SRCFILE(QINLINE)
//DATA FILETYPE(*SRC)
   .
   .   (source statements)
   .
//
//ENDBCHJOB
```

In this example, no file name was specified on the //DATA command. An unnamed spooled file was created when the job was processed by the spooling reader. The CRTPF command must specify QINLINE as the source file name to access the unnamed file. The //DATA command also specifies that the inline file is a source file (*SRC specified for the FILETYPE parameter).

If you specify a file name on the //DATA command, you must specify the same name on the SRCFILE parameter on the CRTPF command. For example:

```
//BCHJOB
CRTPF FILE(DSTPRODLB/ORD199) SRCFILE(ORD199)
//DATA FILE(ORD199) FILETYPE(*SRC)
   .
   .   (source statements)
   .
//
//ENDBCHJOB
```

If a program uses an inline file, the system searches for the first inline file of the specified name. If that file cannot be found, the program uses the first file that is unnamed (QINLINE).

If you do not specify a source file name on a create command, an IBM-supplied source file is assumed to contain the needed source data. For example, if you are creating a CL program but you did not specify a source file name, the IBM-supplied source file QCLSRC is used. You must have placed the source data in QCLSRC.

If a source file is a database file, you can specify a source member that contains the needed source data. If you do not specify a source member, the source data must be in a member that has the same name as the object being created.

Determining Which Source File Member Was Used to Create an Object

When an object is created from source, the information about the source file, library, and member is held in the object. The date/time that the source member was last changed before object creation is also saved in the object.

The information in the object can be displayed with the Display Object Description (DSPOBJD) command and specifying DETAIL(*SERVICE).

This information can help you in determining which source member was used and if the existing source member was changed since the object was created.

You can also ensure that the source used to create an object is the same as the source that is currently in the source member using the following commands:

- The Display File Description (DSPFD) command using TYPE(*MBR). This display shows both date/times for the source member. The `Last source update date/time` value should be used to compare to the `Source file date/time` value displayed from the DSPOBJD command.
- The Display Object Description (DSPOBJD) command using DETAIL(*SERVICE). This display shows the date/time of the source member used to create the object.

Note: If you are using the data written to output files to determine if the source and object dates are the same, then you can compare the *ODSRCD* (source date) and *ODSRCT* (source time) fields from the output file of the DSPOBJD DETAIL(*SERVICE) command to the *MBUPDD* (member update date) and *MBUPDT* (member update time) fields from the output file of the DSPFD TYPE(*MBR) command.

Managing a Source File

This section describes several considerations for managing source files.

Changing Source File Attributes

If you are using SEU to maintain database source files, see the *SEU User's Guide/Reference* for how to change database source files. If you are not using SEU to maintain database source files, you must totally replace the existing member.

If your source file is on a diskette, you can copy it to a database file, change it using SEU, and copy it back to a diskette. If you do not use SEU, you have to delete the old source file and create a new source file.

If you change a source file, the object previously created from the source file does not match the current source. The old object must be deleted, and a create command must be specified to re-create the object using the changed source file. For example, if you change the source file FRSOURCE created in "Creating an Object Using a Source File" on page 17-6, you have to delete the file DSTREF that was created from the original source file, and create it again using the new source file so that DSTREF matches the changed FRSOURCE source file.

Reorganizing Source File Member Data

You usually do not need to reorganize a source file member if you use arrival sequence source files.

To assign unique sequence numbers to all the records, specify the following parameters on the Reorganize Physical File Member (RGZPFM) command:

- KEYFILE(*NONE), so that the records are not reorganized
- SRCOPT(*SEQNBR), so that the sequence numbers are changed
- SRCSEQ with a fractional value such as .10 or .01, so that all the sequence numbers are unique

Note: Deleted records, if they exist, will be compressed out.

A source file with an arrival sequence access path can be reorganized by sequence number if a logical file for which a keyed sequence access path is specified is created over the physical file.

Determining When a Source Statement Was Changed

Each source record contains a date field which is automatically updated by SEU if a change occurs to the statement. This can be used to determine when a statement was last changed. Most high-level language compilers print these dates on the compiler lists. The Copy File (CPYF) and Copy Source File (CPYSRCF) commands also print these dates.

Each source member description contains two date and time fields. The first date/time field reflects changes caused by SEU any time the member is closed after it was updated.

The second date/time field reflects any changes to the member. This includes all changes caused by SEU, the CPYSRCF command, authorization changes, and changes to the file status. For example, the FRCRATIO parameter on the Change Physical File (CHGPF) command changes the member status. This date/time is used by the Save Changed Objects (SAVCHGOBJ) command to determine if the member should be saved. Both date/times can be displayed with the Display File Description (DSPFD) command specifying TYPE(*MBR). There are two change date/times shown with source members:

- `Last source update date/time`. This value reflects any change to the source data records in the member. When a source update occurs, the `Last change date/time` value is also updated although there may be a 1- or 2-second difference in that time/date value.
- `Last change date/time`. This value reflects any changes to the member. This includes all changes caused by SEU, CPYSRCF, authorization changes, or changes to file status. For example, the FRCRATIO parameter on the CHGPF command changes the member status, and therefore, is reflected in the `Last change date/time` value.

Considerations for Saving Source Files

When you save an object to tape or diskette, the system updates the object description with the date and time of the save operation. When you save an object to a save file, you can prevent the system from updating the date and time of the save operation by specifying UPDHST(*NO) on the save command. When you restore an object, the system always updates the object description with the date and time of the restore operation. You can display this and other save/restore related information by using the Display Object Description (DSPOBJD) command with DETAIL(*FULL). Use the Display Save File (DSPSAVF) command to display the objects in a save file.

Specify DATA(*SAVRST) on the Display Diskette (DSPDKT) or Display Tape (DSPTAP) command for a display of the objects on the media.

The last save/restore date for database members can be displayed by typing:

```
DSPFD FILE(file-name) TYPE(*MBR)
```

Using Source Files for Documentation

You can use the IBM-supplied source file QTXTSRC to help you create and update online documentation.

You can create and update QTXTSRC members just like any other application (such as QRPGSRC or QCLSRC) available with SEU. The QTXTSRC file is most useful for narrative documentation, which can be retrieved online or printed. The text that you put in a source member is easy to update by using the SEU add, change, move, copy, and include operations. The entire member can be printed by specifying Yes for the print current source file option on the exit prompt. You can also write a program to print all or part of a source member.

Appendix A. Database File Sizes

The following database file maximums should be kept in mind when designing files on the AS/400 system:

Description	Maximum Value
Number of bytes in a record	32,766 bytes
Number of fields in a record format	8,000 fields
Number of key fields in a file	120 fields[1]
Size of key for physical and logical files	120 characters[1]
Size of key for ORDER BY (SQL/400 and OPNQRYF)	256 characters
Number of records contained in a file	16,777,215 records[2]
Number of bytes in a file	2,147,483,648 bytes
Number of bytes in an access path	1,073,741,824 bytes
Number of keyed logical files built over a physical file	3,686 files
Number of physical file members in a logical file member	32 members
Number of members that can be joined	32 members
Size of a character or DBCS field	32,766 characters
Size of a zoned decimal or packed decimal field	31 digits

[1] The maximum values for the number of key fields in a file and the size of the key for physical and logical files are 115 fields and 115 characters when a first-changed-first-out (FCFO) access path is specified for the file.

[2] For files with keyed sequence access paths, the maximum number of records in a member varies and can be estimated using the following formula:

$$\frac{716{,}800{,}000}{10 + (.8 * \text{key length})}$$

This is an estimated value, the actual maximum number of records can vary significantly from the number determined by this formula.

These are maximum values. There are situations where the actual limit you experience will be less than the stated maximum. For example, certain high-level languages can have more restrictive limits than those described above.

Keep in mind that performance can suffer as you approach some of these maximums. For example, the more logical files you have built over a physical file, the greater the chance that system performance can suffer (if you are frequently changing data in the physical file that causes a change in many logical file access paths).

Normally, an AS/400 database file can grow until it reaches the maximum size allowed on the system. The system normally will not allocate all the file space at once. Rather, the system will occasionally allocate additional space as the file grows larger. This method of automatic storage allocation provides the best combination of good performance and effective auxiliary storage space management.

If you want to control the size of the file, the storage allocation, and whether the file should be connected to auxiliary storage, you can use the SIZE, ALLOCATE, and CONTIG parameters on the Create Physical File (CRTPF) and the Create Source Physical File (CRTSRCPF) commands.

You can use the following formulas to estimate the disk size of your physical and logical files.

- For a physical file (excluding the access path):

  Disk size=(number of valid and deleted records + 1) * (record length + 1) + 5120 * (number of members) + 1024

  The size of the physical file depends on the SIZE and ALLOCATE parameters on the CRTPF and CRTSRCPF commands. If you specify ALLOCATE(*YES), the initial allocation and increment size on the SIZE keyword must be used instead of the number of records.
- For a logical file (excluding the access path):

  Disk size = (2560) * (number of members) + 1024
- For a keyed sequence access path, per member:

  (number of keys) * (key length + 8) * (0.8) * (1.85) + 4096

  Notes:

  1. The number of keys is the number of index entries in the access path.
  2. The key length is the sum of the length of all the key fields.
  3. 0.8 is an estimate of key compression.
  4. 1.85 is an estimate of index overhead and space reserved for inserting keys.

This estimate can differ significantly from your file. The keyed sequence access path depends heavily on the data in your records. The only way to get an accurate size is to load your data and display the file description.

The following is a list of minimum file sizes:

Description	Minimum Size
Physical file without a member	1024 bytes
Physical file with a single member	5632 bytes
Keyed sequence access path	1024 bytes

Note: Additional space is not required for an arrival sequence access path.

In addition to the file sizes, the system maintains internal formats and directories for database files. (These internal objects are owned by user profile QDBSHR.) The following are estimates of the sizes of those objects:

- For any file not sharing another file's format:

  Format size = (75 * number of fields) + 512
- For files sharing their format with any other file:

  Format sharing directory size = (16 * number of files sharing the format) + 272
- For each physical file and each physical file member having a logical file or logical file member built over it:

  Data sharing directory size = (16 * number of files or members sharing data) + 272
- For each file member having a logical file member sharing its access path:

  Access path sharing directory size = (16 * number of files or members sharing access path) + 272

Appendix B. Double-Byte Character Set (DBCS) Considerations

The double-byte character set (DBCS) is a character set that represents each character with 2 bytes. The DBCS supports national languages that contain a large number of unique characters or symbols (the maximum number of characters that can be represented with 1 byte is 256 characters). Examples of such languages include Japanese, Korean, and Chinese.

This appendix describes DBCS considerations as they apply to the database on the AS/400 system.

DBCS Field Data Types

The DBCS data types (specified in position 35 on the *DDS Coding Form*) are:

Entry	**Meaning**
O	DBCS-open
E	DBCS-either
J	DBCS-only

A DBCS-open field is a character string that contains both single- and double-byte characters. A DBCS-only field is a character string that contains only double-byte characters. A DBCS-either field is a character string that contains either all single-byte data or all double-byte data.

Note: When a field contains double-byte characters, a shift-out character comes before the first DBCS character and a shift-out character follows the last DBCS character.

DBCS Field Mapping Considerations

The following chart shows what types of data mapping are valid between physical and logical files for DBCS fields:

Physical File Data Type	**Logical File Data Type**				
	Character	**Hexadecimal**	**DBCS-Open**	**DBCS-Either**	**DBCS-Only**
Character	Valid	Valid	Valid	Valid	Not valid
Hexadecimal	Valid	Valid	Valid	Valid	Valid
DBCS-open	Not valid	Valid	Valid	Not valid	Not valid
DBCS-either	Not valid	Valid	Valid	Valid	Not valid
DBCS-only	Not valid	Valid	Valid	Valid	Valid

DBCS Field Concatenation

When fields are concatenated, the data types can change (the resulting data type is automatically determined by the system).

- OS/400 assigns the data type based on the data types of the fields that are being concatenated. When DBCS fields are included in a concatenation, the general rules are:
  - If the concatenation contains one or more hexadecimal (H) fields, the resulting data type is hexadecimal (H).
  - If all fields in the concatenation are DBCS-only (J), the resulting data type is DBCS-only (J).
  - If the concatenation contains one or more DBCS (O, E, J) fields, but no hexadecimal fields, the resulting data type is DBCS open (O).
- The maximum length of a concatenated field varies depending on the data type of the concatenated field and length of the fields being concatenated. If the concatenated field is zoned decimal (S), its total length cannot exceed 31 bytes; if it is character (A) or DBCS (O, J) its total length cannot exceed 32 766 bytes.
- In join logical files, the fields to be concatenated must be from the same physical file. The first field specified on the CONCAT keyword identifies which physical file is used. The first field must, therefore, be unique among the physical files on which the logical file is based, or you must also specify the JREF keyword to specify which physical file to use.
- The use of a concatenated field must be I (input only).
- REFSHIFT cannot be specified on a concatenated field that has been assigned a data type of O or J.

Notes:

1. When DBCS fields are concatenated, a shift-in at the end of one field and a shift-out at the beginning of the next field are removed, if the resulting data type is hexadecimal, the shift-in and shift-out pairs are only eliminated for DBCS fields that precede the first hexadecimal field.
2. A concatenated field that contains DBCS fields must be an input-only field.

Comparing DBCS Fields

When comparing two fields or constants, the data types must be compatible. The following table describes valid comparisons for DBCS fields:

	Any Numeric	Character	Hexadecimal	DBCS-Open	DBCS-Either	DBCS-Only
Any Numeric	Valid	Not valid	Not valid	Not valid	Not valid	Not valid
Character	Not valid	Valid	Valid	Valid	Valid	Valid
Hexadecimal	Not valid	Valid	Valid	Valid	Valid	Valid
DBCS-Open	Not valid	Valid	Valid	Valid	Valid	Valid
DBCS-Either	Not valid	Valid	Valid	Valid	Valid	Valid
DBCS-Only	Not valid	Not valid	Valid	Valid	Valid	Valid

Using DBCS Fields in the Open Query File (OPNQRYF) Command

This section describes considerations when using DBCS fields in the Open Query File (OPNQRYF) command.

Using the Wild Card Function with DBCS Fields

When using the "wild card" (%WLDCRD) function with any DBCS field, both single-byte and double-byte wild card values (asterisk and underline) are allowed. The following special rules apply:

- A single-byte underline refers to one EBCDIC character; a double-byte underline refers to one double-byte character.
- A single- or double-byte asterisk refers to any number of characters of any type.

Appendix C. Database Lock Considerations

The following table summarizes some of the most commonly used database functions and the types of locks they place on database files.

Function	Command	File Lock	Member/Data Lock	Access Path Lock
Add Member	ADDPFM, ADDLFM	LEAR		LEAR
Change File Attributes	CHGPF,CHGLF	LENR	LEAR	LEAR
Change Member Attributes	CHGPFM, CHGLFM	LSRD	LEAR	
Check Object	CHKOBJ	LSRO		
Clear Physical File Member	CLRPFM	LSRO	LENR	
Create Duplicate Object	CRTDUPOBJ	LENR		
Create File	CRTPF, CRTLF, CRTSRCPF	LENR		
Delete File	DLTF	LENR		LEAR
Grant/Revoke Authority	GRTOBJAUT, RVKOBJAUT	LENR		
Initialize Physical File Member	INZPFM	LSRD	LEAR	
Move Object	MOVOBJ	LENR		
Open File	OPNDBF, OPNQRYF	LSRD	LSRD	LEAR
Rebuild Access Path		LSRD	LSRD	LENR
Remove Member	RMVM	LEAR	LENR	LEAR
Rename File	RNMOBJ	LENR	LENR	LENR
Rename Member	RNMM	LEAR	LENR	LENR
Reorganize Physical File Member	RGZPFM	LENR		
Restore File	RSTLIB, RSTOBJ	LENR		
Save File	SAVLIB, SAVOBJ, SAVCHGOBJ	LSRO	LSRO	

The following table shows the valid lock combinations:

Lock	LENR	LEAR	LSUP	LSRO	LSRD
LENR[1]					
LEAR[2]					X
LSUP[3]			X		X
LSRO[4]				X	X
LSRD[5]		X	X	X	X

[1] Exclusive lock. The object is allocated for the exclusive use of the requesting job; no other job can use the object.

[2] Exclusive lock, allow read. The object is allocated to the job that requested it, but other jobs can read the object.

[3] Shared lock, allow read and update. The object can be shared either for read or change with other jobs.

[4] Shared lock, read only. The object can be shared for read with other jobs.

[5] Shared lock. The object can be shared with another job if the job does not request exclusive use of the object.

Glossary

absolute value. The numeric value of a real number regardless of its sign (positive or negative).

access. To read; the ability to use or read.

access path. (1) The order in which records in a database file are organized for processing by a program. See *arrival sequence access path* and *keyed sequence access path*.

access path journaling. A method of recording changes to an access path as changes are made to the data in the database file so that the access path can be recovered automatically by the system.

active record. Any record format that is currently displayed or an active subfile record.

activity level. A characteristic of main storage or of the processing unit that specifies the maximum number of jobs that can run at the same time in the main storage or in the processing unit.

address. The location in the storage of a computer where particular data is stored. Also, the numbers that identify such a location.

alert. A record sent to a focal point to identify a problem or an impending problem.

algorithm. A finite set of well-defined rules for the solution of a problem in a finite number of steps.

alias. An alternative name used by a high-level language program to identify (file definition) an object.

allocate. To reserve a resource for use in performing a specific task.

alternative collating sequence. A user-defined collating sequence that replaces the standard EBCDIC collating sequence. See *collating sequence*.

application. A particular business task, such as inventory control or accounts receivable.

application program. A program used to perform a particular data processing task such as inventory control or payroll.

arrival sequence access path. An access path to a database file that is arranged according to the order in which records are stored in the physical file. See also *keyed sequence access path* and *access path*.

ascending sequence. The arrangement of data in order from the lowest value to the highest value, according to the rules for comparing data. Contrast with *descending sequence*.

attribute. A characteristic or property of one or more objects.

authority holder. An object that specifies and reserves an authority to a program-described database file before the file is created. When the file is created, the authority specified in the holder is linked to it.

authorization list. A list of two or more user IDs and their authorities for system resources.

authorize. Permit or give authority to.

auxiliary storage. All addressable disk storage other than main storage.

auxiliary storage pool (ASP). A grouping of disk units.

backup. Pertaining to an alternative copy used as a substitute if the original is lost or destroyed.

base. The numbering system in which an arithmetic value is represented.

BASIC (beginner's all-purpose symbolic instruction code). A programming language with a small list of commands and a simple syntax, primarily designed for numeric applications.

batch. Pertaining to a group of jobs to be run on a computer sequentially with the same program with little or no operator action. Contrast with *interactive*.

batch job. A predefined group of processing actions submitted to the system to be performed with little or no interaction between the user and the system. Contrast with *interactive job*.

bit. Either of the binary digits, 0 or 1. Compare with *byte*.

both field. A field that can be used for either input data or output data.

buffer. (1) A routine or an area of storage that corrects for the different speeds of data flow or timings of events, when transferring data from one device to another. (2) A portion of storage used to hold input or output data temporarily.

built-in function. A predefined function, such as a commonly used arithmetic function or a function necessary to high-level language compilers (for example, a function for manipulating character strings or converting data). It is automatically called by a built-in function reference.

business graphics. See *graphics*.

byte. A group of 8 adjacent bits. In the EBCDIC coding system, 1 byte can represent a character. In the double-byte coding system, 2 bytes represent a character.

C & SM. See *communications and systems management (C & SM)*.

carrier. A continuous frequency that can be varied with a second signal to send information.

carrier return. A character that is automatically inserted at the end of a full line of text as you type information on the display. The character is not displayed in the audit window. See also *required carrier return*.

catalog. Tables, maintained by the database utility, that contain descriptions of objects, such as tables, views, and indexes.

chain. A group of logically linked records.

character. Any letter, number, or other symbol in the data character set that is part of the organization, control, or representation of data.

character field. An area that is reserved for information that can contain any of the characters in the character set. Contrast with *numeric field*.

character set. A group of characters used for a specific reason; for example, the set of characters the computer can display, the set of characters a printer can print, or a particular set of graphic characters in a code page; for example, the 256 EBCDIC characters.

character string. A sequence of consecutive characters that are used as a value.

character variable. Character data whose value is assigned and/or changed while the program is running.

chart. Displayed, printed, or plotted output that compares one or more sets of variable data in chart form. The types of charts are bar, line, pie, surface, histogram, Venn diagram, and text.

CL. See *control language (CL)*.

CLEAR. A command used to delete all requests and responses related to the active session.

close. The function that ends the connection between a file and a program, and ends the processing. Contrast with *open*.

COBOL (common business-oriented language). A high-level programming language, based on English, that is used primarily for commercial data processing.

COBOL/400. A licensed program that is a high-level programming language, resembling English. COBOL/400 is especially efficient in the processing of business problems.

code. A type of control character that is used within the text of a document to control the format of the printed output, such as underline, bold, and tab.

code page. A particular assignment of hexadecimal identifiers to graphic characters.

collating sequence. The order in which characters are arranged within the computer for sorting, combining, or comparing.

command. A statement used to request a function of the system. A command consists of the command name, which identifies the requested function and parameters.

command definition. An object that contains the definition of a command (including the command name, parameter descriptions, and validity checking information) and identifies the program that performs the function requested by the command. The system-recognized identifier for the object type is *CMD.

commit. To make all changes permanent that were made to one or more database files since the last commit or rollback operation, and make the changed records available to other users.

commitment control. A means of grouping file operations that allows the processing of a group of database changes as a single unit through the Commit command or the removal of a group of database changes as a single unit through the Rollback command.

compatibility. Ability to work in the system or ability to work with other devices or programs.

compatible. Pertaining to the characteristics that make devices, programs, products, or systems work together.

compilation. Translation of a source program (such as RPG/400 or COBOL specifications) into a program in machine language.

compile. To translate a program written in a high-level programming language into a machine-language program.

compiler. A program that translates programming language into machine language for use by the computer.

completion message. A message that tells the operator when work is successfully ended.

compress. To replace repetitive characters in a file or folder with control characters so that the file or folder takes up less space when saved on diskette.

concatenated field. Two or more fields that are combined to make one field in a logical file.

concept. An abstract idea.

configuration. The physical and logical arrangement of devices and programs that make up a data processing system.

confirmation of delivery. The automatic notification to the sender of a message, note, or document as to when the message, note, or document is received. Confirmation of delivery must be requested by the sender.

conform. To change to a prevailing standard.

constant. Data that has an unchanging, predefined value to be used in processing.

control language (CL). The set of all commands with which a user requests system functions.

control language (CL) program. A program that is created from source statements consisting entirely of control language commands.

control language (CL) variable. A program variable that is declared in a control language program and is available only to the CL program.

creation date. The system date when an object is created.

current position. The position, in user coordinates, that becomes the starting point for the next graphics routine, if that routine does not explicitly specify a starting point.

current record. The record that is available in the record area associated with the file.

data area. A storage area used to communicate data such as CL variable values between the programs within a job and between jobs. The system-recognized identifier for the data area is *DTAARA.

data authority. A specific authority to read, add, update, or delete data.

data definition. Information that describes the contents and characteristics of a field, record format, or file. A data definition can include such things as field names, lengths, and data types. See also *field definition*.

data description specifications (DDS). A description of the user's database or device files that is entered into the system in a fixed form. The description is then used to create files.

data dictionary. An object for storing field, record format, and file definitions.

data file. (1) A collection of related data records organized in a specific order. (2) A file created by the specification of FILETYPE(*DATA) on the create commands. Contrast with *source file*.

data file utility (DFU). The part of the AS/400 Application Development Tools licensed program that is used to enter, maintain, and display records in a database file.

data management. The part of the operating system that controls the storing and accessing of data to or from an application program. The data can be on internal storage (for example, database), on external media (diskette, tape, or printer), or on another system.

data stream. All information (data and control commands) sent over a data link usually in a single read or write operation.

data type. A characteristic used for defining data as numeric or character.

database. The collection of all data files stored in the system.

database file. An object that contains descriptions of how input data is to be presented to a program from internal storage and how output data is to be presented to internal storage from a program. See also *physical file* and *logical file*.

DBCS. See *double-byte character set (DBCS)*.

DBCS-either field. A field that can contain either double-byte data or alphameric data. See also *DBCS-only field* and *DBCS-open field*.

DBCS-only field. A field that must contain only double-byte characters. See also *DBCS-either field* and *DBCS-open field*.

DBCS-open field. A field that can contain a mixture of alphameric and double-byte characters with shift-out and shift-in characters marking the transitions. See also *DBCS-either field* and *DBCS-only field*.

DDM file. A file description, created by a user on the local (source) system, for a database file that is stored on a remote (target) system. The DDM file provides the information needed for a local system to locate a remote system and to access the data in the remote database file.

DDS. See *data description specifications (DDS)*.

debug mode. An environment in which programs can be tested.

default. A value automatically supplied or assumed by the system or program.

default record. A record that consists entirely of default values (numeric fields are filled with zeros, character fields are filled with blanks, or either data type, numeric or character, can be filled with a value specified by the user with the DFT keyword in DDS).

default value. (DDS) The value specified by the user with the DFT keyword in DDS.

delayed maintenance. A method of logging changes to an access path for database files and applying the changes the next time the file is opened instead of rebuilding the access path completely or maintaining it immediately. Contrast with *rebuild maintenance* and *immediate maintenance*.

descending sequence. The arrangement of data in order from the highest value to the lowest value, according to the rules for comparing data. Contrast with *ascending sequence*.

detail record. A record that contains the daily activities or transactions of a business. For example, the items on a customer order are typically stored in detail records. Contrast with *header record*.

device file. A file that contains a description of how data is to be presented to a program from a device or how data is to be presented to the device from the program. Devices can be display stations, printers, a diskette unit, tape units, or a remote system.

DFU. See *data file utility (DFU)*.

DIA. See *Document Interchange Architecture (DIA)*.

diagnostic. Pertaining to the detection and isolation of an error.

diagnostic message. A message that contains information about errors or possible errors. This message is generally followed by an escape message.

digit. Any of the numerals from 0 through 9.

directory. A list of the files that are stored on a disk or diskette. A directory also contains information about the file, such as size and date of last change.

disk. A direct-access storage medium with magnetically recorded data.

diskette. A thin, removable magnetic disk in a protective jacket.

diskette file. A device file created by the user for a diskette unit.

display file. A device file created by the user to support a display station.

distributed data management (DDM). A function of the operating system that allows an application program or user on one system to use files stored on remote systems. The systems must be connected by a communications network, and the remote systems must also be using DDM.

distribution list. A list of system distribution directory entries, which allows users to send messages, notes, and documents to a group of users in one step.

document class. A user-defined character string, 1 through 16 characters long, that characterizes a document. It can be used to search for a filed document. For example, a document that is a memo could have a document class of MEMO; a document that is a report, REPORT.

document description. The 1- through 44-character description of a document, assigned by the user when creating or filing the document.

document descriptor. The internal description of the document supplied by the system, which contains the object name, user ID, and security associated with that user; and may also contain the document details, such as subject, author, document description, or other identifying information defined by the user for that document. See also *document description*.

Document Interchange Architecture (DIA). The rules and structure for the exchange of information between office applications. Document Interchange Architecture includes document library services and document distribution services.

document library. The system library named QDOC that contains all documents and folders.

document library object. Either a document or folder.

document name. The 1- through 12-character name for documents in folders, assigned by the user when creating the document.

document type. The document interchange architecture type.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, displaying, and printing of DBCS characters requires hardware and programs that support DBCS. Four double-byte character sets are supported by the system: Japanese, Korean, Simplified Chinese, and Traditional Chinese.

dynamic. Pertaining to events occurring at run time, or during processing.

dynamic select/omit. Selection and omission of logical file records performed during processing, instead of when the access path (if any) is maintained. Dynamic select/omit may also be used when no keyed access path exists.

edit word. A user-defined word with a specific format that indicates how editing should be done.

end-of-file delay. An interval during which the system holds a file open after the normal end of the file is reached until one or more records are updated or added to the end of the file. The length of the interval can be specified on the EOFDLY parameter.

error log. A record of machine checks, device errors, and media statistics.

expiration date. The date after which a database file member should not be used.

exponent. In floating-point format, an integer constant specifying the power of ten by which the base of the decimal floating-point number is to be multiplied.

exponent (of an E-format number). An integer constant specifying the power of ten by which the base of the decimal floating-point number is to be multiplied.

expression. (DDS) A pair of values that represents a single parameter value.

extent. The number of integers between and including the lower and upper bounds of an array.

externally described data. Data contained in a file for which the fields and the records are described outside of the program (such as with DDS, IDDU, SQL/400), that processes the file.

externally described file. A file in which the records and fields are described to the system when the file is created, and used by the program when the file is processed. Contrast with *program-described file*.

factor. An entry (for example, a field name, file name, literal, or data structure) that identifies the data to be used in an operation.

field. A group of related characters (such as name or amount) that are treated as a unit in a record.

field definition. Information that describes the characteristics of data in a field.

field reference file. A physical file that contains no data, only descriptions of fields.

file. A generic term for the object type that refers to a database file, a device file, or a set of related records treated as a unit. The system-recognized identifier for the object type is *FILE.

file description. A personal computer file that describes another personal computer data file. The description includes the name, data type, field length, and format of the data file. This information is used by the AS/400 PC Support transfer function to transfer data to the AS/400 system.

file identifier. A 3-character identifier used for files being joined in AS/400 Query for a query. The identifiers are used during query definition to uniquely identify each file.

file name. The name used by a program to identify a file. See also *label*.

file organization. The permanent file arrangement established at the time that a file is created.

file overrides. Attributes specified at run time that change the attributes specified in the file description or in the program.

FILE type. A data type that allows the program to read input and write output in AS/400 Pascal.

filed document. Electronic mail or a document that is stored in the document library.

final-form text (FFT). A data stream defined by document content architecture that is used to exchange resolved documents (which can be printed directly by most printers or displayed) between systems. Contrast with *revisable-form text (RFT)*.

fixed length. A specified length for a record or field that cannot be changed.

fixed-length. Pertaining to a characteristic of a file in which all of the records are the same length.

flag. The bit sequence 01111110 used to mark a frame in SDLC.

floating-point. A mathematical notation in which a quantity of a number is shown as one number multiplied by a power of the number of the base.

floating-point format. In binary floating-point representation, the storage format that represents a binary floating-point value.

folder. A directory for documents. A folder is used to group related documents and to find documents by name. The system-recognized identifier for the object type is *FLR. Compare with *library*.

folder path. A folder name, followed by one or more additional folder names, where each preceding folder is found.

font. An assortment of characters of a given size and type style.

font ID. A number that identifies the character style and size for certain printers.

footer. One of more lines of text that prints at the bottom of every page of a document, such as a page number, the date, an outline heading, or the document ID.

form. A physical sheet of paper on which data is printed. Synonymous with medium, physical page, and sheet.

format. A group of related fields, such as a record, in a file.

format selector. A user-defined program (either a CL or a high-level language program) that determines where a record should be placed in the database when an application program does not pass a record format name for a record being added to a logical file.

function key. A keyboard key that allows the user to select keyboard functions or programmer functions.

generic. Relating to, or characteristic of, a whole group or class.

global. Pertains to information available to more than one program or subroutine.

graphics. Pertaining to charts, tables, and their creation.

group authority. Authority to use objects, resources, or functions from a group profile.

header record. A record that contains information, such as customer name and customer address, that is common to detail records. Contrast with *detail record*.

hexadecimal. Pertaining to a numbering system with a base of 16.

high-level language (HLL). A programming language, such as RPG, BASIC, PL/I, Pascal, COBOL, and C used to write computer programs.

high-level language (HLL) pointer. A source pointer that the programmer declares in the user program.

history log. A summary of the system activities, such as system and job information, device status, system operator messages and a record of program temporary fix (PTF) activity on the system.

hop. The transmission from one location to the next in a network.

I/O. See *input/output*.

IDDU. See *interactive data definition utility (IDDU)*.

identifier. A sequence of bits or characters that identifies a user, program, device, or system to another user, program, device, or system.

image. An electronic representation of an original document recorded by a scanning device.

immediate maintenance. A method of maintaining keyed access paths for database files. This method updates the access path whenever changes are made to the database file associated with the access path. Contrast with *rebuild maintenance* and *delayed maintenance*.

include. To insert text into a document when the document is printed.

indicator. A 2-character code that is used by a program to test a field or record or to tell when certain operations are to be performed.

indirect user. A person enrolled as an AS/400 Office user who is authorized to handle mail but has no mail log. An indirect user receives printed mail only.

infinity. A name referring to an indefinitely great number.

information (I) frame. An SDLC transmission frame that is sequentially numbered and used to transmit data.

informational message. A message that provides information to the user about the system as compared with a completion message, which indicates success, and escape or diagnostic messages, which indicate failure.

initial program. A user-profile program that runs when the user signs on and after the command processor program QCMD is started. QCMD calls the first program.

initial program load (IPL). The process that loads the system programs from the system auxiliary storage, checks the system hardware, and prepares the system for user operations.

initialize. To set the addresses, switches, or the contents of storage to zero, or to the starting value set by the manufacturer.

inline. Spooled input data that is read into a job by a reader.

inline data file. A file described by a Data command that is included as part of a job when the job is read from an input device. The file is deleted when the job ends.

input/output. Data provided to the computer or data resulting from computer processing.

inquiry message. A message that gives information and requests a reply.

interactive. Pertaining to the exchange of information between people and a computer. Contrast with *batch*.

interactive data definition utility (IDDU). A function of the operating system that can be used to externally define the characteristics of data and the contents of files.

interactive job. A job started for a person who signs on to a work station. Contrast with *batch job*.

interface. A shared boundary. An interface might be the hardware to connect two devices or it might be a part of main storage, or registers used by two or more computer programs.

inventory. The quantity of materials or goods on hand.

IPL. See *initial program load (IPL)*.

job. A unit of work to be done by a computer.

job log. A record of requests submitted to the system by a job, the messages related to the requests, and the actions performed by the system on the job. The job log is maintained by the system program.

join. An operation that combines data from two or more files using specified fields.

join field. A comparison field that identifies records from two files to be combined into one record.

join logical file. A logical file that combines (in one record format) fields from two or more physical files. See also *logical file*.

journal. A system object used to record entries in a journal receiver when a change is made to the database files associated with the journal. The object type is *JRN. See also *journal receiver.*

journal receiver. A system object that contains journal entries recorded when changes are made to the data in database files or the access paths associated with the database files. The object type is *JRNRCV. See also *journal*.

key. The value used to identify a record in a keyed sequence file.

key field. A field used to arrange the records of a particular type within a file member.

keyed sequence. The order in which indexed records are read by the program.

keyed sequence access path. An access path to a database file that is arranged according to the contents of key fields contained in the individual records. See also *arrival sequence access path* and *access path*.

keyword. (DDS) A name that identifies a function.

label. (1) The name of a file on a diskette or tape. (2) An identifier of a command or program statement generally used for branching.

leading zeros. Zeros that are place holders to the left of numbers that are aligned to the right and have fewer positions than the specified field length.

level checking. A function that compares the record format-level identifiers of a file to be opened with the file description that is part of a compiled program to determine if the record format for the file changed since the program was compiled.

library. An object on disk that serves as a directory to other objects. A library groups related objects, and allows the user to find objects by name. Compare with *folder* and *document library*.

library list. A list that indicates which libraries are to be searched and the order in which they are to be searched. The system-recognized identifier is *LIBL.

library name. A user-defined word that names a library.

licensed program. An IBM-written program that performs functions related to processing user data.

line. The physical path in data transmission.

line format. The defined arrangement of characters on a line. Compare with *page format*.

list ID. A two-part name by which a distribution list is known. The two-part name allows distributions to be sent to both local and remote systems.

literal. A character string whose value is defined by the characters themselves. For example, the numeric constant 7 has the value 7, and the character constant 'CHARACTERS' has the value CHARACTERS.

load. To move data or programs into storage.

local. Pertaining to a device or system that is connected directly to or a file that is read directly from your system, without the use of a communications line. Contrast with *remote*.

local data area. A 1024-byte data area that can be used to pass information between programs in a job. A separate local data area is automatically created for each job.

local system. For interactive jobs, the system to which the display device is directly attached. For batch jobs, the system on which the job is being processed.

lock. The process by which integrity of data is ensured by preventing more than one user from accessing the same data or object at the same time.

lock state. A condition defined for an object that determines how it is locked, how it is used (read or write), and whether the object can be shared (used by more than one job).

logic. The systematized interconnection of digital switching functions, circuits, or devices.

logical file. A description of how data is to be presented to or received from a program. This type of database file contains no data, but it defines record formats for one or more physical files. See also *join logical file*. Contrast with *physical file*.

logical file member. A named logical grouping of data records from one or more physical file members. See also *member*.

mail log. A record of all the electronic and printed mail that an office user has sent or received.

main storage. The part of the processing unit where programs are run. Synonymous with *memory*. Contrast with *auxiliary storage*.

map. In CSP/AD, a layout of a display or a printer format that has been defined using Cross System Product/Application Development on a System/370 development system.

master file. A collection of permanent information, such as a file of customer addresses.

matching record (MR) indicator. An indicator used in calculation or output specifications to indicate operations that are to be performed only when records match in primary and secondary files.

member. Different sets of data within one file. See also *source member*.

menu. A displayed list of available, logically grouped functions for selection by the operator.

merge. To insert records throughout a single output file.

message description. Information describing a particular message.

message queue. A list on which messages are placed when they are sent to a person or program.

mode. The session limits and common characteristics of the sessions associated with advanced-program-to-program communications (APPC) devices managed as a unit with a remote location.

monitor. A functional unit that observes and records selected activities for analysis within a data processing system.

network. A collection of data processing products connected by communications lines for exchanging information between stations.

next record. The record that logically follows the current record of a file.

not-a-number (NaN). In binary floating-point concepts, a value, not interpreted as a mathematical value, which contains a mask and a sequence of binary digits.

notify object. A message queue, a data area, or a database file that contains information identifying the last successful commitment operation. This information can be used by the programmer to find a restarting point for an application following an abnormal end to the system or routing step processing.

numeric field. An area that is reserved for a particular unit of information and that can contain only the digits 0 through 9. Contrast with *character field*.

object. A named storage space that consists of a set of characteristics that describe itself and, in some cases, data. An object is anything that exists in and occupies space in storage and on which operations can be performed. Some examples of objects are programs, files, libraries, and folders.

object authority. A specific authority that controls what a system user can do with an entire object. For example, object authority includes deleting, moving, or renaming an object. There are three types of object authorities: object operational, object management, and object existence.

object description. The characteristics (such as name, type, and owner name) that describe an object.

object existence authority. An object authority that allows the user to delete the object, free storage of the object, save and restore the object, transfer ownership of the object, and create an object that was named by an authority holder.

object name. The name of an object.

object operational authority. An object authority that allows the user to look at the description of an object and use the object as determined by the user's data authorities to the object.

object owner. A user who creates an object or to whom the ownership of an object was reassigned. The object owner has complete control over the object.

offline. Pertaining to the operation of a functional unit that is not under the continual control of the system. Contrast with *online*.

online. Pertaining to the operation of a functional unit that is under the continual control of the system. Contrast with *offline*.

online information. Information, read on the display screen, that explains displays, messages, and programs.

only field. A field that must contain only double-byte characters.

open. The function that connects a file to a program for processing. Contrast with *close*.

open data path (ODP). The path that handles all input/output operations for the file.

operating system. A collection of system programs that control the overall operation of a computer system.

Operating System/400 (OS/400). The operating system used by the AS/400 system.

operation. The result of processing statements in a high-level language.

output. Information or data received from a computer that is shown on a display, printed on the printer, or stored on disk, diskette, or tape.

output file. A database or device file that has been opened to allow records to be written.

overstrike. Pertaining to a character or symbol that occupies the same space as another character or symbol.

owner. The user who creates an object (or is named the owner of an object).

page format. The defined arrangement of lines on a page. Compare with *line format*.

parameter. A value supplied to a command or program that is used either as input or controls the actions of the command or program.

personal mail. Mail that can be accessed only by the receiver, but not by someone working on behalf of the receiver. When mail is sent, it can be assigned the classification personal.

physical file. A description of how data is to be presented to or received from a program and how data is actually stored in the database. A physical file contains one record format and one or more members. Contrast with *logical file*.

physical file member. A named subset of the data records in a physical file. See also *member*.

point. The second byte of a DBCS code, which uniquely identifies double-byte characters in the same ward.

pool. A division of main or auxiliary storage.

primary file. (DDS) In a join logical file, the first physical file specified on the JFILE keyword. Contrast with *secondary file*.

printer file. A device file created by the user to describe a printer device.

processing. The action of performing operations and calculations on data.

processing unit. The part of the system that performs instructions and contains main storage.

profile. Data that describes the characteristics of a user, program, device, or remote location.

program message queue. An object used to hold messages that are sent between program calls of a routing step. The program message queue is part of the job message queue.

program name. A user-defined word that identifies a COBOL source program.

program-described file. A file for which the fields in the records are described only in the program that processes the file. To the operating system, the record appears as a character string. Contrast with *externally described file*.

prompt. A list of values or a request for information provided by the system as a reminder of the type of information or action required.

public authority. The authority given to users who do not have any specific (private) authority to an object, who are not on the authorization list (if one is specified for the object), and whose group profile has no specific authority to the object.

Query. The shortened name for the IBM AS/400 Query licensed program.

query. A request to select and copy from a file or files one or more records based on defined conditions. For example, a request for a list of all customers in a customer master file, whose balance is greater than $1,000.

queue. A list of messages, jobs, or files waiting to be read, processed, printed, or distributed in the order they appear in the list.

random processing. A method of processing in which records can be read from, written to, or deleted from a file order requested by the program that is using them. See also *sequential processing*.

read operation. An input operation that obtains a record from a file and passes it to a program.

rebuild maintenance. A method of maintaining keyed access paths for database files. This method updates the access path only while the file is open, not when the file is closed; the access path is rebuilt when the file is opened. Contrast with *immediate maintenance* and *delayed maintenance*.

receiver. See *journal receiver*.

recipient. The user to whom mail is sent.

record. A collection of related data or words, treated as a unit; such as one name, address, and telephone number.

record format. (IDDU and DDS) The arrangement of data in the records contained in, or processed by, a file. The record format includes the record name, field names, and field descriptions (such as length and data type). The record formats used in a file are contained in the file description (DDS) or the record format definition (IDDU).

record format definition. (IDDU and DDS) A description of the characteristics of fields (for example, type and length) and the arrangement of the fields in a record created by the user.

record selection. The process of selecting particular records from a file and including the information from the records, for example, in a report.

record type. The classification of records in a file. Records of the same type have the same fields in the same order. For program-described files, these records have record identification codes; for externally described files, the records have the same record format name.

RECORD type. (Pascal) The structured type that contains a series of fields. Each field may be of a type different from the other fields in the record.

recovery. The process of rebuilding databases after a system failure.

relative file number. In a join logical file, a sequential number assigned to a physical file based on the position of that file on the JFILE keyword specification.

relative record number. A number that specifies the location of a record in relation to the beginning of a database file, member, or subfile. For example, the first record in a database file, member, or subfile has a relative record number of 1.

remote. Pertaining to a device, system, or file that is connected to another device, system, or file through a communications line. Contrast with *local*.

remote system. Any other system in the network with which your system can communicate.

required carrier return. A manually entered carrier return used after a short line or at the end of a paragraph to ensure that no more text will be added to the line during automatic line adjustment.

resident. Remaining in main storage.

resolve. In programming, to change a predefined, symbolic value to the actual value of the item being processed. For example, a symbolic value of *LAST defined for the name of a file member is resolved to the name of the last member when the member is processed.

resource. Any part of the system required by a job or task, including main storage, devices, the processing unit, programs, files, libraries, and folders.

restore. To copy data from tape, diskette, or a save file to auxiliary storage. Contrast with *save*.

result field. A field that contains the results of calculations performed on numeric fields in a file.

return code. In data communications, a value sent by the system to a program to indicate the results of an operation by that program.

return indicator. An indicator to an RPG/400 program that control should be returned to the calling program.

revisable-form text (RFT). A data stream defined by document content architecture that is used to exchange unresolved documents (which cannot be directly printed or displayed) between systems. Contrast with *final-form text (FFT)*.

roll back. To remove changes that were made to files under commitment control since the last commitment boundary.

rollback. The process of restoring data changed by an application program or user to the state at its last commit point.

routing. The list of users who are to receive an item when it is distributed, including all users named specifically and those users named on distribution lists by the sender.

RPG. Report Program Generator. A programming language designed for writing application programs for business data processing requirements. The application programs range from report writing and inquiry programs to applications such as payroll, order entry, and production planning.

RPG/400. An IBM licensed program that is the SAA RPG programming language available on the AS/400 system, including system-specific functions.

save. To copy specific objects, libraries, or data by transferring them from main or auxiliary storage to magnetic media such as tape, diskettes, or a save file. Contrast with *restore*.

save file. A file allocated in auxiliary storage that can be used to store saved data on disk (without requiring diskettes or tapes), that can be used in I/O operations from a high-level language program, or can be used to receive objects sent through the network.

secondary file. (DDS) In a join logical file, any physical file, other than the first physical file, that is specified on the JFILE keyword. Contrast with *primary file*.

secure. Controlling who can use and to what extent an object can be used by controlling the authority given to the user.

select/omit field. A field in a logical file record format whose value is tested by the system to determine if records including that field are to be used. The test is a comparison with a constant, the contents of another field, a range of values, or a list of values; and the record is either selected or omitted as a result of the test. See also *dynamic select/omit*.

selector. The term in a CASE statement that when evaluated determines which of the possible branches of the CASE statement will be processed.

sequence. To arrange in order.

sequence number. (1) The number of a record that identifies the record within the source member. (2) A field in a journal entry that contains a number assigned by the system. This number is initially 1 and is increased by 1 until the journal is changed or the sequence number is reset by the user.

sequential processing. A method of processing in which records are read, written to, or deleted in the order determined by the value of the key field. See also *random processing*.

SEU. See *source entry utility (SEU)*.

shared file. A file whose open data path can be shared between two or more programs processing in the same routing step. See *open data path*.

shift. A keyboard action to allow uppercase or other characters to be entered.

shift-out character. A control character (hex 0E) that indicates the start of a string of double-byte characters.

source entry utility (SEU). A function of the AS/400 Application Development Tools licensed program that is used to create and change source members.

source file. (1) A file of programming code that is not compiled into machine language. Contrast with *data file*. (2) A file created by the specification of FILETYPE(*SRC) on the Create command. A source file can contain source statements for such items as high-level language programs and data description specifications.

source member. A member of a database source file that contains source statements such as RPG/400, COBOL, BASIC, PL/I, or DDS statements. See also *member*.

source statement. A statement written in symbols of a programming language. For example, RPG/400, COBOL, BASIC, PL/I, and DDS statements are source statements.

specific authority. The types of authority a user can be given to use the system resources, including object authorities and data authorities.

spooled file. A file that holds output data waiting to be printed, or input data waiting to be processed by the program.

spooling. The system function that saves data in a disk file for later processing or printing.

SQL/400. See *Structured Query Language/400 (SQL/400)*.

statement. An instruction in a program.

storage pool. A logical division of storage reserved for processing a job or group of jobs.

store. To put or keep data in a storage device.

string. A group of auxiliary storage devices connected in a series on the system. The order and location in which each device is connected to the system determines the physical address of the device.

structure. A collection of data items that need not have identical attributes.

Structured Query Language/400 (SQL/400). An IBM licensed program that is the SAA version of SQL.

substring. A part of a character string.

subsystem. An operating environment, defined by a subsystem description, where the system coordinates processing and resources.

synchronization (SYN) character. In binary synchronous communications, the transmission control character that provides a signal to the receiving station for timing the characters received.

syntax. The rules for constructing a command or statement.

system group. The second part of a system name in the system distribution directory.

system name. An IBM-supplied name that uniquely identifies the system. It is used as a network value for certain communications applications such as APPC.

system object. One of two machine object classifications. Any of the machine objects shipped with the system or any of the operating system objects created by the system.

system profile. The text profile named system that contains formatting and editing options to be used for creating documents.

system resources. Those items controlled by the system, such as programs, devices, and storage areas that are assigned for use in jobs.

system value. Control information for the operation of certain parts of the system. A user can change the system value to define his working environment. System date and library list are examples of system values.

Systems Application Architecture (SAA). An architecture defining a set of rules for designing a common user interface, programming interface, application programs, and communications support for strategic operating systems such as OS/2, OS/400, VM/370, and MVS/370.

table. (SQL) A named data object consisting of a specific number of columns and some number of unordered rows.

tape file. A device file created by the user to support a tape device.

target. (DDM) The remote system where the request for a file is sent.

target system. In a distributed data management (DDM) network, the system that receives a request from an application program on another system to use one or more files located on the target system.

track. A circular path on the surface of a disk, diskette, or tape on which information is magnetically recorded and from which recorded information is read.

translation table. (1) A system table that provides replacement characters for characters that cannot be printed. (2) An object that contains a set of hexadecimal characters used to translate one or more characters of data. For example, unprintable characters can be translated to blanks, and lowercase alphabetic characters can be translated to uppercase characters. The system-recognized identifier for the object type is *TBL.

type. One of several Pascal data classes that define the permissible values that can be assigned to a variable.

unit. An independently compilable piece of code. The two types of units in AS/400 Pascal are segment units and program units.

update operation. An I/O process that changes the data in a record.

user ID. See *user identification (user ID)*.

user ID/address. The two-part network name used in the system distribution directory and in the office applications to uniquely identify a user and send electronic mail.

user identification (user ID). The name used to associate the user profile with a user when a user signs on the system.

user profile. An object with a unique name that contains the user's password, the list of special authorities assigned to a user, and the objects the user owns.

validity checking. To verify the contents of a field.

value. The smallest unit of data manipulated by the Structured Query Language.

variable. A name used to represent data whose value can be changed while the program is running by referring to the name of the variable.

view. An alternative representation of data from one or more tables. A view can include all or some of the columns contained in the table or tables on which it is defined.

volume. A storage medium that is put on or taken off the system as a unit, for example, magnetic tape or diskette.

write operation. An output operation that sends a processed record to an output device or output file.

writing. The action of making a permanent or temporary recording of data in a storage device or on a data medium.

Index

Application System/400™ Programming:
Database Guide

SC21-9659-1

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

□ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

□ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): Comment(s):

Please contact your IBM representative or your IBM-approved remarketer to request additional publications.

Name ____________________

Company or
Organization ____________________

Address ____________________

____
City State Zip Code

Phone No. ____________________
Area Code

No postage necessary if mailed in the U.S.A.

Cut or
Along

Fold and Tape

Please do not staple

Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
3605 North Hwy 52
ROCHESTER MN 55901-9986

Fold and Tape

Please do not staple

Fold and Tape

IBM®

Cut or Fc
Along Lir

Application System/400™ Programming:
Database Guide

SC21-9659-1

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

□ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

□ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): Comment(s):

Please contact your IBM representative or your IBM-approved remarketer to request additional publications.

Name

Company or
Organization

Address

City State Zip Code

Phone No.

Area Code

No postage necessary if mailed in the U.S.A.

Cut or
Along l

Fold and Tape

Please do not staple

Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
3605 North Hwy 52
ROCHESTER MN 55901-9986

Fold and Tape

Please do not staple

Fold and Tape

IBM®

Cut or F
Along Li

Program Number
5728-SS1

21F2721

SC21-9659-1

Printed in U.S.A.