IBM

System i

Database
Database programming

Version 6 Release 1

IBM

System i

Database
Database programming

Version 6 Release 1

> **Note**
>
> Before using this information and the product it supports, read the information in "Notices," on page 293.

Contents

Database programming

DB2® for i5/OS® provides a wide range of support for setting up, processing, and managing database files.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 290.

What's new for V6R1

Read about new or significantly changed information for the Database programming topic collection.

Journaling for libraries

You can now journal libraries and, optionally (based on your definition of the inheritance rules), journal database files that are created, moved, or restored into the libraries. However, if a journaled library contains the QDFTJRN data area, files that are created, moved, or restored into the library automatically start journaling based on the rules defined in QDFTJRN. For more information about journaling libraries and the QDFTJRN data area, see "Starting or ending journaling for libraries" on page 232 and "Implicit physical file journaling" on page 38.

Enhancements to the database programming information

Several topics in the Database programming topic collection have been enhanced this release:
- References to floating point are modified where appropriate. The term *floating point* comprises both binary floating point and decimal floating point unless otherwise specified.
- The types of authority that you need to read or change data in a physical file through a logical file have been added to the "Using logical files to secure data" on page 97 topic, along with an example that illustrates how logical files can be used to secure data.
- Some subtopics in the "Setting up a join logical file" on page 58 topic have been merged to eliminate redundant and scattering information.
- The subtopics in the "Using Open Query File (OPNQRYF) command" on page 123 topic have been reorganized to improve retrievability.
- A trigger program example has been added to the "Creating trigger programs" on page 268 topic in place of the original trigger program examples. This example of external triggers was originally in the SQL programming topic collection.

How to see what's new or changed

To help you see where technical changes have been made, the information center uses:
- The » image to mark where new or changed information begins.
- The « image to mark where new or changed information ends.

In PDF files, you might see revision bars (|) in the left margin of new and changed information.

To find other information about what's new or changed this release, see the Memo to users.

PDF file for Database programming

You can view and print a PDF file of this information.

To view or download the PDF version of this document, select Database programming (about 4300 KB).

Saving PDF files

To save a PDF on your workstation for viewing or printing:
1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe® Reader installed on your system to view or print these PDFs. You can download a free

copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

> **Related reference**
>
> "Related information for Database programming" on page 289
> Product manuals and other information center topic collections contain information that relates to the
> Database programming topic collection. You can view or print any of the PDF files.

Database file concepts

This introduction to i5/OS database files includes information about DB2 for i5/OS interfaces to database files, the types and maximum sizes of database files, and the ways of describing and protecting database files.

DB2 for i5/OS

DB2 for i5/OS is the integrated relational database manager on the i5/OS operating system.

DB2 for i5/OS is part of the i5/OS operating system. It provides access to and protection for data. It also provides advanced functions such as referential integrity and parallel database processing.

With DB2 for i5/OS, independent auxiliary storage pools (ASPs), also known as independent disk pools, allow you to have one or more separate databases associated with each ASP group. You can set up databases using primary independent disk pools.

> **Related concepts**
>
> Independent disk pools examples

Interfaces to DB2 for i5/OS

DB2 for i5/OS provides several independent interfaces to the database.

Traditional system interface
The i5/OS traditional system interface is the full set of system commands and other non-SQL facilities that you can use to access and change DB2 for i5/OS data.

The traditional system interface provides control language (CL) commands to create and manage database objects. The system interface also has an integrated facility for describing data called data description specifications (DDS).

The IBM® WebSphere® Development Studio for System i™ licensed program provides several utilities to describe and process data. The data file utility (DFU) can add, change, and delete data in a database file that is described by RPG, DDS, and the interactive data description utility (IDDU). The source entry utility (SEU) can specify and change data in files.

SQL

Structured Query Language (SQL) is a standardized language that can be used within host programming languages or interactively to define, access, and manipulate data in a relational database.

SQL uses a relational model of data; that is, it perceives all data as existing in tables. The DB2 for i5/OS database has SQL processing capability integrated into the system. It processes compiled programs that contain SQL statements. To develop SQL applications, you need the IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program for the system on which you develop your applications.

Interactive SQL is a function of the IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program that allows SQL statements to run dynamically instead of in batch mode. Every interactive SQL statement is read from the workstation, prepared, and run dynamically.

> **Related concepts**
>
> SQL programming

System i Navigator

System i Navigator is a no-charge feature of the IBM System i Access for Windows® licensed program. It provides a graphical, Microsoft® Windows interface to common i5/OS management functions, including database.

Most database operations that you can access using System i Navigator are based on SQL functions. However, some operations are based on the traditional system interface, such as control language (CL) commands.

> **Related concepts**
>
> Connecting to System i

IBM Query for i5/OS

The IBM Query for i5/OS licensed program can be used to select, format, and analyze information from database files to produce reports and other files.

Database files

A database file is one of the several types of the system object type *FILE. A database file 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.

Database files contain members and records.

Source file

A source file contains uncompiled programming code and input data needed to create some types of objects. It can contain source statements for such items as high-level language programs and data description specifications (DDS). A source file can be a source physical file, diskette file, tape file, or inline data file.

Physical file

A physical file is a database file that stores application data. It contains 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 consists of fixed-length records that can have variable-length fields. It contains one record format and one or more members. From the perspective of the SQL interface, physical files are identical to tables.

Logical file

A logical file is a database file that logically represents one or more physical files. It contains 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.

Logical files let users access data in a sequence and format that are different from the physical files they represent. From the perspective of the SQL interface, logical files are identical to views and indexes.

Member

Members are different sets of data, each with the same format, within one database file. Before you perform any input or output operations on a file, the file must have at least one member.

As a general rule, a database file has only one member, the one created when the file is created. If a file contains more than one member, each member serves as a subset of the data in the file.

Record

A record is a group of related data within a file. From the perspective of the SQL interface, records are identical to rows.

Related concepts

"Why source files are used" on page 14
A source file contains input (source) data that is needed to create some types of objects. A source file is used when a command alone cannot provide sufficient information for creating an object.

How database files are described

Records in database files can be described to the field or record level.

- Field-level description. The fields in the record are described to the system. For each field you can describe the name, length, data type, and validity checks. You can also add a 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.

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.

Externally and program-described data

Programs can use either externally described or program-described files.

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 data is called externally described data.
- 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.

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 system.

RBAFO500-0
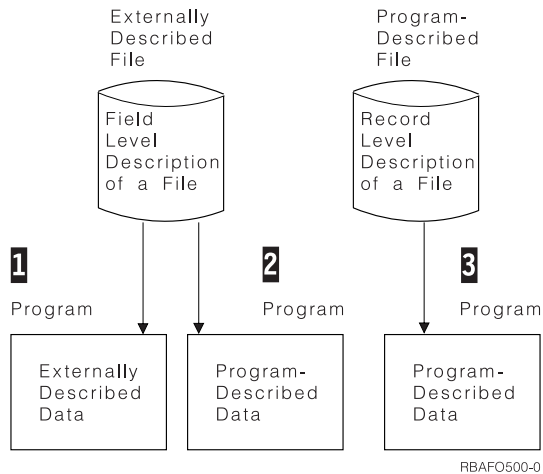
1 Externally Described Data

      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 Program-Described Data

      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 Program-Described Data

      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 a system that originally came from a traditional file system. Those programs use program-described data, and the file is described only to the record level. Later, you describe the file to the field level (externally described file) to use more of the database functions that are available on the system. Your old programs that contain 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

You can define a program-described or an externally described file with the record format description that is stored in the data dictionary.

You can describe the record format information using the interactive data definition utility (IDDU). Even though the file is program described, IBM Query for i5/OS, System i Access, and the data file utility (DFU) use the record format description that is stored in the data dictionary.

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

Record format description

When you describe a database file to the system, you describe two major parts of the file: the record format and the access path. 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
| ←→ || ←――――――――――――→ || ←→ |
35406HAMMER               01486
92201SCREWDRIVER          00649
```

RBAFO501-0

In this example of specifications for record format ITMMST, there are three fields. Field *ITEM* is zoned decimal, 5 digits, with no decimal position. Field *DESCRP* is character, with 18 positions. Field *PRICE* is zoned decimal, 5 digits, with two decimal positions.

A physical file can have only one record format. The record format in a physical file describes the way the data is actually stored.

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 can 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 that are 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 can describe a field *FLDA* as a packed decimal field of 5 digits, and a logical file that uses *FLDA* might redefine it as a zoned decimal field of 7 digits. In this case, when your program uses the logical file to read a record, the system automatically converts (unpacks) *FLDA* to zoned decimal format.

Access path description

An access path of a database file describes the order in which records are to be retrieved. When you describe an access path, you describe whether it is a keyed sequence access path or an arrival sequence access path.

Related concepts

"Describing access paths for database files" on page 83
An access path describes how records in a database file are retrieved. You can define the access path for a database file in various ways.

Naming conventions for a database file

The file name, record format name, and field name can be as long as 10 characters and must follow all system naming conventions. Some high-level languages have more restrictive naming conventions than the system has.

For example, the RPG/400® language 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. For more information about renaming database fields in programs, see your high-level language topic collection.

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.

Database data protection and monitoring

To ensure data integrity and consistency, you can enforce either business rules or data type rules.

You can enforce business rules using the following methods:
- Referential constraints let you put controls (constraints) on data in files you define as having a mutual dependency. A referential constraint lets you specify rules to be followed when changes are made to files with constraints.
- Triggers let you run your own program to take any action or evaluate changes when files are changed. When predefined changes are made or attempted, a trigger program is run.

The system performs data type checking in certain instances to ensure, for example, that data in a numeric field is really numeric.

In addition, the system protects data from loss using the following methods:
- Journaling and commitment control functions
- System-managed access path protection (SMAPP) support

   Related concepts

   "Ensuring data integrity with referential constraints" on page 255
   You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

   "Triggering automatic events in your database" on page 267
   A *trigger* is a set of actions that run automatically when a specified change or read operation is performed on a specified database file. You can define a set of trigger actions in any high-level language that is supported on the i5/OS operating system.

   "Recovering and restoring your database" on page 226
   You can use several i5/OS save and restore functions to recover your database after the system loses data.

Database file sizes

Before you design and create a database file, you need to know the maximum size allowed for the file.

The following table lists the maximum values for database files.

Description	Maximum value
Number of bytes in a record	32 766 bytes
Number of fields in a record format	8 000 fields

Description	Maximum value
Number of key fields in a file	120 fields
Size of key for physical and logical files	32 768 characters[1]
Size of key for ORDER BY (SQL) and KEYFLD (OPNQRYF)	10 000 bytes
Number of records contained in a file member	4 294 967 294 records[2]
Number of bytes in a file member	1 869 162 846 624 bytes[3]
Number of bytes in an access path	1 099 511 627 776 bytes[3] [5]
Number of keyed logical files built over a physical file member	3686 files
Number of physical file members in a logical file member	32 members
Number of members that can be joined	256 members
Size of a character or DBCS field	32 766 bytes[4]
Size of a zoned decimal or packed decimal field	63 digits
Maximum number of distinct database files that can be in use at one time	~500 000
Maximum number of members in a physical or logical file	32 767
Maximum number of constraints per physical file	300 constraints
Maximum number of triggers per physical file	300 triggers
Maximum number of recursive insert and update trigger calls	200

[1] When a first-changed-first-out (FCFO) access path is specified for the file, the maximum value for the size of the key for physical and logical files is 32 763 for ACCPTHSIZ(*MAX1TB) and 1995 characters for ACCPTHSIZ(*MAX4GB).

[2] For files with keyed sequence access paths, the maximum number of records in a member varies and can be estimated using the following formulas.

When ACCPTHSIZ(*MAX4GB) is specified, use the following formula:

$$\frac{2{,}867{,}200{,}000}{10 + (.8 \times \text{key length})}$$

When ACCPTHSIZ(*MAX1TB) is specified, use the following formula:

$$\frac{725{,}680{,}000{,}000}{12 + (.8 \times \text{key length})}$$

These are estimated values. The actual maximum number of records can vary significantly.

[3] Both the number of bytes in a file member and the number of bytes in an access path must be looked at when message CPF5272 is sent indicating that the maximum system object size has been reached.

[4] The maximum size of a variable-length character or DBCS field is 32 740 bytes. DBCS-graphic field lengths are expressed in terms of characters; therefore, the maximums are 16 383 characters (fixed length) and 16 370 characters (variable length).

[5] The maximum is 4 294 966 272 bytes if the access path is created with a maximum size of 4 gigabytes (GB), ACCPTHSIZ(*MAX4GB).

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 i5/OS database file can grow until it reaches the maximum size allowed on the operating system. The operating system normally does not allocate all the file space at once. Rather, it occasionally allocates 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) that does not contain null-capable fields:

  Disk size = (number of valid and deleted records + 1) x
  　　　　　(record length + 1) + 20480 x (number of members) + 8192

  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 physical file (excluding the access path) that contains null-capable fields:

  Disk size = (number of valid and deleted records + 1) x
  　　　　　　　(record length + 1) + 20480 x (number of members) +
  　　　　　　　8192 + ((number of fields in format ÷ 8) rounded up) x
  　　　　　　　(number of valid and deleted records + 1)

  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 = (12288) x (number of members) + 8192
- For a keyed sequence access path the generalized equation for index size, per member, is:

  let a = (LimbPageUtilization - LogicalPageHeaderSize) *
  　　　　(LogicalPageHeaderSize - LeafPageUtilization - 2 * NodeSize)

  let b = NumKeys * (TerminalTextPerKey + 2 * NodeSize) *
  　　　　(LimbPageUtilization - LogicalPageHeaderSize + 2 * NodeSize)
  　　　　+ CommonTextPerKey * [LimbPageUtilization + LeafPageUtilization
  　　　　- 2 * (LogicalPageHeaderSize - NodeSize)]
  　　　　- 2 * NodeSize * (LeafPageUtilization - LogicalPageHeaderSize
  　　　　+ 2 * NodeSize)

  let c = CommonTextPerKey * [2 * NodeSize - CommonTextPerKey
  　　　　- NumKeys * (TerminalTextPerKey + 2 * NodeSize)]

  then NumberLogicalPages = ceil([-b - sqrt(b ** 2 - 4 * a * c)]
  　　　　　　　　　　　/ (2 * a))

and TotalIndexSize = NumberLogicalPages * LogicalPageSize

This equation is used for both three and four byte indexes by changing the set of constants in the equation as follows.

Constant	Three-byte index	Four-byte index
NodeSize	3	4
LogicalPageHeaderSize	16	64
LimbPageUtilization	.75 * LogicalPageSize	.75 * LogicalPageSize
LeafPageUtilization	.75 * LogicalPageSize	.80 * LogicalPageSize

The remaining constants, CommonTextPerKey and TerminalTextPerKey, are probably best estimated by using the following formulas:

$$
\begin{aligned}
\text{CommonTextPerKey} = [\ & \min(\max(\text{NumKeys} - 256,0),256) \\
& + \min(\max(\text{NumKeys} - 256 * 256,0),256 * 256) \\
& + \min(\max(\text{NumKeys} - 256 * 256 * 256,0), \\
& \quad 256 * 256 * 256) \\
& + \min(\max(\text{NumKeys} - 256 * 256 * 256 * 256,0), \\
& \quad 256 * 256 * 256 * 256)\ ] \\
& * (\text{NodeSize} + 1)\ /\ \text{NumKeys}
\end{aligned}
$$

TerminalTextPerKey = KeySizeInBytes - CommonTextPerKey

This should reduce everything needed to calculate the index size to the type of index (that is, 3 or 4 byte), the total key size, and the number of keys. The estimate should be greater than the actual index size because the common text estimate is minimal.

Given this generalized equation for index size, the LogicalPageSize is as follows.

Table 1. LogicalPageSize values

Key Length	*MAX4GB (3-byte) LogicalPageSize	*MAX1TB (4-byte) LogicalPageSize
1 - 500	4096 bytes	8192 bytes
501 - 1000	8192 bytes	16 384 bytes
1001 - 2000	16 384 bytes	32 768 bytes
2001 - 10 000	N/A	65 536 bytes
10 001 - 18 000	N/A	131 072 bytes
18 001 - 26 000	N/A	262 144 bytes
26 001 - 32 768	N/A	524 288 bytes

The LogicalPageSizes in Table 1 generate the following LimbPageUtilizations.

Key Length	*MAX4GB (3-byte) LimbPageUtilization	*MAX1TB (4-byte) LimbPageUtilization
1 - 500	3072 bytes	6144 bytes
501 - 1000	6144 bytes	12 288 bytes
1001 - 2000	12 288 bytes	24 576 bytes
2001 - 10 000	N/A	49 152 bytes
10 001 - 18 000	N/A	98 304 bytes
18 001 - 26 000	N/A	196 608 bytes
26 001 - 32 768	N/A	393 216 bytes

The LogicalPageSizes in Table 1 on page 10 generate the following LeafPageUtilizations.

Key Length	*MAX4GB (3-byte) LeafPageUtilization	*MAX1TB (4-byte) LeafPageUtilization
1 - 500	3072 bytes	6554 bytes
501 - 1000	6144 bytes	13 107 bytes
1001 - 2000	12 288 bytes	26 214 bytes
2001 - 10 000	N/A	52 428 bytes
10 001 - 18 000	N/A	104 857 bytes
18 001 - 26 000	N/A	209 715 bytes
26 001 - 32 768	N/A	419 430 bytes

Then to simplify the generalized equation for index size, let:

CommonTextPerKey = 0

which would cause:

TerminalTextPerKey = KeySizeInBytes

b = NumKeys * (KeySizeInBytes + 2 * NodeSize) *
    (LimbPageUtilization - LogicalPageHeaderSize + 2 * NodeSize)
    - 2 * NodeSize * (LeafPageUtilization - LogicalPageHeaderSize
    + 2 * NodeSize)

c = 0

NumberLogicalPages = ceil([-b - sqrt(b ** 2)]
                          / (2 * a))
                   = ceil[(-2 * b) / (2 * a)]
                   = ceil[-b/a]

Example: Database file sizes

This example shows how to estimate the maximum size of a database file.

A *MAX1TB (4-byte) access path with 120 byte keys and 500 000 records TotalIndexSize has a TotalIndexSize in bytes as follows:

```
a = (LimbPageUtilization - LogicalPageHeaderSize) *
    (LogicalPageHeaderSize - LeafPageUtilization - 2 * NodeSize)
  = (6144 - 64) *
    (64 - 6554 - 2 * 4)
  = 6080 * -6498
  = -39,507,840

b = NumKeys * (KeySizeInBytes + 2 * NodeSize) *
    (LimbPageUtilization - LogicalPageHeaderSize + 2 * NodeSize)
    - 2 * NodeSize * (LeafPageUtilization - LogicalPageHeaderSize
    + 2 * NodeSize)
  = 500,000 * (120 + 2 * 4) *
    (6144 - 64 + 2 * 4)
    - 2 * 4 * (6554 - 64 + 2 * 4)
  = 500,000 * 128 *
    6088
    - 8 * 6498
  = 3.896319e+11

NumberLogicalPages = ceil[ -b/a ]
                   = ceil[ -3.896319e+11/-39507840 ]
```

```
                    = 9863

TotalIndexSize = NumberLogicalPages * LogicalPageSize
               = 9863 * 8192
               = 80,797,696 bytes
```

The equation for index size in previous versions of the operating system produces the following result:

```
TotalIndexSize = (number of keys) * (key length + 8) *
                 (0.8) * (1.85) + 4096
               = (NumKeys) * (KeySizeInBytes + 8) *
                 (0.8) * (1.85) + 4096
               = 500000 * 128 *
                 .8 * 1.85 + 4096
               = 94,724,096
```

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 table shows a list of minimum file sizes.

Description	Minimum size
Physical file without a member	8192 bytes
Physical file with a single member	20 480 bytes
Keyed sequence access path	12 288 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 = (144 x number of fields) + 4096

- For files sharing their format with any other file:

  Format sharing directory size = (16 x number of files sharing the format) + 4096

- 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 x number of files or members sharing data) + 4096

- For each file member having a logical file member sharing its access path:

  Access path sharing directory size = (16 x number of files or members sharing access path) + 4096

Setting up database files

You can create, define, and secure a database file using the traditional system interface or System i Navigator.

Related concepts

"Traditional system interface" on page 2
The i5/OS traditional system interface is the full set of system commands and other non-SQL facilities that you can use to access and change DB2 for i5/OS data.

Getting started with System i Navigator

Creating and describing database files

You can create and describe a library and a database file using the traditional system interface.

The system supports several methods for creating and describing a database file:

* Interactive data definition utility (IDDU)

  You can create a database file by using IDDU, part of the IBM WebSphere Development Studio for System i licensed program. If you are using IDDU to describe your database files, you might also consider using it to create your files.

* Control language (CL), using the source entry utility (SEU) or the data file utility (DFU) to specify data description specifications (DDS)

  You can create a database file by using CL. The CL database file create commands are Create Physical File (CRTPF), Create Logical File (CRTLF), and Create Source Physical File (CRTSRCPF). After a database file is created, you can use SEU or DFU to describe data in the file. SEU and DFU are part of the IBM WebSphere Development Studio for System i licensed program. These topics focus on how to create files using these methods.

* Structured Query Language (SQL)

  You can create and describe a database file (table) by using SQL statements. SQL is the IBM relational database language. It can be used to interactively describe and create database files.

* System i Navigator

  You can also create a database file (table) using System i Navigator.

  Related concepts

  Creating a table

  Getting started with System i Navigator

  SQL programming

  "Traditional system interface" on page 2
  The i5/OS traditional system interface is the full set of system commands and other non-SQL facilities that you can use to access and change DB2 for i5/OS data.

Creating a library

A *library* is a system object that serves as a directory to other objects. A library groups related objects and allows you to find objects by name. To create a library, use System i Navigator or the Create Library (CRTLIB) command.

The system-recognized identifier for the object type is *LIB. Before you can create a database file, you must create a library to store it. You can create a library in the following ways:

* You can use System i Navigator to create a library (in SQL, called a *schema*).
* You can use the Create Library (CRTLIB) command to create the library.

When creating a library, you can specify the auxiliary storage pool (ASP) where the library is to be stored. This allows you to create multiple, separate databases.

  Related tasks

  Creating a schema

  Related reference

  Create Library (CRTLIB) command

Setting up source files

You can either create a source file or use an IBM-supplied source file.

Related concepts

"Using source files" on page 243
DB2 for i5/OS provides a range of support for source files.

Why source files are used:

A source file contains input (source) data that is needed to create some types of objects. A source file is used when a command alone cannot provide sufficient information for creating an object.

For example, to create a control language (CL) program, you must use a source file that contains source statements in the form of commands. To create a logical file, you must use a source file that contains data description specifications (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

To create the following objects, source files can be used, but are *not* required:
- Physical files
- Display files
- Printer files
- Translate tables

A source file can be a database file, a diskette file, a tape file, or an inline data file. (An inline data file is included as part of a job.) A source database file is another type of database file. You can use a source database file as you use any other database file on the system.

Related concepts

"Database files" on page 3
A database file is one of the several types of the system object type *FILE. A database file 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.

Creating a source file:

Before creating a source file, you should first create a library. Then use the Create Source Physical File (CRTSRCPF), Create Physical File (CRTPF), or Create Logical File (CRTLF) command to create a source file.
- Create Source Physical File (CRTSRCPF) command

  Normally, you 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.
- Create Physical File (CRTPF), or Create Logical File (CRTLF) command

  If you want to create a source file and define the record format and fields using data description specifications (DDS), use the Create Physical File (CRTPF) or Create Logical File (CRTLF) command.

As an alternative to creating a source file, you can use source files supplied with the i5/OS and other licensed programs.

Related concepts

"Creating a library" on page 13

A *library* is a system object that serves as a directory to other objects. A library groups related objects and allows you to find objects by name. To create a library, use System i Navigator or the Create Library (CRTLIB) command.

Related reference

Create Physical File (CRTPF) command

Create Logical File (CRTLF) command

Creating a source file using the Create Source Physical File (CRTSRCPF) command:

You can create a source file using the default values of the Create Source Physical File (CRTSRCPF) command.

```
CRTSRCPF FILE(QGPL/FRSOURCE) TEXT('Source file')
```

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).

Related reference

Create Source Physical File (CRTSRCPF) command

Creating a source file with DDS:

If you want to create a source file with data description specifications (DDS), use the Create Physical File (CRTPF) or Create Logical File (CRTLF) command.

If you want to create a source file for which you need to define the record format, use the CRTPF or 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.

The following example shows the DDS needed to define the record format for a source file using the CRTPF command:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  A*   R RECORD1
  A          F1              6S 2
  A          F2              6S
  A          F3             92A
```

Related reference

Create Physical File (CRTPF) command

Create Logical File (CRTLF) command

Creating a source file without DDS:

When you create a source physical file without using DDS, but by specifying the record length (RCDLEN parameter) on the Create Source Physical File (CRTSRCPF) command, 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 that consists of the following three fields is automatically used for a source physical file that is created with the 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.

IBM-supplied source files:

For your convenience, the i5/OS licensed program and other licensed programs provide a database source file for each type of source.

This table shows these IBM-supplied source files.

File name	Library name	Used to create
QCBLLESRC	QGPL	ILE COBOL programs
QCBLSRC	QGPL	System/38™ compatible COBOL programs
QCLSRC	QGPL	CL programs
QCMDSRC	QGPL	Command definition statements
QCPPSRC	QGPL	C++ programs
QCSRC	QGPL	C programs
QDDSSRC	QGPL	Files definition statements
QFMTSRC	QGPL	Sort source
QLBLSRC	QGPL	COBOL/400® programs
QPLISRC	QGPL	PL/I programs
QPNLSRC	QGPL	Panel group (UIM) definition statements
QREXSRC	QGPL	REXX™ procedures
QRPGLESRC	QGPL	ILE RPG programs
QRPGSRC	QGPL	RPG/400 and System/38 compatible RPG programs
QS36PRC	#LIBRARY	System/36™ compatible COBOL and RPG II procedures
QS36SRC	#LIBRARY	System/36 compatible COBOL and RPG programs
QTBLSRC	QGPL	Translation tables
QTXTSRC	QGPL	Text

You can either add your source members to these files or create your own source files. Normally, you want to create your own source files using the same names as the IBM-supplied files, but in different libraries. The IBM-supplied source files are created with the file names used for the corresponding create command (for example, the Create CL Program (CRTCLPGM) command uses the QCLSRC file name as the default). Additionally, the IBM-supplied programmer menu uses the same default names. (If you use the same file names as the IBM-supplied names, ensure that the library containing your source files precedes the library containing the IBM-supplied source files in the library list.)

Source file attributes:

Here are the attributes common to most source files and the restrictions on using these 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 data description specifications (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.

Describing database files

You can use several methods to describe i5/OS database files. This topic discusses how to describe a database file with data description specifications (DDS) because DDS has the most options for defining data.

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: interactive data definition utility (IDDU), Structured Query Language (SQL) commands, or data description specifications (DDS).

Interactive data definition utility (IDDU)

> Physical files can be described with IDDU. You might use IDDU because it is a menu-driven, interactive method of describing data. You might be familiar with describing data using IDDU in a System/36 environment. In addition, IDDU allows you to describe multiple-format physical files for use with IBM Query for i5/OS, System i Access, and the data file utility (DFU).

> When you use IDDU to describe your files, the file definition becomes part of the i5/OS data dictionary.

DB2 for i5/OS Structured Query Language (SQL)

> SQL can be used to describe a database file. It supports statements to describe the fields in the database file and to create the file.

> SQL was created by IBM to meet the need for a standard and common database language. It is currently used on all IBM DB2 platforms and on many other database implementations from many different manufacturers.

When you create a database file using the DB2 for i5/OS SQL, the file description is automatically added to a data dictionary in the SQL schema. The data dictionary (or catalog) is then automatically maintained by the system.

SQL is the language of choice for accessing databases on many other platforms. It is the only language for distributed databases and heterogeneous systems.

Data description specifications (DDS)

Externally described files can be described with DDS. DDS provides descriptions of the field-level, record-level, 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 form provides a common format for describing data externally. DDS data is column sensitive. The examples that follow have numbered columns and show the data in the correct columns.

After a database file is described, you can view the description.

Related concepts

📄 IDDU Use PDF

SQL programming

DB2 for i5/OS SQL reference

"Displaying information about database files" on page 216
By using System i Navigator or CL commands, you can display various types of information about database files.

Describing database files using DDS:

When you describe a database file using data description specifications (DDS), you can describe the file at the file, record-format, join, field, key-field, and select/omit-field levels.

- File-level DDS provides 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 provides 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 provides the system information about the physical files that are used in a join logical file. For example, you can specify how to join two physical files.
- Field-level DDS provides 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 provides 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 provides the system information about which records are to be returned to the program during file processing. Select/omit specifications apply to logical files only.

Related concepts

DDS for physical and logical files

Example: Describing a physical file using DDS:

This example shows how to describe a physical file using DDS.

The DDS for a physical file, as shown in the next example, must be in the following order:

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.

5 Comment (optional).

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A* ORDER HEADER FILE (ORDHDRP)
    A  5
    A                                   1  UNIQUE
    A        2  R ORDHDR                    TEXT('Order header record')
    A           3  CUST         5   0       TEXT('Customer number')
    A              ORDER        5   0       TEXT('Order number')
    A              .
    A              .
    A              .
    A           K CUST
    A        4  K ORDER
```

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

Record format of physical file ORDHDRP

Customer number (CUST) — Packed decimal length 5, No decimals
Order number (ORDER) — Packed Decimal Length 5, No decimals
Order date (ORDATE) — Packed decimal length 6, No decimals
Purchase order number (CUSORD) — Packed decimal length 15, No decimals
Shipping instructions (SHPVIA) — Character length 15
Order status (ORDSTS) — Character length 1
...
State (STATE) — Character length 2

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    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 Order No.')
    A              SHPVIA      15            TEXT('Shipping Instr')
    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
```

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	Binary floating point
H	Hexadecimal
L	Date
T	Time
Z	Timestamp

Notes:

1. For double-byte character set (DBCS) data types, see "Double-byte character set considerations" on page 285.
2. The system performs arithmetic operations more efficiently for the packed-decimal than for the zoned-decimal data type.
3. Some high-level languages do not support binary floating-point data.
4. Here are some special considerations for using binary floating-point fields:
   - The precision associated with a binary floating-point field is a function of the number of bits (single or double precision) and the internal representation of the binary floating-point value. This translates into the number of decimal digits supported in the significand and the maximum values that can be represented in the binary floating-point field.
   - When a binary 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 binary 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 binary floating-point format, so that identical results are obtained when a binary 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.

Character or hexadecimal data can be defined as variable length by specifying the VARLEN field-level keyword. Generally you would use variable length fields, for example, as an employee name within a database. Names usually can be stored in a 30-byte field; however, there are times when you need 100 bytes to store a very long name. If you always define the field as 100 bytes, you waste storage. If you always define the field as 30 bytes, some names are truncated.

You can use the DDS VARLEN keyword to define a character field as variable length. You can define this field as:

- Variable-length with no allocated length. This allows the field to be stored using only the number of bytes equal to the data (plus two bytes per field for the length value and a few overhead bytes per record). However, performance might be affected because all data is stored in the variable portion of the file, which requires two disk read operations to retrieve.
- Variable-length with an allocated length equal to the most likely size of the data. This allows most field data to be stored in the fixed portion of the file and minimizes unused storage allocations common with fixed-length field definitions. Only one read operation is required to retrieve field data with a length less than the allocated field length. Field data with a length greater than the allocated length is stored in the variable portion of the file and requires two read operations to retrieve the data.

Example: Describing a logical file using DDS:

This example shows how to describe a logical file using DDS.

The DDS for a logical file, as shown in the next example, must be in the following order:

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 format-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 never see a record whose *OPNSTS* field contains an N value.

6   Comment.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A* ORDER HEADER FILE (ORDHDRP)
    A    6
    A                              1  UNIQUE
    A    2  R ORDHDR                  PFILE(ORDHDRP)
    A       3  ORDER                  TEXT('Order number')
    A          CUST                     TEXT('Customer number')
    A             .
    A             .
    A             .
    A    4  K ORDER
    A          O OPNSTS            5  CMP(EQ 'N')
    A          S                     ALL
```

A logical file must be created after all physical files on which it is based are created. The PFILE keyword in this 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.

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.

> **Related concepts**
>
> "Sharing existing record format descriptions in a database file" on page 25
> A record format can be described once in either a physical or a logical file (except a join logical file) and can be used by many files. When you describe a new file, you can specify that the record format of an existing file is to be used by the new file.
>
> "Setting up logical files" on page 39
> A *logical file* contains no data, but it defines record formats for one or more physical files. You can create various logical files and describe their record formats and access paths using data description specifications (DDS).
>
> Control language

Additional DDS 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 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).
- Content and default value keywords to control the null content and default data for a field. The ALWNULL keyword specifies whether a null value is allowed in the field. If ALWNULL is used, the default value of the field is null. If ALWNULL is not present at the field level, the null value is not allowed, character and hexadecimal fields default to blanks, and numeric fields default to zeros, unless the DFT (default) keyword is used to specify a different value.

Using existing field descriptions and field reference files to describe a database file:

If you want to use a field description in an existing file, you can copy that field description into your new file description. You can also create a field reference file to contain the field descriptions that you need for any group of files.

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 that field descriptions are used consistently. 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 to 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. The following example shows the DDS needed to describe DSTREFP.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    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       15         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('Cust purchase ord no.')
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                                       COLHDG('OPERATOR NAME')
A             ORDAMT        9  2        TEXT('Total order value')
A                                       COLHDG('ORDER' 'AMOUNT')
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('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 no. this item')
A                                       COLHDG('LINE' 'NO')
A             QTYORD        3  0        TEXT('Quantity ordered')
A                                       COLHDG('QTY' 'ORDERED'
A                                       CMP(GE 0)
A             EXTENS        6  2        TEXT('Ext 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      A
```

Assume that the DDS in the previous example is entered into a source file FRSOURCE; the member name is DSTREFP. To create a field reference file, use the CRTPF command as follows:

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

The parameter MBR(*NONE) tells the system not to add a member to the file (because the field reference file never contains data and therefore does not need a member).

To describe the physical file ORDHDRP by referring to DSTREFP, use the following DDS example:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  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
A
```

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 CRTPF command. Assume that the DDS in the previous example 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 topic collection refer to this field reference file.

Using a data dictionary for field reference in a database file:

You can use a data dictionary and the interactive data description utility (IDDU) as an alternative to using a DDS field reference file. IDDU allows you to define fields in a data dictionary.

   Related concepts

 IDDU Use PDF

Sharing existing record format descriptions in a database file:

A record format can be described once in either a physical or a logical file (except a join logical file) and can be used by many files. When you describe a new file, you can specify that the record format of an existing file is to be used by the new file.

Sharing existing record format descriptions can help reduce the number of DDS statements that you 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 example shows the DDS for two files. The first file describes a record format, and the second file shares the record format of the first file:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  A          R RECORD1               PFILE(CUSMSTP)
  A            CUST
  A            NAME
  A            ADDR
  A            SEARCH
  A          K CUST
  A
```

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          R RECORD1                    PFILE(CUSMSTP)
    A                                       FORMAT(CUSMSTL)
    A            K NAME
    A
```

The first example 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 the second example 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 the first example. 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. (In the Structured Query Language (SQL), a *view* is an alternative representation of data from one or more tables. It can include all or some of the columns contained in the table or tables on which it is defined.)

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.

Record format relationships between physical and logical files:

When you change, add, or delete fields with the Change Physical File (CHGPF) command, a certain relationship exists between the physical and logical files that share the same record format.
- When you change the length of a field in a physical file, you also change the length of the logical file's field.
- When you add a field to the physical file, the field is also added to the logical file.
- When you delete a field in the physical file, the field is also deleted from the logical file unless there is another dependency in DDS, such as a keyed field or a select/omit statement.

Record format sharing limitation with physical and logical files:

You might encounter this record format sharing limitation when you are duplicating the same database object multiple times.

A record format can only be shared by 32KB objects. Error messages are issued when you reach this limitation.

Note: Format sharing is performed for files that are duplicated. The format is shared up to 32 767 times. Beyond 32 767 times, if a file that shares the format is duplicated, a new format is created for the duplicated file.

Specifying database file and member attributes:

When you create a database file, database attributes are stored with the file and members. You specify attributes with database command parameters.

> **Related reference**
> Create Physical File (CRTPF) command
> Create Logical File (CRTLF) command
> Create Source Physical File (CRTSRCPF) command
> Add Physical File Member (ADDPFM) command
> Add Logical File Member (ADDLFM) command
> Change Physical File (CHGPF) command
> Change Physical File Member (CHGPFM) command
> Change Logical File (CHGLF) command
> Change Logical File Member (CHGLFM) command
> Change Source Physical File (CHGSRCPF) command

Specifying the file name and member name (FILE and MBR) parameters:

The FILE and MBR parameters specify the names of a database file and a file member.

You name a file with the FILE parameter on a create command. You also name the library where the file is stored. When you create a physical or logical file, the system normally creates a member with the same name as the file. You can, however, specify a member name with the MBR parameter on a create command. You can also choose not to create any members by specifying MBR(*NONE).

Note: The system does not automatically create a member for a source physical file.

Specifying the physical file data members (DTAMBRS) parameter:

The DTAMBRS parameter on the Create Logical File (CRTLF) command specifies the physical files and members that contain the data associated with a logical file member.

You can specify:
- The order in which the physical file members are to be read.
- The number of physical file members to be used.

> **Related concepts**
> "Defining logical file members" on page 43
> You can define members in a logical file to separate the data into logical groups. A logical file member can be associated with one or several physical file members.

Specifying the source file and source member (SRCFILE and SRCMBR) parameters:

The SRCFILE and SRCMBR parameters specify the names of the source file and the source file member that contain DDS statements.

If you do not specify a name:
- The default source file name is QDDSSRC.
- The default member name is the name specified on the FILE parameter.

Specifying the file type (FILETYPE) parameter:

The FILETYPE parameter specifies the type of database file.

A database file type is either data (*DATA) or source (*SRC). The Create Physical File (CRTPF) and Create Logical File (CRTLF) commands use the default data file type (*DATA).

Specifying the maximum number of members (MAXMBRS) parameter:

The MAXMBRS parameter specifies the maximum number of members that a database file can hold.

The default maximum number of members for physical and logical files is one, and the default for source physical files is *NOMAX.

Specifying the preferred storage unit (UNIT) parameter:

The system finds a place for a database file on auxiliary storage. The UNIT parameter specifies where to store the file.

Note: Effective for Version 3 Release 6 the UNIT parameter is a no-operation (NOP) function for the following commands:
- Create Physical File (CRTPF)
- Create Logical File (CRTLF)
- Create Source Physical File (CRTSRCPF)
- Change physical File (CHGPF)
- Change Logical File (CHGLF)
- Change Source Physical File (CHGSRCPF)

The parameter can still be coded; its presence does not cause an error. It will be ignored.

The UNIT parameter specifies:
- The location of data records in physical files.
- The access path for both physical files and logical files.

The data is placed on different units if:
- There is not enough space on the unit.
- The unit is not valid for your system.

An informational message indicating that the file was not placed on the requested unit is sent when file members are added. (A message is *not* sent when the file member is extended.)

UNIT parameter tips

In general, you should not specify the UNIT parameter. Let the system place the file on the disk unit of its choosing. This is usually better for performance, and relieves you of the task of managing auxiliary storage. If you specify a unit number and also an auxiliary storage pool, the unit number is ignored.

> **Related concepts**
> Independent disk pools examples

Specifying the force write ratio (FRCRATIO) parameter:

The FRCRATIO parameter on the create, change, or override database file command controls when database file changes are written to auxiliary storage.

Normally, the system determines when to write changed data from main storage to auxiliary storage. Closing the file (except for a shared close) and the force-end-of-data operation forces remaining updates, deletions, and additions to auxiliary storage. If you are journaling the file, the FRCRATIO parameter should normally be *NONE.

FRCRATIO parameter tip

Using the FRCRATIO parameter has performance and recovery considerations for your system.

> **Related concepts**
> "Recovering and restoring your database" on page 226
> You can use several i5/OS save and restore functions to recover your database after the system loses data.

Specifying the force keyed access path (FRCACCPTH) parameter:

The FRCACCPTH parameter controls whether access path changes are forced to auxiliary storage along with the associated records in a database file.

FRCACCPTH (*YES) forces access path changes to auxiliary storage whenever the access path is changed. This reduces the chance that the access path needs rebuilding if the system fails.

FRCACCPTH parameter tips

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.

> **Related concepts**
> "Recovering and restoring your database" on page 226
> You can use several i5/OS save and restore functions to recover your database after the system loses data.

Specifying the record format level check (LVLCHK) parameter:

When a database file is opened, the system can check whether the file definition is changed. The LVLCHK parameter specifies the check for record format description changes.

When the file changes to an extent that your program might not be able to process the file, the system notifies your program. The default is to do level checking. You can specify if you want level checking when you:
* Create a file.
* Use a change database file command.

You can override the system and ignore the level check using the Override with Database File (OVRDBF) command.

Example: Record format level check

Assume that you compiled your program two months ago. At that time, the file was defined as having three fields in each record. Last week another programmer decided to add a field to the record format, so now each record has four fields. When your program tries to open the file, the system notifies your program that a significant change has occurred to the file definition since the last time the program was compiled. This notification is known as a record format level check.

Specifying the access path maintenance (MAINT) parameter:

The MAINT parameter specifies the type of access path maintenance for all members of a database file.

If a file member is open, the system maintains the access path for the member when changes are made to the data in it. However, because more than one access path might exist for the same data, changing data in one file member might cause changes in the access paths for other file members that are not open (in use).

Here are the ways you can maintain access paths for all members of a database file:

- **Immediate** maintenance of an access path means that the access path is maintained for a file member whenever changes are made to its associated data, whether the member is open or not. Access paths used by referential constraints are always in immediate maintenance.
- **Rebuild** maintenance of an access path means that the access path is maintained for a file member only when the member is open, not when the member is closed; the access path is rebuilt when the member is opened the next time. When a file member with rebuild maintenance is closed, the system stops maintaining the access path. When the file member is opened again, the access path is totally rebuilt. If one or more programs have opened a specific file member with rebuild maintenance specified, the system maintains the access path for the member until the last user closes it.
- **Delayed** maintenance of an access path means that any maintenance of the access path is done after the file member is opened the next time and when it remains open. However, the access path is not rebuilt as it is with rebuild maintenance. Changes 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.

If you do not specify the type of maintenance for a file, the default is immediate maintenance.

MAINT parameter comparison:

This table compares the effect of the immediate, rebuild, and delayed maintenance of access paths on opening and processing files.

Function	Immediate maintenance	Rebuild maintenance	Delayed maintenance
Open	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.
Process	Slower update and output operations when many access paths with immediate maintenance are built over changing data (the system must maintain the access paths).	Faster update and 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 and 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.			

MAINT parameter tips:

The type of access path maintenance to specify depends on the number of records and the frequency of add, delete, or update operations on the file when it is closed.

You should use delayed maintenance for files that have relatively few changes to the access paths while the file members are closed. Delayed maintenance reduces system overhead by reducing the number of access paths that are maintained immediately. It might also result in faster open processing, because the access paths do not have to be rebuilt.

You might 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 might 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.

Specifying the access path recovery (RECOVER) parameter:

The RECOVER parameter specifies when changed access paths that are not journaled or forced to auxiliary storage are rebuilt after the system failure.

You can use the RECOVER parameter on the following commands to specify when the access path is to be rebuilt:
- Create Physical File (CRTPF)
- Create Logical File (CRTLF)
- Create Source Physical File (CRTSRCPF)

Note: Access paths are rebuilt during the initial program load (IPL), after the IPL, or when a file is opened.

Table 2 shows your choices for possible combinations of duplicate key and maintenance options.

Table 2. Recovery options

With this duplicate key option	And this maintenance option	Your recovery options are
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)

RECOVER parameter tip

A list of files with access paths that need to be recovered is shown on the Edit Rebuild of Access Paths display during the next IPL if the IPL is in manual mode. You can edit the original recovery option for the file by selecting the desired option on the display. After the IPL is complete, you can use the Edit Rebuild of Access Paths (EDTRBDAP) command to set the sequence in which access paths are rebuilt. If the IPL is unattended, the Edit Rebuild of Access Paths display is not shown and the access paths are rebuilt in the order determined by the RECOVER parameter. You only see the *AFTIPL and *NO (open) access paths.

Related concepts

"Recovering and restoring your database" on page 226
You can use several i5/OS save and restore functions to recover your database after the system loses data.

Backup and recovery

Specifying the share open data path (SHARE) parameter:

The system allows multiple users to access and change a database file at the same time. The SHARE parameter specifies whether the open data path (ODP) is shared with other programs in the same routing step.

The SHARE parameter allows sharing of opened files in the same job. For example, sharing a file in a job allows programs in the job to share the file's status, record position, and buffer. The file sharing can improve performance by reducing:
- The amount of storage the job needs.
- The time required to open and close the file.

   Related concepts

   "Sharing database files in the same job or activation group" on page 109
   By default, the database management system allows one file to be read and changed by many users at the same time. You can also share a file in the same job or activation group by specifying the SHARE parameter.

Specifying the maximum file and record wait time (WAITFILE and WAITRCD) parameters:

The WAITFILE and WAITRCD parameters specify how long a program can wait for a file and 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.

   Related concepts

   "Locking records" on page 105
   DB2 for i5/OS has built-in integrity for records.

   "Locking files" on page 106
   When a database file is allocated exclusively, any program trying to open the file has to wait until it is released. However, you can set a wait time for the file to become available using the WAITFILE parameter.

Specifying the authority (AUT) parameter:

Public authority is the authority given to users who do not have any specific 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. The AUT parameter specifies public authority to a database file.

   Related concepts

   "Specifying public authority" on page 95
   Public authority is given to users who do not have any specific authority to an object, who are not on the authorization list of the object, or whose group profile has no specific authority to the object. When you create a file, you can specify and grant public authority.

Specifying the system (SYSTEM) parameter:

The SYSTEM parameter specifies whether a database file is created on the local system or on the remote system that supports distributed data management (DDM).

   Related concepts

Distributed database programming

Specifying the text description (TEXT) parameter:

The TEXT parameter specifies the text that briefly describes a database file and member.

Specifying the coded character set identifier (CCSID) parameter:

A *coded character set identifier* (CCSID) is a 16-bit number that includes a specific set of encoding scheme identifiers, character set identifiers, code page identifiers, and other information that uniquely identifies the coded graphic-character representation. The CCSID parameter specifies the CCSID that describes character data in the fields of a file.

> **Related concepts**
> i5/OS globalization

Specifying the sort sequence (SRTSEQ) parameter:

The SRTSEQ parameter specifies the sort sequence for a database file.

The values of the SRTSEQ parameter, along with the values of the CCSID and LANGID parameters, determine which sort sequence table the file uses.

You can specify:
- System-supplied sort sequence tables with unique or shared collating weights. There are sort sequence tables for each supported language.
- Any user-created sort sequence table.
- The hexadecimal value of the characters in the character set.
- The sort sequence of the current job or the one specified in the ALTSEQ parameter. The sort sequence table is stored with the file, except when the sort sequence is *HEX.

Specifying the language identifier (LANGID) parameter:

The LANGID parameter specifies the language identifier that the system uses when the sort sequence (SRTSEQ) parameter value is *LANGIDSHR or *LANGIDUNQ.

The values of the LANGID, CCSID, and SRTSEQ parameters determine which sort sequence table the file uses. You can set the LANGID parameter for physical and logical files.

You can specify any language identifier supported on your system, or you can specify that the language identifier for the current job be used.

Setting up physical files

A *physical file* is 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. You can create and describe a physical file using the traditional system interface.

> **Related concepts**
> "Describing access paths for database files" on page 83
> An access path describes how records in a database file are retrieved. You can define the access path for a database file in various ways.
> **Related reference**
> "Example: Describing a physical file using DDS" on page 18
> This example shows how to describe a physical file using DDS.

Creating a physical file:

You can create a physical file using data description specifications (DDS).

You need to create a library and a source file before you create a physical file.

To create a physical file, follow these steps:

1. If you are using DDS, enter DDS for the physical file into a source file. You can do this using the source entry utility (SEU). SEU is part of the IBM WebSphere Development Studio for System i licensed program.

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.

Tables are similar to physical files. You can create tables using System i Navigator or using the CREATE TABLE SQL statement.

Related concepts

"Creating a library" on page 13
A *library* is a system object that serves as a directory to other objects. A library groups related objects and allows you to find objects by name. To create a library, use System i Navigator or the Create Library (CRTLIB) command.

"Creating a source file" on page 14
Before creating a source file, you should first create a library. Then use the Create Source Physical File (CRTSRCPF), Create Physical File (CRTPF), or Create Logical File (CRTLF) command to create a source file.

"Working with source files" on page 243
You can use various methods to enter and maintain data in a source file.

"Describing database files" on page 17
You can use several methods to describe i5/OS database files. This topic discusses how to describe a database file with data description specifications (DDS) because DDS has the most options for defining data.

Getting started with System i Navigator

Related reference

Create Physical File (CRTPF) command

Create Source Physical File (CRTSRCPF) command

CREATE TABLE

Specifying physical file and member attributes:

You can specify the attributes of physical files and members 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.

Expiration date:

The EXPDATE parameter specifies an expiration date for each member in a physical 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.

Related concepts

"Checking the expiration date of a physical file member" on page 104
The system can check whether the data in a physical file member is still current. You can specify whether the system checks the expiration date of a physical file member using the EXPCHK parameter on the Override with Database File (OVRDBF) command.

Related reference

Add Physical File Member (ADDPFM) command

Change Physical File (CHGPF) command

Change Physical File Member (CHGPFM) command

Change Source Physical File (CHGSRCPF) command

Create Physical File (CRTPF) command

Create Source Physical File (CRTSRCPF) command

Size of a physical file member:

The SIZE parameter specifies the maximum number of records that can be placed in each physical file 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	1000
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. When the file is extended beyond its maximum size, the minimum extension is 10% of the current size, even if this is larger than the specified increment. Instead of taking the default size or specifying a size, you can specify *NOMAX.

Related reference

"Database file sizes" on page 7
Before you design and create a database file, you need to know the maximum size allowed for the file.

Change Physical File (CHGPF) command

Change Source Physical File (CHGSRCPF) command

Create Physical File (CRTPF) command

Create Source Physical File (CRTSRCPF) command

Storage allocation:

The ALLOCATE parameter controls the storage that is allocated to each member when it is added to a physical file (CRTPF, CHGPF, CRTSRCPF, and CHGSRCPF commands).

The storage allocated is 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.

Related reference

Change Physical File (CHGPF) command

Change Source Physical File (CHGSRCPF) command

Create Physical File (CRTPF) command

Create Source Physical File (CRTSRCPF) command

Method of allocating storage:

The CONTIG parameter controls the method of allocating physical storage for each physical file 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 after creation, regardless of what you specified on the CONTIG parameter.

Related reference

Create Physical File (CRTPF) command

Create Source Physical File (CRTSRCPF) command

Record length:

The RCDLEN parameter specifies the length of records in a physical 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 (the system automatically determines the length of records in the file from the field-level descriptions).

Related reference

Create Physical File (CRTPF) command

Create Source Physical File (CRTSRCPF) command

Deleted records:

The DLTPCT parameter specifies the percentage of deleted records that each physical file member 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 the value specified in the DLTPCT parameter, a message is sent to the history log. (For information about processing the history log, see the Control language topic collection. 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 can run the Reorganize Physical File Member (RGZPFM) command to reclaim the space. 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.

REUSEDLT parameter specifies whether deleted record space should be reused on subsequent write operations (CRTPF and CHGPF commands). When you specify *YES for the REUSEDLT parameter, all insert requests on that file try to reuse deleted record space. Reusing deleted record space allows you to reclaim space used by deleted records without having to issue a RGZPFM command. When the CHGPF command is used to change a file to reuse deleted records, it might take a long time to run, especially if the file is large and there are already a lot of deleted records in it. It is important to note the following items:

- The term *arrival order* loses its meaning for a file that reuses deleted record space. Records are no longer always inserted at the end of the file when deleted record space is reused.
- If a new physical file is created with the reuse deleted record space attribute and the file is keyed, the FIFO or LIFO access path attribute cannot be specified for the physical file, nor can any keyed logical file with the FIFO or LIFO access path attribute be built over the physical file.
- You cannot change an existing physical file to reuse deleted record space if there are any logical files over the physical file that specify FIFO or LIFO ordering for duplicate keys, or if the physical file has a FIFO or LIFO duplicate key ordering.
- Reusing deleted record space should not be specified for a file that is processed as a direct file or if the file is processed using relative record numbers.

  Related concepts

  "Reorganizing a physical file member" on page 210
  You can reorganize a physical file member to change the manner in which records are stored on the i5/OS operating system.

  "Reusing deleted records" on page 101
  Sometimes you might want to reuse deleted records for your database files. In this case, you can use the REUSEDLT parameter.

  Related reference

  Change Physical File (CHGPF) command

  Change Source Physical File (CHGSRCPF) command

  Create Physical File (CRTPF) command

  Create Source Physical File (CRTSRCPF) command

Physical file capabilities:

File capabilities are used to control which input/output operations are allowed for a database file independent of database file authority. The ALWUPD and ALWDLT parameters specify whether records in a physical file can be updated and deleted (CRTPF and CRTSRCPF commands).

  Related concepts

  "Securing database files" on page 92
  You can secure database files in various ways.

  Related reference

Create Physical File (CRTPF) command

Create Source Physical File (CRTSRCPF) command

Source type:

The SRCTYPE parameter specifies the source type for a member in a source file (ADDPFM and CHGPFM commands).

The source type determines the syntax checker, prompting, and formatting that are used for the member. If you specify a unique source type other than the types that are supported on the i5/OS operating system, such as COBOL and RPG, you must provide the programming to handle the unique type.

If the source type is changed, it is only reflected when the member is subsequently opened; members currently open are not affected.

> **Related reference**
>
> Add Physical File Member (ADDPFM) command
>
> Change Physical File Member (CHGPFM) command

Implicit physical file journaling:

A physical file can be journaled automatically when it is created.

If a data area called QDFTJRN exists in the same library into which the physical file is created, and if the user is authorized to the QDFTJRN data area, journaling is started to the journal named in QDFTJRN if all of the following conditions are met:

- The identified library for the physical file must not be QSYS, QSYS2, QRECOVERY, QSPL, QRCL, QRPLOBJ, QGPL, QTEMP, or any of the independent auxiliary storage pool (ASP) equivalents of these libraries. For example, an independent ASP equivalent of the QRPLOBJ library is QRPL*xxxxx*, where *xxxxx* is the number of a primary ASP.
- The journal specified in the QDFTJRN data area must exist, but authority to the specified journal is not required.
- The first 10 bytes of the QDFTJRN data area must contain the name of the library in which to find the journal.
- The second 10 bytes must contain the name of the journal.
- The third *n* bytes must contain the value *FILE. The value *NONE can be used to prevent journaling from being started.

If the QDFTJRN data area is used, the creation of the physical file cannot be replayed by the Apply Journaled Changes (APYJRNCHG) command, although it can be replayed by the Apply Journaled Changes Extend (APYJRNCHGX) command.

> **Related concepts**
>
> Automatically starting journaling
>
> **Related tasks**
>
> "Starting or ending journaling for libraries" on page 232
> You can journal a library or a list of libraries and, optionally, journal database files that are created, moved, or restored into the library or the list of libraries.
>
> **Related reference**
>
> Apply Journaled Changes (APYJRNCHG) command
>
> Apply Journaled Changes Extend (APYJRNCHGX) command

Setting up logical files

A *logical file* contains no data, but it defines record formats for one or more physical files. You can create various logical files and describe their record formats and access paths using data description specifications (DDS).

Many of the rules for setting up logical files apply to all categories of logical files. In these topics, 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.

Creating a logical file

You can create a logical file using data description specifications (DDS).

The physical file or files on which the logical file is based must already exist before you create a logical file.

To create a logical file, follow these steps:

1. Type the DDS for the logical file into a source file. You can do this using the source entry utility (SEU). The following example shows the DDS for logical file ORDHDRL (an order header file):

   ```
   |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A* ORDER HEADER LOGICAL FILE (ORDHDRL)
      A          R ORDHDR                    PFILE(ORDHDRP)
      A          K ORDER
   ```

   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 for 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 example shows how the CRTLF command can 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 uses 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 ORDHDRL file with one member of the same name is placed in the DSTPRODLB library.

You can create multiple logical files over a single physical file. The maximum number of logical files that can be created over a single physical file is 32KB.

Views are similar to logical files. You can create views using System i Navigator or using the CREATE VIEW SQL statement.

> **Related concepts**
>
> "Working with source files" on page 243
> You can use various methods to enter and maintain data in a source file.
>
> "Identifying which record format to add in a file with multiple formats" on page 200
> 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.
>
> Creating and using a view
>
> **Related reference**
>
> Create Logical File (CRTLF) command
> CREATE VIEW

Creating a logical file with more than one record format:

A multiple-format logical file allows you to use records from two or more physical files by referring to only one logical file.

Each record format of such a multiple-format logical file is always associated with one or more physical files. You can use the same physical file in more than one record format.

The following example shows the data description specifications (DDS) for a physical file, ORDDTLP, built from a field reference file:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     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
```

The following example shows the DDS for the ORDHDRP physical file:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     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            ORDMNT    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
     A
```

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 ORDHDRP physical file; the other is defined for order detail records from the ORDDTLP physical file.

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.

The following example shows the DDS for the ORDFILL logical file:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     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
```

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

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

The DDS source is in the ORDFILL member of the QDDSSRC file. The ORDFILL file with a member of
the same name is placed in the DSTPRODLB library. The access path for the ORDFILL logical file
member 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 ORDHDRP header file and second from the ORDDTLP detail file. Because FIFO, LIFO, or
FCFO are not specified, the order of retrieval of duplicate keys in the same file is not guaranteed.

Note: In some circumstances, it is better to use multiple logical files, rather than to use a multiple-format
logical file. For example, when keyed access is used with a multiple-format logical file, it is
possible to experience poor performance if one of the files has very few records. Even though there
are multiple formats, the logical file has only one index, with entries from each physical file.
Depending on the kind of processing being done by the application program (for example, using
RPG SETLL and READE with a key to process the small file), the system might have to search all
index entries in order to find an entry from the small file. If the index has many entries, searching
the index might take a long time, depending on the number of keys from each file and the
sequence of keys in the index. (If the small file has no records, performance is not affected, because
the system can take a fast path and avoid searching the index.)

Controlling how records are retrieved in a logical file with multiple formats:

Key field definitions are required in a logical file with more than one record format.

Each record format can have its own key field definition. The record format key fields can be defined to
merge the records from different formats. Each record format does not have to contain every key field in
the key.

Consider the following records.

Table 3. Header record format

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

Table 4. 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 data description specifications (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. When such records are retrieved by key from more than one record format, only key fields that appear in all record formats before the *NONE are used. To increase the number of key fields that are used, limit the number of record formats considered.

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) **1**
EMPHIST	EMPHIST	Empnbr, Empdat (employed date) **2**
EMPEDUC	EMPEDUC	Empnbr, Clsnbr (class number) **3**

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

The DDS for this example is:
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A
     A          K EMPNBR   1
     A
     A          K EMPNBR   2
     A          K EMPDAT
     A
     A          K EMPNBR   3
     A          K *NONE
     A          K CLSNBR
     A
```

*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 table 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 predictable.

> **Related concepts**
> DDS concepts

Controlling how records are added to a logical file with multiple formats:

To add a record to a multiple-format logical file, you need to identify the member of the based-on physical file to which you want the record to be written.

If the application you are using does not allow you to specify a particular member within a format, each of the formats in the logical file needs to be associated with a single physical file member. If one or more of the based-on physical files contain more than one member, you need to use the DTAMBRS parameter, described in "Defining logical file members," to associate a single member with each format. Finally, give each format in the multiple format logical file a unique name. If the multiple format logical file is defined in this way, then when you specify a format name on the add operation, you target a particular physical file member into which the record is added.

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.

> **Related concepts**
> "Identifying which record format to add in a file with multiple formats" on page 200
> 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.

Defining logical file members:

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

The following figure illustrates this concept.

LF1
M1

LF1
M1 M2 M3

LF1
M1 M2

PF1

PF2

PF1

M1 M2 M3

M1 M2 M3

M1 M2 M3

LF1
M1 M2 M3

PF1

PF2

PF3

M1 M2

M1 M2 M3

M1 M2 M3

RBAFO506-0
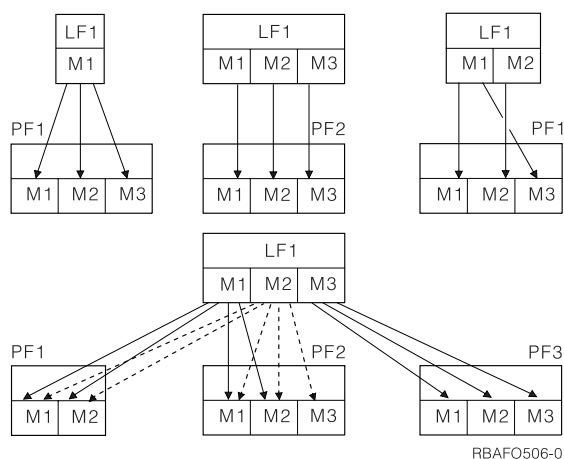
The record formats used with all logical members in a logical file must be defined in data description specifications (DDS) when the file is created. If new record formats are needed, another logical file or record format must be created.

The attributes of an access path are determined by the 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 CRTLF or the 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:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A
00010A         R LOGRCD2                  PFILE(PF1 PF2)
     A           .
     A           .
     A           .
00020A         R LOGRCD3                  PFILE(PF1 PF2 PF3)
     A           .
     A           .
     A           .
     A
```

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.

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

*FILE is the default for the MBR parameter and means that 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) member. The text description is the text description of the member.

Describing logical file record formats

For every logical file record format described with data description specifications (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:
* 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. Consider this example of a simple logical file:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  A
  A          R ORDDTL                  PFILE(ORDDTLP)
  A
```

* 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. Consider this example of a simple logical file with fields specified::

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A
      A          R ORDHDR                      PFILE(ORDHDRP)
      A            ORDER
      A            CUST
      A            SHPVIA
      A
```

- Specify the name of a database file for the file name on the FORMAT keyword. The record format is shared from this database file by the logical file 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. Consider this example:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A
      A          R CUSRCD                      PFILE(CUSMSTP)
      A                                        FORMAT(CUSMSTL)
      A
```

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

Physical File

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

RBAFO503-0

For the logical file, the DDS would be as follows:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A
      A          R LOGREC                      PFILE(PF1)
      A            D             10S 0
      A            A
      A            C              5S 0
      A
```

For the physical file, the DDS would be as follows:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A
      A          R PHYREC
      A            A              8S 2
      A            B             32
      A            C              2B 0
      A            D             10
      A
```

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	Date	Time	Timestamp
Character or Hexadecimal	Valid	See Notes 1	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid
Zoned	See Notes 1	Valid	Valid	See Notes 2	Valid	Not valid	Not valid	Not Valid
Packed	Not valid	Valid	Valid	See Notes 2	Valid	Not valid	Not valid	Not valid
Binary	Not valid	See Notes 2	See Notes 2	See Notes 3	See Notes 2	Not valid	Not valid	Not valid
Floating point	Not valid	Valid	Valid	See Notes 2	Valid	Not valid	Not valid	Not valid
Date	Not valid	Valid	Not valid	Not valid	Not valid	Valid	Not valid	Not valid
Time	Not valid	Valid	Not valid	Not valid	Not valid	Not valid	Valid	Not valid
Timestamp	Not valid	Not valid	Not valid	Not valid	Not valid	Valid	Valid	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.

Related concepts

"Double-byte character set considerations" on page 285
A double-byte character set (DBCS) is a character set that represents each character with 2 bytes. The database on the i5/OS operating system supports DBCS.

Describing field use for logical files:

You can specify that the fields in a logical file be input-only, both (input/output), or neither fields.

To describe field use for a logical file, specify one of the following items 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.

Describing field use for logical files: 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.

Describing field use for logical 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.

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).

Describing field use for logical files: Neither:

A neither field is used neither for input nor for output.

A neither field is valid only for join logical files. It 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.

> **Related reference**
>
> "Example 5: Describing fields that never appear in the record format" on page 72
> Neither fields (where N is specified in position 38) can be used in a join logical file for neither input nor output. Neither fields are not included in the record format. This example shows how to describe such fields that never appear in the record format.

Deriving new fields from existing fields:

Fields in a logical file can be derived from fields in a physical file on which the logical file is based 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:

You can use the Concatenate (CONCAT) DDS keyword to combine two or more fields from a physical file record format into 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:

- The operating system 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 a concatenated field must be I (input only) if the concatenated field is variable length. Otherwise, the use can be B (both input and output).
- REFSHIFT cannot be specified on a concatenated field that has been assigned a data type of O or J.
- If any of the fields contain the null value, the result of concatenation is the null value.

Note: For information about concatenating DBCS fields, see "Double-byte character set considerations" on page 285.

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. Date, time, timestamp, and floating-point fields cannot be included in a concatenated field.

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

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A
00101A          MONTH
00102A          DAY
00103A          YEAR
00104A          DATE                         CONCAT(MONTH DAY YEAR)
     A
```

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 formats 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 Substring (SST) DDS keyword to specify which fields (character, hexadecimal, or zoned decimal) are in a substring.

You can also use substring with a packed field in a physical file by specifying S (zoned decimal) as the data type in the logical file.

For example, assume that you defined the *Date* field in physical file *PF1* as 6 characters in length. You can describe the logical file with three fields, each 2 characters in length. You can 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.

The following example shows the field descriptions in data description specifications (DDS) for these substring fields. The SST keyword is used to specify the field to substring.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  A          R REC1                    PFILE(PF1)
  A
  A            MM              I        SST(DATE 1 2)
  A            DD              I        SST(DATE 3 2)
  A            YY              I        SST(DATE 5 2)
  A
```

Note: The starting position of the substring is specified according to its position in the field being operated on (*Date*), not according to its position in the file. The I in the Usage column indicates input-only.

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

Renamed fields:

You can use the Rename (RENAME) DDS keyword to name a field in a logical file differently than in a physical file.

You might want to rename a field in a logical file because the program uses a different field name or because the original field name does not conform to the naming restrictions of your high-level language.

Translated fields:

You can use the Translation Table (TRNTBL) DDS keyword to specify a translation table for a field in a logical file.

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 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 double-byte character set (DBCS) field.

Mapping between floating-point fields of different precision or between floating-point fields and other numeric fields can result in rounding or a loss of precision. For example, 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. Binary floating-point 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 that your program did not explicitly change. For binary 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 binary floating-point number causes it to be rounded when it is mapped to the logical file, then updating 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.

DB2 for i5/OS supports both binary floating-point and decimal floating-point fields. Although DDS cannot be used to specify a decimal floating-point field type, you can create a logical file with a decimal floating-point field by sharing the format of an SQL table. Decimal floating-point fields can be mapped to or from the same field types as binary floating point. Decimal floating-point rounding for traditional system interfaces is round to nearest, ties to even. For more information about decimal floating point support, see the DB2 for i5/OS SQL reference topic collection.

Describing access paths for logical files

You can specify the access path for a logical file in several ways.

- 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.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
A          R CUSRCD                    PFILE(CUSMSTP)
A          K ARBAL
A          K CRDLMT
A
```

- Encoded vector access path specification. You define the encoded vector access path with the SQL CREATE INDEX statement.

- 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).

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
A          R CUSRCD                    PFILE(CUSMSTP)
```

- 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.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
                                        REFACCPTH(DSTPRODLIB/ORDHDRL)
     A          R CUSRCD                PFILE(CUSMSTP)
```

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.

> **Related reference**
> CREATE INDEX

Selecting and omitting records for logical files:

You can select and omit records for a logical file. This helps exclude records from 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 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 selected records that match the select/omit criteria can be retrieved through the select/omit access path.

In DDS, to specify select or omit, you specify an S (select) or O (omit) in position 17 of the DDS 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.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          S ITMNBR                VALUES(301542 306902 382101 422109 +
     A                                  431652 486592 502356 556608 590307)
     A
```

- 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.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          S ITMNBR                RANGE(301000 599999)
     A
```

- 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:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          S ITMNBR                  CMP(LE 599999)
    A
```

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 of 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 leave a space in position 17 of the DDS 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.

The following examples show ways to code select/omit functions. In these 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. **3** shows the most efficient way.
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          S ST                      CMP(EQ 'NY')       1
    A            REP                      CMP(EQ 'JSMITH')
    A            YEAR                     CMP(LT 88)
    A
```
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          O YEAR                    CMP(GE 88)         2
    A          S ST                      CMP(EQ 'NY')
    A            REP                      CMP(EQ 'JSMITH')
    A
```
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          O REP                     CMP(NE 'JSMITH')   3
    A          O ST                      CMP(NE 'NY')
    A          S YEAR                    CMP(LT 88)
    A
```

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 must 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 items:
- 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.

If you create the preceding example with a sort sequence table, the select/omit fields are translated according to the sort table before the comparison. For example, with a sort sequence table using shared weightings for uppercase and lowercase, NY and ny are equal.

The following diagram shows the logic of this example.

RBAFO504-0
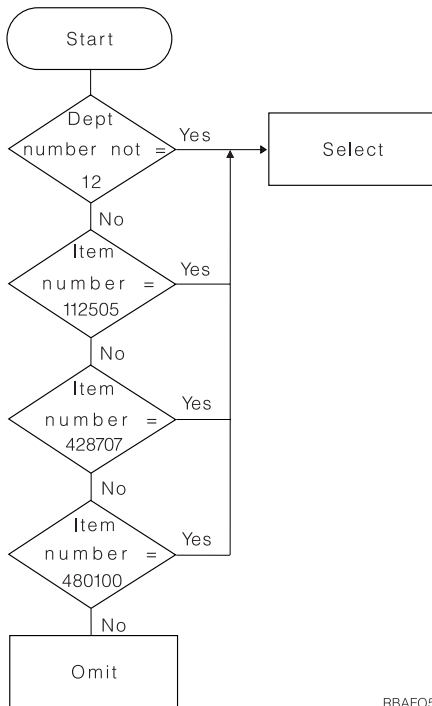
This example uses the DDS select and omit functions:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  A           S DPTNBR                    CMP(NE 12)
  A           S ITMNBR                    VALUES(112505 428707 480100)
  A
```

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.

You can also use the Open Query File (OPNQRYF) command to select or omit records.

Related concepts

DDS concepts

Access path select/omit:

With the access path select/omit operation, the access path only contains keys that meet the select/omit values specified for the logical file.

When you specify key fields for a logical file, an access path is kept for the file and maintained by the system when you add or update records in the physical files 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 the dynamic select/omit operation, the system only returns those records that meet the select/omit values when a program reads records from the file. 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 sequence 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.

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 might 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.

Related concepts

"Sharing existing access paths between logical files"
When two or more logical files are based on the same physical files with the same key fields in the same order, they automatically share the same keyed sequence access path.

Selecting and omitting logical file records using the Open Query File (OPNQRYF) command:

As an alternative to using DDS, you can select and omit records for a logical file by specifying 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, however, remains in existence until you specifically delete it.

Related concepts

"Using Open Query File (OPNQRYF) command" on page 123
By using the Open Query File (OPNQRYF) command, you can open a file to a set of database records that satisfies a database query request.

Sharing existing access paths between logical files:

When two or more logical files are based on the same physical files with 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 the access paths and the amount of auxiliary storage used by the files are reduced. So 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 that satisfies the following conditions must exist on the system:

- 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. However, when a file shares a partial set of keys from an existing access path, any record updates made to fields that are part of the set of keys for the shared access path might change the record position in that access path.
- 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, and RECOVER 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, and RECOVER 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 list illustrates how this occurs:

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

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.

> **Related concepts**
>
> "Arranging duplicate keys" on page 89
> If you do not specify the Unique (UNIQUE) keyword in data description specifications (DDS), you can specify how the system stores records with duplicate key values.

Example: Implicitly shared access paths:

This example shows how to implicitly share an access path between logical files.

Two logical files, LFILE1 and LFILE2, are built over the physical file PFILE. LFILE1, which was created first, has two key fields, KFD1 and KFD2. LFILE2 has three key fields, KFD1, KFD2, and KFD3. The two logical files use two of the same key fields, but no access path is shared because the logical file with three key fields was created after the file with two key fields.

Table 5. Physical and logical files before save and restore

Description	Physical file (PFILE)	Logical file 1 (LFILE1)	Logical file 2 (LFILE2)
Access path		KFD1, KFD2	KFD1, KFD2, KFD3
Fields	KFD1, KFD2, KFD3, A, B, C, D, E, F, G	KFD1, KFD2, KFD3, F, C, A	KFD1, KFD2, KFD3, D, G, F, E

An application uses LFILE1 to access the records and to change the KFD3 field to blank if it contains a C, and to a C if it is blank. This application does not cause any unexpected results because the access paths are not shared. However, after the physical file and both logical files are saved and restored, the program appears to do nothing and takes longer to process.

Unless you change the way you restore the logical files (for example, if you restore LFILE1 and LFILE2 separately with the Restore Object (RSTOBJ) command), the system restores the logical file with the largest number of keys first or does not build unnecessary access paths.

Because it has three key fields, LFILE2 is restored first. After recovery, LFILE1 implicitly shares the access path for LFILE2. Users who do not understand implicitly shared access paths do not realize that when they use LFILE1 after a recovery, they are really using the key for LFILE2.

Table 6. Physical and logical files after save and restore. Note that the only difference from before the save and restore is that the logical files now share the same access path.

Description	Physical file (PFILE)	Logical file 1 (LFILE1)	Logical file 2 (LFILE2)
Access path		KFD1, KFD2, KFD3	KFD1, KFD2, KFD3
Fields	KFD1, KFD2, KFD3, A, B, C, D, E, F, G	KFD1, KFD2, KFD3, F, C, A	KFD1, KFD2, KFD3, D, G, F, E

The following records are tested and changed.

Relative record	KFD1	KFD2	KFD3
001	01	01	<blank>
002	01	01	<blank>
003	01	01	<blank>
004	01	01	<blank>

The first record is read through the first key of 0101<blank> and changed to 0101C. The records now look like this.

Relative record	KFD1	KFD2	KFD3
001	01	01	C
002	01	01	<blank>
003	01	01	<blank>
004	01	01	<blank>

When the application issues a get next key, the next higher key above 0101<blank> is 0101C. This is the record that was just changed. However, this time the application changes the KFD3 field from C to blank.

Because the user does not understand implicit access path sharing, the application accesses and changes every record twice. The end result is that the application takes longer to run, and the records look like they have not changed.

Setting up a join logical file

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

To create a join logical file, follow this process:

1. Find the 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. Optional: Specify the following items:
   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.

The examples that follow show how to create various join logical files.

In general, these examples include a picture of the files, data description specifications (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 these 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, and PF3.

> **Related concepts**
>
> Joining data from more than one table

Example 1: Joining two physical files:

This example shows how to create a join logical file that joins two physical files.
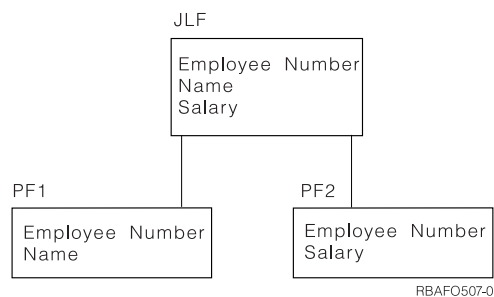
In this example, the join logical file (JLF) has field *Employee number*, *Name*, and *Salary*. Physical file 1 (PF1) has *Employee number* and *Name*, while physical file 2 (PF2) has *Employee number* and *Salary*. *Employee number* is common to both physical files (PF1 and PF2), but *Name* is found 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 data file utility (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.

This example uses the following DDS:

```
JLF
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     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
```

PF1
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R REC1
     A            NBR             10
     A            NAME            20
     A          K NBR
     A
```

PF2
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R REC2
     A            NBR             10
     A            SALARY           7  2
     A          K NBR
     A
```

The following list describes the DDS for the join logical file in example 1:

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.

JFLD   Identifies the join fields that join records from the physical files specified on the JOIN keyword. 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.

   Join fields, except character type fields, must have the same attributes (data type, length, and decimal positions). If the fields are character type fields, they do not need to have the same length. 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.

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.

Related concepts

DDS concepts

Related reference

"Example 7: Joining three or more physical files" on page 74
This example shows how to use a join logical file to join three or more physical files.

"Example 4: Using join fields whose attributes are different" on page 70
This example shows how to handle join fields when their attributes (length, data type, and decimal positions) are different.

Reading a join logical file:

These cases show how the join logical file in example 1 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 things:

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 might 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 "Case 1: Matching records in primary and secondary files" on page 62.

   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 "Case 2A: A record missing in the secondary file (JDFTVAL keyword not specified)" on page 63 and "Case 2B: A record missing in the secondary file (JDFTVAL keyword specified)" on page 63.

  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.

Case 1: Matching records in primary and secondary files:

This example join logical file contains a single record for each record in the primary file because each record in the primary file has a match in the secondary file.

Assume that a join logical file is specified as in "Example 1: Joining two physical files" on page 59, and four records are contained in both PF1 and PF2, as shown in the following two tables.

Table 7. Physical file 1 (PF1)

Employee number	Name
235	Anne
440	Doug
500	Mark
729	Sue

Table 8. Physical file 2 (PF2)

Employee number	Salary
235	1700.00
440	950.50
500	2100.00
729	1400.90

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

Table 9. Join logical file (JLF)

Employee number	Name	Salary
235	Anne	1700.00

Table 9. Join logical file (JLF) (continued)

Employee number	Name	Salary
440	Doug	950.50
500	Mark	2100.00
729	Sue	1400.90

Case 2A: A record missing in the secondary file (JDFTVAL keyword not specified):

This example join logical file misses one record in the primary file because it does not have a match in the secondary file and the Join Default Values (JDFTVAL) DDS keyword is not specified.

Assume that a join logical file is specified as in "Example 1: Joining two physical files" on page 59, and there are four records in PF1 and three records in PF2, as shown in the following two tables.

Table 10. Physical file 1 (PF1)

Employee number	Name
235	Anne
440	Doug
500	Mark
729	Sue

Table 11. Physical file 2 (PF2)

Employee number	Salary
235	1700.00
440	950.50
729	1400.90

In PF2, no record is found for number 500.

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

Table 12. Join logical file (JLF)

Employee number	Name	Salary
235	Anne	1700.00
440	Doug	950.50
729	Sue	1400.90

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

Case 2B: A record missing in the secondary file (JDFTVAL keyword specified):

Because the Join Default Values (JDFTVAL) DDS keyword is specified, this example join logical file contains a single record for each record in the primary file even though one record does not have a match in the secondary file.

Assume that a join logical file is specified as in "Example 1: Joining two physical files" on page 59, except that the JDFTVAL keyword is specified, as shown in the following DDS:

```
JLF
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    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
```

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

Table 13. Join logical file (JLF)

Employee number	Name	Salary
235	Anne	1700.00
440	Doug	950.50
500	Mark	0000.00
729	Sue	1400.90

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 zeros. Therefore, in this case, the value for the missing record in the join record is 0. However, if the DFT keyword is specified for the field in the physical file, the default value specified on the DFT keyword is used.

Case 3: Secondary file has more than one match for a record in the primary file:

This example join logical file contains a duplicate record in the primary file because it has more than one match in the secondary file.

Assume that a join logical file is specified as in "Example 1: Joining two physical files" on page 59, and there are four records in PF1 and five records in PF2, as shown in the following two tables.

Table 14. Physical file 1 (PF1)

Employee number	Name
235	Anne
440	Doug
500	Mark
729	Sue

Table 15. Physical file 2 (PF2)

Employee number	Salary
235	1700.00
235	1500.00
440	950.50
500	2100.00
729	1400.90

In PF2, the record for 235 is duplicated.

The program gets five records.

Table 16. Join logical file (JLF)

Employee number	Name	Salary
235	Anne	1700.00
235	Anne	1500.00
440	Doug	950.50
500	Mark	0000.00
729	Sue	1400.90

In the join records, the record for 235 is duplicated. The order of the records received for the duplicated record is unpredictable unless the JDUPSEQ keyword is used.

Related reference

"Example 3: Reading duplicate records in the secondary file" on page 69
This example shows how a join logical file reads duplicate records in the secondary file based on the specification of the Join Duplicate Sequence (JDUPSEQ) DDS keyword.

Case 4: An extra record in the secondary file:

This example join logical file does not include an extra record in the secondary file because the record does not have a match in the primary file.

Assume that a join logical file is specified as in "Example 1: Joining two physical files" on page 59, and four records are contained in PF1 and five records in PF2.

The record for 301 exists only in PF2.

The program reads the join logical file and gets only four records. The record for 301 does not appear.

Table 17. Join logical file (JLF)

Employee number	Name	Salary
235	Anne	1700.00
440	Doug	950.50
500	Mark	2100.00
729	Sue	1400.90

These results would be the same even if the JDFTVAL keyword were specified, because a record must always be contained in the primary file to receive a join record.

Case 5: Random access:

This example join logical file returns records for a random access read operation.

Assume that a join logical file is specified as in "Example 1: Joining two physical files" on page 59. Note that the join logical file has key fields defined.

Assume that PF1 and PF2 have the following records.

Table 18. Physical file 1 (PF1)

Employee number	Name
235	Anne
440	Doug
500	Mark
729	Sue
997	Tim

Table 19. Physical file 2 (PF2)

Employee number	Salary
235	1700.00
440	950.50
729	1400.90
984	878.25
997	331.00
997	555.00

In PF2, no record is found for record 500, record 984 exists only in PF2, and duplicate records are found for 997.

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.

Employee number	Name	Salary
235	Anne	1700.00

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.

Employee number	Name	Salary
500	Mark	0000.00

Note: If the JDFTVAL keyword was not specified in the join logical file, no record is 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.

Employee number	Name	Salary
997	Tim	331.00

or

Employee number	Name	Salary
997	Tim	555.00

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.

Notes:

1. With random access, the application programmer must be aware that duplicate records can be contained in PF2, and ensure that the program does more than one read operation for records with duplicate keys. If the program is using sequential access, a second read operation gets 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 that 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 can share the access path of PF2.)

Related reference

"Example 3: Reading duplicate records in the secondary file" on page 69
This example shows how a join logical file reads duplicate records in the secondary file based on the specification of the Join Duplicate Sequence (JDUPSEQ) DDS keyword.

Example 2: Using more than one field to join files:

You can specify more than one join field to join a pair of files. This example shows the fields in the logical file and the two physical files.
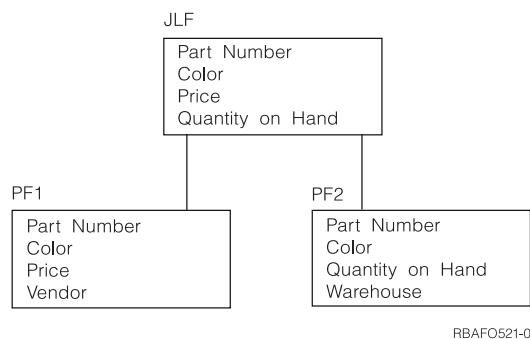
The join logical file (JLF) has fields *Part number*, *Color*, *Price*, and *Quantity on hand*. Physical file 1 (PF1) has *Part number*, *Color*, *Price*, and *Vendor*, while physical file 2 (PF2) has *Part number*, *Color*, *Quantity on hand*, and *Warehouse*. The data description specifications (DDS) for these files are shown as follows:

JLF
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    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
```

PF1

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A           R REC1
      A             PTNBR         4
      A             COLOR        20
      A             PRICE         7  2
      A             VENDOR       40
      A
```

PF2
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A           R REC2
      A             PTNBR         4
      A             COLOR        20
      A             QUANTOH       5  0
      A             WAREHSE      30
      A
```

Assume that the physical files have the following records.

Table 20. Physical file 1 (PF1)

Part number	Color	Price	Vendor
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.

Table 21. Physical file 2 (PF2)

Part number	Color	Quantity on hand	Warehouse
100	Black	23	ABC Corp.
100	White	15	Ajax Inc.
120	Yellow	102	ABC Corp.
187	Green	**0**	ABC Corp.
187	Red	**2**	ABC Corp.
190	Blue	**2**	Ajax Inc.

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

Table 22. Join logical file (JLF)

Part number	Color	Price	Quantity on hand
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**
190	Blue	40.00	**2**

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

Example 3: Reading duplicate records in the secondary file:

This example shows how a join logical file reads duplicate records in the secondary file based on the specification of the Join Duplicate Sequence (JDUPSEQ) DDS keyword.

The data description specifications (DDS) for the physical files and for the join logical file are shown as follows:

```
JLF
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          R JREC                     JFILE(PF1 PF2)
    A          J                          JOIN(PF1 PF2)
    A                                      JFLD(NAME1 NAME2)
    A                                      JDUPSEQ(TELEPHONE)
    A            NAME1
    A            ADDR
    A            TELEPHONE
    A

PF1
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          R REC1
    A            NAME1        10
    A            ADDR         20
    A

PF2
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          R REC2
    A            NAME2        10
    A            TELEPHONE     8
    A
```

The physical files have the following records.

Table 23. Physical file 1 (PF1)

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

Table 24. Physical file 2 (PF2)

Name	Telephone
Anne	555–1111
Anne	555–6666
Anne	555–2222
Doug	555–5555

The join logical file returns the following records.

Table 25. Join logical file (JLF)

Name	Address	Telephone
Anne	120 1st St.	555–1111
Anne	120 1st St.	555–2222
Anne	120 1st St.	555–6666
Doug	40 Pillsbury	555–5555

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
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    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
```

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

Table 26. Join logical file (JLF)

Name	Address	Telephone
Anne	120 1st St.	555–6666
Anne	120 1st St.	555–2222
Anne	120 1st St.	555–1111
Doug	40 Pillsbury	555–5555

Note: The JDUPSEQ keyword applies only to the join specification in which it is specified.

> **Related reference**
> "Example 10: A complex join logical file" on page 79
> This example shows a more complex join logical file.

Example 4: Using join fields whose attributes are different:

This example shows how to handle join fields when their attributes (length, data type, and decimal positions) are different.

For example, as in "Example 3: Reading duplicate records in the secondary file" on page 69, 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.

You can also use character type fields that have different lengths as join fields without requiring any redefinition of the fields. For example, if the NAME1 field of PF1 is 10 characters long and the NAME2 field of PF2 is 15 characters long, those fields can be used as join fields without redefining one of the fields.

The following example shows the join fields that do not have the same attributes. Both physical files have fields for employee number. 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 example illustrates the fields in the logical and physical files:
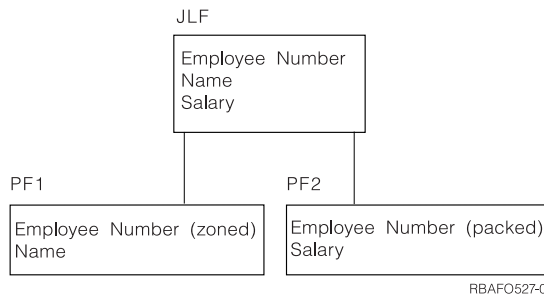
The join logical file (JLF) contains *Employee number*, *Name*, and *Salary* fields. Physical file 1 (PF1) contains *Employee number* (zoned) and *Name*. Physical file 2 (PF2) contains *Employee number* (packed) and *Salary*. The data description specifications (DDS) for these files are shown as follows:

```
JLF
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     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
```

```
PF1
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R REC1
     A            NBR              3S 0 <-Zoned
     A            NAME             20
     A          K NBR
     A
```

```
PF2
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R REC2
     A            NBR              3P 0 <-Packed
     A            SALARY            7  2
     A          K NBR
     A
```

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).

Example 5: Describing fields that never appear in the record format:

Neither fields (where N is specified in position 38) can be used in a join logical file for neither input nor output. Neither fields are not included in the record format. This example shows how to describe such fields that never appear in the record format.

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 operation, 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.

JLF

> Description
> Price
> Quantity on Hand

PF1

> Description
> Part Number

PF2

> Part Number
> Price
> Quantity on Hand

RBAFO528-0
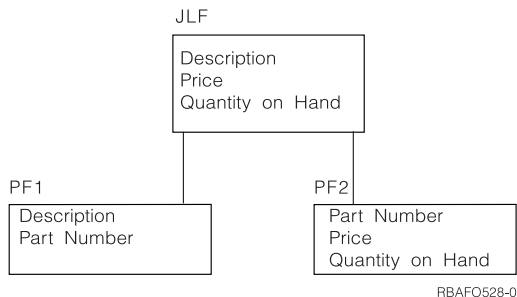
The join logical file (JLF) has fields *Description*, *Price*, and *Quantity on hand*. Physical file 1 (PF1) has *Description* and *Part number*, while physical file 2 (PF2) has *Part number*, *Price*, and *Quantity on hand*. The data description specifications (DDS) for these files are shown as follows:

JLF
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     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
     A
```

PF1
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R REC1
     A            DESC          30
     A            PRTNBR         6P 0
     A
```

PF2
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R REC2
     A            PRTNBR         6S 0
     A            PRICE          7  2
     A            QUANT          8  0
     A
```

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.

Example 6: Specifying key fields in a join logical file:

This example illustrates the rules for specifying key fields in a join logical file.

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 example illustrates the rules for key fields:

```
JLF
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     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
```

```
PF1
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R REC1
     A            NBR              4
     A            F1              20
     A            FLD2             7   2
     A            FLD3            40
     A            NAME            20
     A
```

```
PF2
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R REC2
     A            NUMBER           4
     A            FLD2             7   2
     A            FLD31           35
     A            AREA             3
     A            LOCAL            7
     A
```

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)

Example 7: Joining three or more physical files:

This example shows how to use a join logical file to join three or more physical files.

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 chart shows the fields in the files and one field common to all the physical files in the logical file.

RBAFO529-0
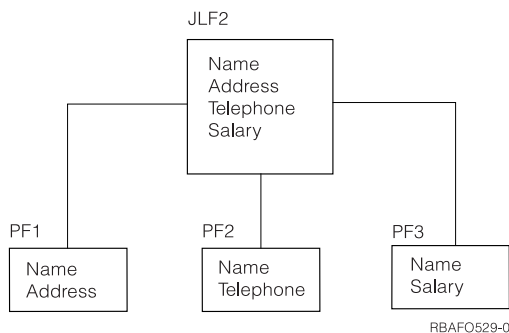
The join logical file (JLF2) contains *Name*, *Address*, *Telephone*, and *Salary*. Physical file 1 (PF1) has *Name* and *Address*, physical file 2 (PF2) has *Name* and *Telephone*, and physical file 3 (PF3) has *Name* and *Salary*. In this example, the *Name* field is common to all the physical files (PF1, PF2, and PF3), and serves as the join field.

The following example shows the data description specifications (DDS) for the physical and logical files:

JLF
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A           R JOINREC               JFILE(PF1 PF2 P3)
    A           J                       JOIN(PF1 PF2)
    A                                   JFLD(NAME NAME)
    A           J                       JOIN(PF2 PF3)
    A                                   JFLD(NAME NAME)
    A             NAME                  JREF(PF1)
    A             ADDR
    A             TELEPHONE
    A             SALARY
    A           K NAME
    A
```

PF1
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A           R REC1
    A             NAME          10
```

```
      A            ADDR          20
      A          K NAME
      A
```

PF2
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A          R REC2
      A            NAME          10
      A            TELEPHONE      7
      A          K NAME
      A
```

PF3
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A          R REC3
      A            NAME          10
      A            SALARY         9  2
      A          K NAME
      A
```

Assume that the physical files have the following records.

Table 27. Physical file 1 (PF1)

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

Table 28. Physical file 2 (PF2)

Name	Telephone
Anne	555–1111
Doug	555–5555
Mark	555–0000
Sue	555–3210

Table 29. Physical file 3 (PF3)

Name	Salary
Anne	1700.00
Doug	950.00
Mark	2100.00

The program reads the following logical file records.

Table 30. Join logical file (JLF)

Name	Address	Telephone	Salary
Anne	120 1st St.	555–1111	1700.00
Doug	40 Pillsbury	555–5555	950.00
Mark	2 Lakeside Dr.	555–0000	2100.00

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.

Example 8: Joining a physical file to itself:

This example shows how to use a join logical file to combine records from one physical file.

The following chart shows how you can join a physical file to itself.

RBAFO532-0
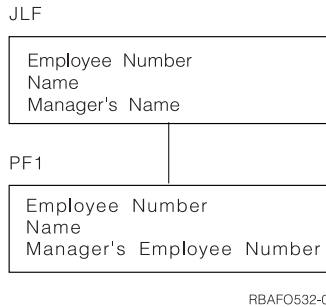
The join logical file (JLF) contains *Employee number*, *Name*, and *Manager's name*. The physical file (PF1) contains *Employee number*, *Name*, and *Manager's employee number*. The following example shows the data description specifications (DDS) for these files:

JLF
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    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
```

PF1
```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A          R RCD1
    A            NBR           3
    A            NAME         10        DFT('none')
    A            MGRNBR        3
    A
```

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 that the following records are contained in PF1.

Table 31. Physical file 1 (PF1)

Employee number	Name	Manager's employee number
235	Anne	440
440	Doug	729

Table 31. Physical file 1 (PF1) (continued)

Employee number	Name	Manager's employee number
500	Mark	440
729	Sue	888

The program reads the following logical file records.

Table 32. Join logical file (JLF)

Employee number	Name	Manager's name
235	Anne	Doug
440	Doug	Sue
500	Mark	Doug
729	Sue	none

Notes:

1. A record is returned for the manager name of Sue because the JDFTVAL keyword was specified.
2. The value `none` is returned because the DFT keyword was used on the *Name* field in the PF1 physical file.

Example 9: Using defaults for missing records from secondary files:

This example shows how to use the default value for a missing join field in a secondary file to join to the other secondary file.

If the DFT keyword is specified in the secondary file, the value specified for the DFT keyword is used in the logical file.

The data description specifications (DDS) for the files are shown as follows:

```
JLF
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  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
```

```
PF1
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  A           R RCD1
  A             NAME         20
  A             ADDR         40
  A             COUNTRY      40
  A
```

```
PF2
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  A           R RCD2
  A             NAME         20
  A             TELEPHONE     8       DFT('999-9999')
```

```
      A

PF3
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      A            R RCD3
      A              TELEPHONE     8
      A              LOC           30          DFT('No location assigned')
      A
```

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

Table 33. Physical file 1 (PF1)

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

Table 34. Physical file 2 (PF2)

Name	Telephone
Anne	555–1234
Doug	555–2222
Sue	555–1144

Table 35. Physical file 3 (PF3)

Telephone	Location
555–1234	Room 312
555–2222	Main lobby
999–9999	No telephone number

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

Table 36. Join logical file (JLF)

Name	Address	Telephone	Location
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

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).

Example 10: A complex join logical file:

This example shows a more complex join logical file.

Assume that the data is in the following three physical files:

`Vendor Master File (PF1)`

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     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
```

`Order File (PF2)`

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R RCD2                     TEXT('VENDORS ORDER')
     A            VDRNUM        5S 0        TEXT('VENDOR NUMBER')
     A            JOBNBR        6           TEXT('JOB NUMBER')
     A            PRTNBR        5S 0        TEXT('PART NUMBER')
     A                                      DFT(99999)
     A            QORDER        3S 0        TEXT('QUANTITY ORDERED')
     A            UNTPRC        6S 2        TEXT('PRICE')
     A
```

`Part File (PF3)`

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R RCD3                     TEXT('DESCRIPTION OF PARTS')
     A            PRTNBR        5S 0        TEXT('PART NUMBER')
     A                                      DFT(99999)
     A            DESCR         25          TEXT('DESCRIPTION')
     A            UNITPRICE     6S 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
```

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 data description specifications (DDS) for this join logical file are shown as follows:

```
Join Logical File (JLF)
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A                                 1  DYNSLT
     A                                 2  JDFTVAL
     A           R RECORD1                JFILE(PF1 PF2 PF3)
     A     3  J                           JOIN(1 2)
     A                                    JFLD(VDRNBR VDRNUM)
     A                                 4  JDUPSEQ(JOBNBR)
     A     5  J                           JOIN(2 3)
     A                                 6  JFLD(PRTNBR PRTNBR)
     A                                    JFLD(UNTPRC UNITPRICE)
     A       7  VDRNUM        5A  N       TEXT('CHANGED ZONED TO CHAR')
     A           VDRNAM
     A           ADDRESS                8  CONCAT(STREET CITY STATE +
     A                                       ZIPCODE)
     A           JOBNBR
     A           PRTNBR                 9  JREF(2)
     A           DESCR
     A           QORDER
     A           UNTPRC
     A           WHSNBR
     A           PRTLOC
     A    10  S VDRNAM                     COMP(EQ 'SEWING COMPANY')
     A        S QORDER                     COMP(GT 5)
     A
```

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 that 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: They come from two different physical files.)

Join logical file considerations:

Consider performance, data integrity, and rules for join logical files.

Performance considerations:

Use these tips to improve performance of join logical files.
- If the physical files you join 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.
- Consider describing your join logical file so it can automatically share an existing access path.

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.

Related concepts

"Dynamic select/omit" on page 55
With the dynamic select/omit operation, the system only returns those records that meet the select/omit values when a program reads records from the file. 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.

"Sharing existing access paths between logical files" on page 55
When two or more logical files are based on the same physical files with the same key fields in the same order, they automatically share the same keyed sequence access path.

Data integrity considerations:

These situations can occur unless you lock the physical files that are used by the join logical file.

- 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:

This is a summary of rules for joining database files.

Requirements:

Here are the principle requirements for join logical files.

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

Join fields:

Here are the rules for join fields.
- Every physical file you join 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. If the join fields are of character type, the field lengths might be different.
- 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.
- The maximum length of fields used in joining physical files is equal to the maximum size of keys for physical and logical files.

  Related reference

  "Database file sizes" on page 7
  Before you design and create a database file, you need to know the maximum size allowed for the file.

Fields in join logical files:

Here are the rules for fields in join logical files.
- 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 join logical file, but in some circumstances the DYNSLT keyword is required.
- Fields specified on the select/omit statements cannot be fields defined as neither fields (N specified in position 38 for the field).
- 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.

Describing access paths for database files

An access path describes how records in a database file are retrieved. You can define the access path for a database file in various ways.

Records in a physical or logical file can be retrieved through an arrival sequence access path or a keyed sequence access path. For logical files, you can also select and omit records based on one or more field values in each record. A key field is a field that is used to arrange the records of a particular type within a file member.

> **Related concepts**
>
> "Access path description" on page 6
> An access path of a database file describes the order in which records are to be retrieved. When you describe an access path, you describe whether it is a keyed sequence access path or an arrival sequence access path.

Using arrival sequence access paths for database files

An arrival sequence access path is based on the order in which records arrive and are stored in a database 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 files:
- Physical files
- Logical files in which each member of the logical file is based on only one physical file member
- Join logical files
- Views

You can use arrival sequence access paths in the following ways:
- 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. This method requires explicit insertion of a record given a relative record number that you provide. Another method, in which the system manages the space created by deleting records, is the reuse deleted records attribute that can be specified for physical files.
- Through your high-level language, the Display Physical File Member (DSPPFM) command, and the Copy File (CPYF) command, you can 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.
- Through your high-level language, you can 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.
- 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.)

> **Related concepts**
>
> "Reusing deleted records" on page 101
> Sometimes you might want to reuse deleted records for your database files. In this case, you can use the REUSEDLT parameter.

The delete operation allows you to delete an existing database record.

Using keyed sequence access paths for database files

A keyed sequence access path is based on the contents of the key fields as defined in data description specifications (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 UNSIGNED (unsigned value) or ABSVAL (absolute value) DDS 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 an alternative collating sequence:

You can arrange key fields that are defined as character fields either in a sequence for EBCDIC characters or in an alternative collating sequence.

Consider the following records.

Record	Empname	Deptnbr	Empnbr
1	Jones, Mary	45	23318
2	Smith, Ron	45	41321
3	JOHNSON, JOHN	53	41322
4	Smith, ROBERT	27	56218
5	JONES, MARTIN	53	62213

If the *Empname* is the key field and is a character field, using the sequence for EBCDIC characters, the records can be arranged as follows.

Record	Empname	Deptnbr	Empnbr
1	Jones, Mary	45	23318
3	JOHNSON, JOHN	53	41322
5	JONES, MARTIN	53	62213
2	Smith, Ron	45	41321
4	Smith, ROBERT	27	56218

Notice that the EBCDIC sequence causes an unexpected sort order because the lowercase characters are sorted before uppercase characters. Thus, `Smith, Ron` sorts before `Smith, ROBERT`. An alternative collating sequence can be used to sort the records when the records were entered using uppercase and lowercase as shown in the following example.

Record	Empname	Deptnbr	Empnbr
3	JOHNSON, JOHN	53	41322
5	JONES, MARTIN	53	62213
1	Jones, Mary	45	23318
4	Smith, ROBERT	27	56218
2	Smith, Ron	45	41321

To use an alternative collating sequence for a character key field, specify the ALTSEQ DDS keyword, and specify the name of the table containing the alternative collating sequence. When setting up a table, each 2-byte position in the table corresponds to a character. To change the order in which a character is sorted, change its 2-digit value to the same value as the character it should be sorted equal to. For information about sorting uppercase and lowercase characters regardless of their case, the QCASE256 table in library QUSRSYS is provided for you.

> **Related concepts**
>
> DDS concepts

Arranging key fields with the SRTSEQ parameter:

You can arrange key fields that contain character data in a sort sequence order by using the SRTSEQ parameter.

Consider the following records.

Record	Empname	Deptnbr	Empnbr
1	Jones, Marilyn	45	23318
2	Smith, Ron	45	41321
3	JOHNSON, JOHN	53	41322
4	Smith, ROBERT	27	56218
5	JONES, MARTIN	53	62213
6	Jones, Martin	08	29231

If the *Empname* field is the key field and is a character field, the *HEX sequence (the EBCDIC sequence) arranges the records as follows.

Record	Empname	Deptnbr	Empnbr
1	Jones, Marilyn	45	23318
6	Jones, Martin	08	29231
3	JOHNSON, JOHN	53	41322
5	JONES, MARTIN	53	62213
2	Smith, Ron	45	41321
4	Smith, ROBERT	27	56218

Notice that with the *HEX sequence, all lowercase characters are sorted before the uppercase characters. Thus, Smith, Ron sorts before Smith, ROBERT, and JOHNSON, JOHN sorts between the lowercase and uppercase Jones. You can use the *LANGIDSHR sort sequence to sort records when the records were entered using a mixture of uppercase and lowercase. The *LANGIDSHR sequence, which uses the same collating weight for lowercase and uppercase characters, results in the following record.

Record	Empname	Deptnbr	Empnbr
3	JOHNSON, JOHN	53	41322
1	Jones, Marilyn	45	23318
5	JONES, MARTIN	53	62213
6	Jones, Martin	08	29231
4	Smith, ROBERT	27	56218
2	Smith, Ron	45	41321

Notice that with the *LANGIDSHR sequence, the lowercase and uppercase characters are treated as equal. Thus, JONES, MARTIN and Jones, Martin are equal and sort in the same sequence they have in the base file. While this is not incorrect, it would look better in a report if all the lowercase Jones preceded the uppercase JONES. You can use the *LANGIDUNQ sort sequence to sort the records when the records

were entered using an inconsistent uppercase and lowercase. The *LANGIDUNQ sequence, which uses different but sequential collating weights for lowercase and uppercase characters, results in the following record.

Record	Empname	Deptnbr	Empnbr
3	JOHNSON, JOHN	53	41322
1	Jones, Marilyn	45	23318
6	Jones, Martin	08	29231
5	JONES, MARTIN	53	62213
4	Smith, ROBERT	27	56218
2	Smith, Ron	45	41321

The *LANGIDSHR and *LANGIDUNQ sort sequences exist for every language supported in your system. The LANGID parameter determines which *LANGIDSHR or *LANGIDUNQ sort sequence to use. Use the SRTSEQ parameter to specify the sort sequence and the LANGID parameter to specify the language.

Arranging key fields in ascending or descending sequence:

You can arrange key fields in either ascending or descending sequence. When you describe a key field, the default is ascending sequence. However, you can use the Descend (DESCEND) DDS keyword to specify that you want to arrange a key field in 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 shown in the following table.

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 shown in the following table.

Record	Empnbr	Clsnbr	Clsnam	Cpdate
3	64002	412	Welding I	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

Record	Empnbr	Clsnbr	Clsnam	Cpdate
4	23318	412	Welding I	032188

Using more than one key field:

You can use more than one key field to arrange records in a database 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:

DB2 for i5/OS allows records with duplicate key values in a database file. However, you can prevent duplicate key values in 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 data description specifications (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. You can also use unique constraints to enforce the integrity of unique keys.

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.

If any of the key fields allow null values, null values that are inserted into those fields might or might not cause duplicates depending on how the access path was defined when the file was created. The *INCNULL parameter of the UNIQUE keyword indicates that null values are included when determining whether duplicates exist in the unique access path. The *EXCNULL parameter indicates that null values are not included when determining whether duplicate values exist.

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

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
   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
```

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.

> **Related concepts**
>
> "Controlling the integrity of your database with constraints" on page 250
> A *constraint* is a restriction or limitation placed on a database file to ensure that the data in your database remains consistent when you add, change, and remove records.
>
> DDS concepts

Arranging duplicate keys:

If you do not specify the Unique (UNIQUE) keyword in data description specifications (DDS), you can specify how the system stores records with duplicate key values.

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 (**1**), records with duplicate key values are retrieved in LIFO order by the physical sequence of the records. Here is an example of DDS using the LIFO keyword.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A* ORDERP2
    A                             1  LIFO
    A           R ORDER2
    A             .
    A             .
    A             .
    A           K ORDER
    A
```

- First-in-first-out (FIFO). If the FIFO keyword is specified, records with duplicate key values are retrieved in FIFO 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 FCFO order by the physical sequence of the keys.
- No specific order for duplicate key fields (the default). When the FIFO, FCFO, or LIFO keyword is 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.

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 of 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 DDS concepts.

The LIFO or FIFO order of records with duplicate key values is not determined by the sequence of the updates made to the contents of the key fields, but only 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 FIFO order), and that the following table 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, with ascending key values.

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 can 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, with descending key values.

Record number	Key value
5	D
3	C
4	C
2	B
1	A

If the key value of physical record 1 is changed to C, the sequence of the access path for the physical file is FIFO, with ascending key values.

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, with descending key values.

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 preceding example, after record 1 is changed such that the key value is C, the sequence of the access path is FCFO, with ascending key values only.

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.

If the reuse deleted records attribute is specified for a physical file, the duplicate key ordering must be allowed to default or must be FCFO. The reuse deleted records attribute is not allowed for the physical file if either the key ordering for the file is FIFO or LIFO, or if any of the logical files defined over the physical file have duplicate key ordering of FIFO or LIFO.

> **Related concepts**
>
> "Sharing existing access paths between logical files" on page 55
> When two or more logical files are based on the same physical files with the same key fields in the same order, they automatically share the same keyed sequence access path.

Using existing access path specifications

You can use the Reference Access Path Definition (REFACCPTH) DDS keyword to copy another file's access path specifications. After you create the file, 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 only 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 binary floating-point fields in database file access paths

The collating sequence for records in a keyed database file depends on the presence of the Signed (SIGNED), Unsigned (UNSIGNED), and Absolute Value (ABSVAL) keywords in data description specifications (DDS). For binary floating-point fields, the sign is the farthest-left bit, the exponent is next, and the significand is last.

Here is the collating sequence with the UNSIGNED keyword specified:
- Positive real numbers, positive infinity
- Negative real numbers, negative infinity

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

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

The following binary floating-point collating sequences are observed:
- Zero (positive or negative) collates in the same manner as any other positive or 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.

Securing database files

You can secure database files in various ways.

> **Related concepts**
>
> Security reference

Granting file and data authority

DB2 for i5/OS provides several methods to grant file and data authority.

- You can use System i Navigator to authorize a user or group.
- You can use the Grant Object Authority (GRTOBJAUT) command to specify the authority you want users to have to access data in your database files.
- You can use the SQL GRANT statement.

> **Related reference**
>
> Grant Object Authority (GRTOBJAUT) command
>
> GRANT (Table or View Privileges)

Authorizing a user or group using System i Navigator:

Some users might need specific authority to an object. You can grant specific authority to a user or group using System i Navigator.

To authorize a user or group to an object, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Navigate until the object for which you want to edit permissions is visible.
4. Right-click the object for which you want to add permissions, and click **Permissions**.
5. In the Permissions window, click **Add**.
6. In the Add Users and Groups window, either specify or select one or more users or groups.
7. Click **OK** to close the Add Users and Groups window. This adds one or more users or groups to the authorities list.
8. Click **OK** to close the Permissions window.

Note: The user or group is given the default authority to the object. You can change a user's authority to one of the types defined by the system or you can customize the authority.

You can also remove and customize authority using System i Navigator.

Types of object authority:

An object authority controls what a user can do with an entire object.

The types of object authority follow:

Object operational authority

Users need object operational authority 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

Users need object existence authority 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

Users need object management authority to:
- Create a logical file with a keyed sequence 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.)

Object alter authority

Users need object alter authority for many of the same operations as object management authority (see preceding section). Object alter authority is a replacement authority for object management authority.

Object reference authority

Users need object reference authority to refer to an object from another object so that the operations on that object can be restricted by the referencing object.

Adding a physical file referential constraint checks for either object management authority or object reference authority to the parent file.

   Related concepts

"Controlling the integrity of your database with constraints" on page 250
A *constraint* is a restriction or limitation placed on a database file to ensure that the data in your database remains consistent when you add, change, and remove records.

"Ensuring data integrity with referential constraints" on page 255
You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

Types of data authority:

A data authority is a specific authority to read, add, update, or delete data in a database file, to run a program, or to search a library or directory.

The types of data authority follow:

Read authority

Users can read the records in the file.

Add authority

Users can add new records to the file.

Update authority

Users can update existing records. (To read a record for update, you must also have read authority.)

Delete authority

Users can delete existing records. (To read a record for deletion, you must also have read authority.)

Execute authority

You can use execute authority to work with libraries and to start programs. For example, if you are changing a file associated with a trigger, you must have execute authority to the trigger program. If you do not have execute authority, the system will not start the trigger program.

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.

If object operational authority is not granted to a user for a file, the user cannot open the 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 (*OBJOPR), read (*READ), and add (*ADD) 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: The user was not given authority for LF3 and, therefore, cannot use it.
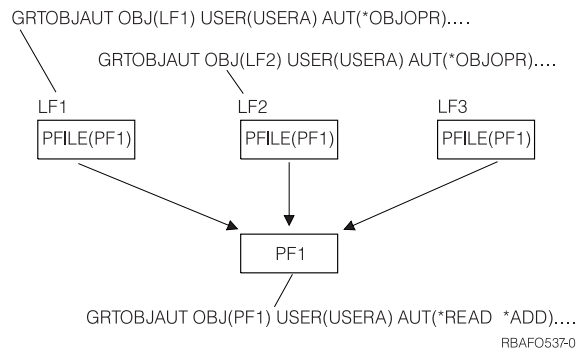
```
GRTOBJAUT OBJ(LF1) USER(USERA) AUT(*OBJOPR)....

        GRTOBJAUT OBJ(LF2) USER(USERA) AUT(*OBJOPR)....

  LF1                    LF2                   LF3
┌──────────┐      ┌──────────┐      ┌──────────┐
│PFILE(PF1)│      │PFILE(PF1)│      │PFILE(PF1)│
└──────────┘      └──────────┘      └──────────┘
         \              │              /
          \             │             /
           \            ▼            /
            ┌──────────────┐
            │     PF1      │
            └──────────────┘
          /
 GRTOBJAUT OBJ(PF1) USER(USERA) AUT(*READ  *ADD)....
                                        RBAFO537-0
```

Related concepts

"Triggering automatic events in your database" on page 267
A *trigger* is a set of actions that run automatically when a specified change or read operation is performed on a specified database file. You can define a set of trigger actions in any high-level language that is supported on the i5/OS operating system.

Specifying public authority

Public authority is given to users who do not have any specific authority to an object, who are not on the authorization list of the object, or whose group profile has no specific authority to the object. When you create a file, you can specify and grant public authority.

You can specify public authority through the AUT parameter on the Create Physical File (CRTPF) or Create Source Physical File (CRTSRCPF) command. 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:

- *LIBCRTAUT. The library in which the file is created is checked to determine the public authority of the file when the file is created. An authority is associated with each library. This authority is specified when the library is created, and all files created into the library are given this public authority if the *LIBCRTAUT value is specified for the AUT parameter of the Create File (CRTLF, CRTPF, and CRTSRCPF) commands. The *LIBCRTAUT value is the default public authority.
- *CHANGE. All users that do not have specific user or group authority to the file have authority to change data in the file.
- *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. The authorization list is a list of users and their authorities. The list allows users and their different authorities to be grouped together.

Note: When you create 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 grant public authority in the following ways:
- Define public authority using System i Navigator.
- Use the Edit Object Authority (EDTOBJAUT), Grant Object Authority (GRTOBJAUT), or Revoke Object Authority (RVKOBJAUT) command to grant or revoke the public authority of a file.

You can also use System i Navigator to set default public authority for a new file.

Related reference

Create Physical File (CRTPF) command

Create Source Physical File (CRTSRCPF) command

Edit Object Authority (EDTOBJAUT) command

Grant Object Authority (GRTOBJAUT) command

Revoke Object Authority (RVKOBJAUT) command

Defining public authority using System i Navigator:

Public authority is defined for every object on the system to describe what type of access a user has to the object when that user has no specific authority to it. You can define public authority for a database file using System i Navigator.

To define public authority, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Navigate until the object for which you want to edit permissions is visible.
4. Right-click the object for which you want to add permissions, and click **Permissions**.
5. In the Permissions window, select **Public** from the group list.
6. From the **Authorities view** list, select **Details** to implement detailed permissions. Select the appropriate check box to apply the permissions that you want for public authority.
7. Click **OK** to close the Permissions window.

Setting a default public authority using System i Navigator:

By setting a default public authority, you can have a common authority to all new objects when they are created in a library. You can edit the permissions for individual objects that require a different level of security.

To set a default public authority, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas**.
4. Right-click the schema for which you want to set a public authority, and click **Permissions**.
5. In the Permissions window, click **New Object**.
6. In the New Object window, select a default public authority. To use an authorization list, click **Browse** for the name of the authorization list. To view authorization list properties, click **Edit**.
7. Click **OK** to close the Permissions window.

Here is a list of default public authorities:

Use system value
Specifies to use the system value for the default public authority for a new object.

Use
Allows access to the object attributes and use of the object. The public can view, but not change, the objects.

Change
Allows the contents of the object (with some exceptions) to be changed.

All
Allows all operations on the object, except those that are limited to the owner. The user or group can control the existence of the object, specify the security for the object, change the object, and perform basic functions on the object. The user or group can also change the ownership of the object.

Exclude
   Prohibits all operations on the object. No access nor operations are allowed to the object for the users
   and groups that have this permission. Specifies that the public is not allowed to use the object.

Use authorization list
   Allows you to specify an authorization list to use to secure the object.

Using database file capabilities to control I/O operations

When you create a physical file, you can specify that the file capabilities control which input/output
(I/O) operations are allowed for the file, independent of database file authority.

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 after 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 and then re-create it
with the capability that you want. The Display File Description (DSPFD) command can be used to
determine the capabilities of a file.

   Related reference

   Create Physical File (CRTPF) command

   Create Source Physical File (CRTSRCPF) command

   Display File Description (DSPFD) command

Limiting access to specific fields in a database file

You can restrict update and read requests to the specific fields in a database file.

Use one of the following ways to limit access to the specific fields in a database file:
- Create a logical view of the physical file that includes only the fields to which you want your users to
  have access.
- Use the SQL GRANT statement to grant update authority to specific columns of a Structured Query
  Language (SQL) table.

   Related concepts

   "Using logical files to secure data"
   You can use logical files to prevent data in physical files from being read or changed.

   SQL programming

Using logical files to secure data

You can use logical files to prevent data in physical files from being read or changed.

You can use a logical file to prevent a field in a physical file from being read. This is accomplished by
describing a logical file record format that does not include fields that you do not want the user to see.

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 form.

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.

To read data in a physical file through a logical file, you need object operational and read authorities to the logical file and read authority to the physical file.

To change (add, update, or delete) data in a physical file through a logical file, you need object operational and appropriate data authorities to the logical file and appropriate data authority to the physical file.

To change file authorities, you can use the Edit Object Authority (EDTOBJAUT) command.

The following example illustrates how logical files can be used to secure data:

1. Create a physical file.

   a. Enter DDS for physical file PF into source file DDSSRC. DDSSRC is in the SAMPLE library:

   ```
   |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
   00010A          R FORMAT1
   00020A            F1            1A
   00030A            F2            1A
   00040A            F3            1A
   00050A            F4            1A
   ```

   b. Create PF in the SAMPLE library:

   ```
   CRTPF FILE(SAMPLE/PF) SRCFILE(SAMPLE/DDSSRC)
   ```

   c. Insert the following records into PF:

   ```
   INSERT INTO SAMPLE/PF VALUES('A', 'A', 'A', 'A')
   INSERT INTO SAMPLE/PF VALUES('B', 'B', 'B', 'B')
   ```

2. Create a logical file to allow access to only some fields in the physical file.

   a. Enter DDS for logical file LF1 into source file DDSSRC:

   ```
   |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
   00010A          R FORMAT1                    PFILE(SAMPLE/PF)
   00020A            F1
   00030A            F2
   ```

   b. Create LF1 in the SAMPLE library:

   ```
   CRTLF FILE(SAMPLE/LF1) SRCFILE(SAMPLE/DDSSRC)
   ```

   c. Use the following SQL statement:

   ```
   SELECT * from SAMPLE/LF1
   ```

   Only data for fields F1 and F2 is returned.

3. Create a logical file to allow access to only the data that meets the specified condition in the physical file.

   a. Enter DDS for logical file LF2 into source file DDSSRC:

   ```
   |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
   00010A          R FORMAT1                    PFILE(SAMPLE/PF)
   00020A            F1
   00030A            F2
   00040A          K F1
   00050A          S F1                         COMP(EQ 'A')
   ```

   b. Create LF2 in the SAMPLE library:

   ```
   CRTLF FILE(SAMPLE/LF2) SRCFILE(SAMPLE/DDSSRC) SRCMBR(LF1)
   ```

   c. Use the following SQL statement:

   ```
   SELECT * from SAMPLE/LF2
   ```

   Only records where F1 equals 'A' are returned.

Related concepts

"Describing logical file record formats" on page 45
For every logical file record format described with data description specifications (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).

"Describing field use for logical files" on page 47
You can specify that the fields in a logical file be input-only, both (input/output), or neither fields.

"Selecting and omitting records for logical files" on page 52
You can select and omit records for a logical file. This helps exclude records from a file for processing convenience or for security.

"Types of object authority" on page 92
An object authority controls what a user can do with an entire object.

"Types of data authority" on page 94
A data authority is a specific authority to read, add, update, or delete data in a database file, to run a program, or to search a library or directory.

Related reference
Edit Object Authority (EDTOBJAUT) command

Processing database files

DB2 for i5/OS provides various processing options for you to open, manipulate, and close database files. You can also monitor database file errors in your program.

Database file processing: Runtime considerations

Before a database file is opened for processing, you need to consider how you want to use the file in the program or job. A better understanding of the runtime file processing parameters can help you avoid unexpected results and 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 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.

These topics describe the file processing parameters and other methods or considerations that can be used to affect database file processing. 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.

Related concepts

 ILE Concepts PDF

Control language

Related reference

Create Physical File (CRTPF) command

Create Logical File (CRTLF) command

Create Source Physical File (CRTSRCPF) command

Add Physical File Member (ADDPFM) command

Add Logical File Member (ADDLFM) command

Change Physical File (CHGPF) command

Change Physical File Member (CHGPFM) command

Change Logical File (CHGLF) command

Change Logical File Member (CHGLFM) command

Change Source Physical File (CHGSRCPF) command

Override with Database File (OVRDBF) command

Open Database File (OPNDBF) command

Open Query File (OPNQRYF) command

Close File (CLOF) command

File and member name

Before you can process data in a database file, you must identify which file and member you want to use with the FILE and MBR parameters.

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 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

DB2 for i5/OS provides several runtime file processing options.

Specifying the type of processing:

When you use a database file in a program, the system needs to know what type of operation you plan to use for the file. You can specify the type of processing with the OPTION parameter.

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 the 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 OPNQRYF or OPNDBF command, 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.

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.

Related concepts

"Types of data authority" on page 94
A data authority is a specific authority to read, add, update, or delete data in a database file, to run a program, or to search a library or directory.

"Locking shared data" on page 105
By definition, all database files can be used by many users at the same time. However, some operations can lock files, members, or records to prevent them from being shared across jobs.

Specifying the initial file position:

After a database file is opened, the system needs to know where it should start processing the file. You can specify the initial file position with the POSITION parameter.

The default is to start just before the first record in the file (the first sequential read operation reads 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.

Related concepts

"Setting a position in the file" on page 192
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.

Reusing deleted records:

Sometimes you might want to reuse deleted records for your database files. In this case, you can use the REUSEDLT parameter.

When you specify REUSEDLT(*YES) on the Create Physical File (CRTPF) or Change Physical File (CHGPF) command, the following operations might work differently:

- Arrival order becomes meaningless for a file that reuses deleted record space. Records might not be added at the end of the file.
- End-of-file delay does not work for the files that reuse deleted record space.
- One hundred percent reuse of deleted record space is not guaranteed. A *file full* condition might be reached or the file might be extended even though deleted record space still exists in the file.

Note: Because of the way the system reuses deleted record space, the following types of files should not be created or changed to reuse deleted record space:

  - Files processed using relative record numbers, and files used by an application to determine a relative record number that is used as a key into another file
  - Files used as queues
  - Any files used by applications that assume new record insertions are at the end of the file
  - When DB2 Symmetric Multiprocessing is installed, files on which you expect to have parallel index maintenance performed when rows are updated, inserted, or deleted

If you decide to change an existing physical file to reuse deleted record space, and there are logical files with access paths with first-in-first-out (FIFO) or last-in-first-out (LIFO) duplicate key ordering over the physical file, you can re-create the logical files without the FIFO or LIFO attribute and avoid rebuilding the existing access path by following these steps:

1. Rename the existing logical file that has the FIFO or LIFO attribute.

2. Create a second logical file identical to the renamed file except that duplicate key ordering should not be specified for the file. Give the new file the original file name. The new file shares the access path of the renamed file.

3. Delete the renamed file.

Ignoring the keyed sequence access path:

If a key field is defined for a database file, the system automatically uses the keyed sequence access path. However, sometimes you can use the ACCPTH parameter to ignore the keyed sequence access path for better performance.

You can use the ACCPTH parameter to ignore the keyed sequence access path and process 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 were 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:

When your program reaches the end of a database file, the system normally signals your program that there is no more data to read. If you want the system to hold your program until more data arrives in the file, use the EOFDLY parameter to delay the end-of-file processing.

If you use the EOFDLY parameter on the Override with Database File (OVRDBF) command, the program can read the newly arrived records when more data arrives in the file.

Note: End-of-file delay should not be used for files that reuse deleted records.

    Related concepts

    "Waiting for more records when end of file is reached" on page 196
    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.

Specifying the record length:

As an option, you can specify the record length in your high-level language program.

The system needs to know the length of the record your program will be processing, but it is not required that you specify the record length in your program. The system automatically determines this information from the attributes and description of the file named in your 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 that you want to use all the formats defined for the file. However, if you do not want to use all the formats, you can specify which formats you want to use and 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 topic collection.

Determining whether duplicate keys exist:

The set of keyed sequence access paths used to determine whether the key is a duplicate key differs depending on the input/output (I/O) operation that is performed. You can determine whether duplicate keys exist using the DUPKEYCHK parameter.

For input operations (reads), the keyed sequence access path used is the one that the file is opened with. Any other keyed sequence access path 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 nonunique 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. Only keyed sequence access paths that have *REBLD and *DLY maintenance are considered if the access paths are actively open over the file at feedback time.

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) command. However, in COBOL, having duplicate key feedback returned can cause a decline in performance.

Data recovery and integrity

You can specify several file processing parameters for data recovery and integrity.

Protecting your files with journaling and commitment control:

Journaling and commitment control are the preferred methods for i5/OS data and transaction recovery. You can use the COMMIT parameter to protect your file with journaling and commitment control.

Database file journaling is started by running the Start Journal Physical File (STRJRNPF) command for the file. Access path journaling is started by running the Start Journal Access Path (STRJRNAP) command for the file or by using System-Managed Access-Path Protection (SMAPP). 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.

If you are performing insert, update, or delete operations on a file that is associated with a referential constraint and the delete rule, update rule, or both are other than RESTRICT, you must use journaling.

> **Related concepts**
>
> "Managing journals" on page 226
> Journal management allows you to record all the data changes that occur to one or more database files. You can then use the journal for recovery.
>
> "Ensuring data integrity with commitment control" on page 234
> Commitment control allows you to define and process a number of changes to database files in a single unit (transaction).

"Ensuring data integrity with referential constraints" on page 255
You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

Writing data and access paths to auxiliary storage:

Normally, DB2 for i5/OS determines when to write changed data and access paths from main storage to auxiliary storage. However, you can control when database changes are written to auxiliary storage.

If you want to control when changed data are written from main storage to auxiliary storage, you can use the force write ratio (FRCRATIO) parameter on 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.

    Related concepts

    "Recovering and restoring your database" on page 226
    You can use several i5/OS save and restore functions to recover your database after the system loses data.

Checking changes to the record format description:

When a database file is opened, the system checks whether the description of the record format has been changed since the program was compiled to an extent that it cannot process the file.

The system normally notifies your program of a level check. When you use the create or change file command, 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 Override with Database File (OVRDBF) command.

    Related concepts

    "Effects of changing fields in a file description" on page 222
    The system uses the information in the record format description to determine the level identifier. Changes to the fields in a file description cause the level identifier to change. Changes in key fields or select/omit fields might cause unexpected results in programs using the new access path.

Checking the expiration date of a physical file member:

The system can check whether the data in a physical file member is still current. You can specify whether the system checks the expiration date of a physical file member using the EXPCHK parameter on the Override with Database File (OVRDBF) command.

If you check the expiration date and the current date is greater than the expiration date, a message is sent to the system operator when the physical file member is opened.

    Related concepts

    "Expiration date" on page 34
    The EXPDATE parameter specifies an expiration date for each member in a physical file (ADDPFM, CHGPFM, CRTPF, CHGPF, CRTSRCPF, and CHGSRCPF commands).

Preventing the job from changing data in a file:

If you want to test your program, but do not want to change data in a file that is used for the test, you can tell the system not to write (inhibit) any changes to the file that the program attempts to make. For this operation, you can use the INHWRT parameter.

To inhibit any changes to the file, specify INHWRT(*YES) on the Override with Database File (OVRDBF) command.

Locking shared data

By definition, all database files can be used by many users at the same time. However, some operations can lock files, members, or records to prevent them from being shared across jobs.

When files, members, or records are locked, no other jobs can read the same data for update operations, which keeps another job from unintentionally deleting the first job's update.

You can lock a row by opening a table from System i Navigator and editing the row that you want to lock or by using the SQL LOCK TABLE statement. In addition to rows, you can lock files, members, and data records. You can display locked records using the Display Record Locks (DSPRCDLCK) command or System i Navigator.

> **Related reference**
> LOCK TABLE

Locking records:

DB2 for i5/OS has built-in integrity for records.

Here is an example of built-in integrity for records. If PGMA reads a record for an update operation, it locks the record. Another program cannot read the same record for an update operation until PGMA releases the record, but another program can read the record just for inquiry.

In this way, the system ensures the integrity of the database. However, you can set your own wait time for a locked record to be released using the WAITRCD parameter.

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 can send a message to the operator that the requested record is currently unavailable).

If the record lock is being implicitly acquired as a result of a referential integrity CASCADE DELETE, SET NULL, or SET DEFAULT delete rule, the lock wait time is limited to 30 seconds.

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 about how to release a record lock, see your high-level language topic collection.

Note: Using commitment control changes the record locking rules.

You can display locked records using either the Display Record Locks (DSPRCDLCK) command or System i Navigator.

> **Related concepts**
> Commitment control

"Displaying locked records using the Display Record Locks (DSPRCDLCK) command" on page 108
You can use the Display Record Locks (DSPRCDLCK) command to display the current lock status
(wait or held) of records in a physical file member.

Related tasks

"Displaying locked rows using System i Navigator" on page 108
You can use System i Navigator to display locked rows in a table.

Locking files:

When a database file is allocated exclusively, any program trying to open the file has to wait until it is
released. However, you can set a wait time for the file to become available using the WAITFILE
parameter.

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.

Locking members:

Member operations (such as add and remove) automatically allocate the file exclusively to prevent other
file operations from occurring at the same time.

Input/output (I/O) operations on the same member cannot run, but I/O operations on other members of
the same file can run at the same time.

Locking record format data:

Sometimes you might want to lock the entire set of records associated with a record format (for example,
all the records in a physical file). In this case, you can use the RCDFMTLCK parameter on the Override
Database File (OVRDBF) command.

Database lock considerations:

These tables show the types of locks that some of the database functions place on database files, the valid
lock combinations, and the types of locks for constraints.

Table 37 summarizes some of the most commonly used database functions and the types of locks they
place on database files. The types of locks are explained on the next page.

Table 37. Database functions and locks

Function	Command	File lock	Member/Data lock	Access path lock
Add Member	ADDPFM, ADDLFM	*EXCLRD		*EXCLRD
Change File Attributes	CHGPF, CHGLF	*EXCL	*EXCLRD	*EXCLRD
Change Member Attributes	CHGPFM, CHGLFM	*SHRRD	*EXCLRD	
Change Object Owner	CHGOBJOWN	*EXCL		
Check Object	CHKOBJ	*SHRNUPD		

Table 37. Database functions and locks (continued)

Function	Command	File lock	Member/Data lock	Access path lock
Clear Physical File Member	CLRPFM	*SHRRD	*EXCLRD[3]	
Create Duplicate Object	CRTDUPOBJ	*EXCL (new object) *SHRNUPD (object)		
Create File	CRTPF, CRTLF, CRTSRCPF	*EXCL		
Delete File	DLTF	*EXCL		*EXCLRD
Grant/Revoke Authority	GRTOBJAUT, RVKOBJAUT	*EXCL		
Initialize Physical File Member	INZPFM	*SHRRD	*EXCLRD	
Move Object	MOVOBJ	*EXCL		
Open File	OPNDBF, OPNQRYF	*SHRRD	*SHRRD	*EXCLRD
Rebuild Access Path	EDTRBDAP, OPNDBF	*SHRRD	*SHRRD	*EXCLRD
Remove Member	RMVM	*EXCLRD	*EXCL	*EXCLRD
Rename File	RNMOBJ	*EXCL	*EXCL	*EXCL
Rename Member	RNMM	*EXCLRD	*EXCL	*EXCL
Reorganize Physical File Member	RGZPFM	*SHRRD	*EXCL[4]	
Restore File	RSTLIB, RSTOBJ	*EXCL		
Save File	SAVLIB, SAVOBJ, SAVCHGOBJ	*SHRNUPD[1]	*SHRNUPD[2]	

[1] For save-while-active, the file lock is *SHRUPD initially, and then the lock is reduced to *SHRRD. See Save-while-active function for a description of save-while-active locks for the save commands.

[2] For save-while-active, the member/data lock is *SHRRD.

[3] The clear operation does not happen if the member is open in this process or in any other process.

[4] If ALWCANCEL(*YES) is specified, the LOCK keyword can specify a *SHRUPD or *EXCLRD lock instead.

The following table shows the valid lock combinations:

Lock	*EXCL	*EXCLRD	*SHRUPD	*SHRNUPD	*SHRRD
*EXCL[1]					
*EXCLRD[2]					X
*SHRUPD[3]			X		X
*SHRNUPD[4]				X	X
*SHRRD[5]		X	X	X	X

[1] Exclusive lock (*EXCL). The object is allocated for the exclusive use of the requesting job; no other job can use the object.

[2] Exclusive lock, allow read (*EXCLRD). The object is allocated to the job that requested it, but other jobs can read the object.

[3] Shared lock, allow read and update (*SHRUPD). The object can be shared either for read or change with other jobs.

[4] Shared lock, read only (*SHRNUPD). The object can be shared for read with other jobs.

[5] Shared lock (*SHRRD). The object can be shared with another job if the job does not request exclusive use of the object.

Table 38 on page 108 shows database locking for constraints of a database file, depending on whether the constraint is associated with the parent file (PAR) or the dependent file (DEP).

Table 38. Database constraint locks

Type of function	File type	File[5]	Member[5]	Other file	Other member
ADDPFM[1]	DEP	*EXCL	*EXCL	*EXCL	*EXCL
ADDPFM[1]	PAR	*EXCL	*EXCL	*EXCL	*EXCL
ADDPFCST[7]	*REFCST	*EXCL	*EXCL	*EXCL	*EXCL
ADDPFCST[6]	*UNQCST *PRIKEY	*EXCL	*EXCL	*EXCL	*EXCL
ADDPFCST	*UNIQUE *PRIKEY	*EXCL	*EXCL		
RMVM[2]	DEP	*EXCL	*EXCL	*EXCL	*EXCL
RMVM[2]	PAR	*EXCL	*EXCL	*EXCL	*EXCL
DLTF[3]	DEP	*EXCL	*EXCL	*EXCL	*EXCL
DLTF[3]	PAR	*EXCL	*EXCL	*EXCL	*EXCL
RMVPFCST[7]	*REFCST	*EXCL	*EXCL	*EXCL[4]	*EXCL
RMVPFCST[6]	*UNQCST *PRIKEY	*EXCL	*EXCL	*EXCL	*EXCL
RMVPFCST	*UNIQUE *PRIKEY	*EXCL	*EXCL		
CHGPFCST		*EXCL	*EXCL	*SHRRD	*EXCL

[1] If adding a physical file member causes a referential constraint to be established.

[2] If removing a physical file member causes an established referential constraint to become defined.

[3] When deleting a dependent or parent file that has constraints established or defined for the file.

[4] When the Remove Physical File Constraint (RMVPFCST) command is invoked for the parent file which has constraints established or defined, the parent and any logical files over the parent file are all locked *EXCL.

[5] For referential constraints, the column refers to the dependent file or the dependent member.

[6] Unique constraint or primary key constraint is a parent key in a referential constraint where the other file is a dependent file.

[7] The other file is a parent file.

Displaying locked rows using System i Navigator:

You can use System i Navigator to display locked rows in a table.

To display locked rows, follow these steps:
1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the table for which you want to display locked rows.
4. Click **Tables**.
5. Right-click the table for which you want to display locked rows, and click **Locked Rows**. The Locked Rows window displays the rows that are locked.

Displaying locked records using the Display Record Locks (DSPRCDLCK) command:

You can use the Display Record Locks (DSPRCDLCK) command to display the current lock status (wait or held) of records in a physical file member.

The DSPRCDLCK command will also indicate what type of lock is currently held. 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.

> **Related concepts**
> Control language

Backup and recovery

Related reference

Display Record Locks (DSPRCDLCK) command

Sharing database files in the same job or activation group

By default, the database management system allows one file to be read and changed by many users at the same time. You can also share a file in the same job or activation group by specifying the SHARE parameter.

You can share a file in the same job or activation group by opening the database file:

* More than once in the same program.
* In different programs in the same job or activation group.

Note: For more information about open sharing in the Integrated Language Environment® model, see the

ILE Concepts manual.

The SHARE parameter on the create file, change file, and override database file commands allows sharing in a job or activation group, including sharing the file, its status, its positions, and its storage area. Sharing files in the job or activation group can improve performance by reducing the amount of main storage needed and by reducing the time needed to open and close the file.

Using the SHARE(*YES) parameter lets two or more programs running in the same job or activation group share an open data path (ODP). 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 you do not specify the SHARE(*YES) parameter, a new open data path is created every time a file is opened. If an active file is opened more than once in the same job or activation group, you can use the active ODP for the file with the current open of the file. You do not have to create a new open data path.

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 or activation group. SHARE(*YES) must be specified for the first open and other opens of the same file for the ODP to be shared. A well-designed (for performance) application normally shares an ODP with files that are opened in multiple programs in the same job or activation group.

Specifying SHARE(*NO) tells the system not to share the ODP 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 if 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 or activation group 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 or an activation group:

Here are the considerations for opening a database file that is shared in the same job or activation group.

* Make sure that when the shared file is opened for the first time in a job or activation group, 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 Open Database File (OPNDBF) or Open Query File (OPNQRYF) command parameters, to remove any incompatible options.)

  For example, PGMA is the first program to open FILE1 in the job or activation group 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 might have to use file operation statements that are never run. See your high-level language topic collection for more details.) You can also specify the file processing option on the OPTION parameter on the OPNDBF and OPNQRYF commands.

- Sometimes sharing a file within a job or activation group is not desirable. For example, one program needs records from a file in arrival sequence and another program needs records in keyed sequence. In this situation, you should not share the open data path (ODP). Specify SHARE(*NO) on the Override with Database File (OVRDBF) command to ensure that the file is not shared within the job or activation group.

- 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 ODP 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 OVRDBF command (*LIBL is used), the system assumes that 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 that 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 parameter of 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 OPNQRYF command. Any parameter values specified on the OVRDBF command other than TOFILE, MBR, LVLCHK, and SEQONLY are ignored by the OPNQRYF command.

- The OPNDBF and OPNQRYF commands scope the ODP to the level specified on the Open Scope (OPNSCOPE) parameter according to the following rules:
  - The system searches for shared opens in the activation group first, and then in the job.
  - Shared opens that are scoped to an activation group might not be shared between activation groups.
  - Shared opens that are scoped to the job can be shared throughout the job, by any number of activation groups at a time.

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 OPNQRYF command never shares an existing shared ODP in the job or activation group. If a shared ODP already exists in the job or activation group with the same file, library, and member name as the one specified on the OPNQRYF command, the system sends an error message and the query file is not opened.

Input/output considerations for files shared in a job or an activation group:

Here are the considerations for processing a database file that is shared in the same job or activation group.

- Because only one open data path is allowed for a shared file, only one record position is maintained for all the programs in the job or activation group that is sharing the file. If a program establishes a

position for a record using a read or a read-for-update operation, and then calls another program that also uses the shared file, the record position might 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 operation, 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 or activation group to open a shared file. PGMA intends to update records in the file; therefore, when the program reads a record for update operation, it locks 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 does 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 or an activation group:

Here are the considerations for closing a database file that is shared in the same job or activation group.

- 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 also be used to close a file opened by the OPNQRYF command when one of the following parameters is specified:
  - OPNSCOPE(*ACTGRPDFN), and the open is requested from the default activation group.
  - TYPE(*NORMAL)

  If one of the following parameters is specified, the file remains open even if the Reclaim Resources (RCLRSC) command is run:
  - OPNSCOPE(*ACTGRPDFN), and the open is requested from an activation group other than the default.
  - OPNSCOPE(*ACTGRP)
  - OPNSCOPE(*JOB)
  - TYPE(*PERM)

Example 1: A single set of files with similar processing options:

You use the same set of files each time you sign on.

A control language (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 the Override with Database File (OVRDBF) command should precede each of the Open Database File (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.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 290.

```
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
```

The files opened in PGMA are either scoped to the job, or PGMA, PGM11, and PGM12 run in the same activation group and the file opens are scoped to that activation group.

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 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) command is 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. For more information about how to reclaim resources in the Integrated Language Resources, see the ILE Concepts book.

Note: The display file (DISPLAY) in PGMB can also be specified as a shared file, which can 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: 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.

Related concepts

ILE Concepts PDF

Example 2: Multiple sets of files with similar processing options:

You use a different set of files for each program. You normally work with one program for a considerable length of time before selecting a new program.

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.

An example of the accounts receivable programs follows:

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 290.

```
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 can be reduced by placing the OPNDBF commands in a separate program and calling it. A separate program can also be created for the CLOF commands. The OPNDBF commands can 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 might 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 CLOF command. 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 does *not* close files that are opened with the following parameters specified on the OPNDBF or Open Query File (OPNQRYF) command:

- OPNSCOPE(*ACTGRPDFN), and the open is requested from some activation group other than the default.
- OPNSCOPE(*ACTGRP) reclaims if the RCLRSC command is from an activation group with an activation group number that is lower than the activation group number of the open.
- OPNSCOPE(*JOB).
- TYPE(*PERM).

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: A single set of files with different processing options:

You use the same set of files for programs that need read-only file processing and for programs that need some or all of the options (input, update, add, and delete).

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 can be used to specify OPTION(*ALL) and the open data path would be opened shared (if, for example, a previous Override with Database File (OVRDBF) command was used to specify SHARE(*YES)). Each program can 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 OPNDBF command and the Start Commitment Control (STRCMTCTL) command specifies LCKLVL(*ALL) or LCKLVL(*CS), 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 can 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 of database files

If your program processes a database file sequentially for input only or output only, you might be able to improve performance by specifying the sequential-only processing (SEQONLY) parameter on the Override with Database File (OVRDBF) or the Open Database File (OPNDBF) command.

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 that contains positioning operations does not automatically request sequential-only processing. (The SETLL operation in the RPG language and the START operation in the COBOL language are examples of file positioning operations.) Even though the high-level language program cannot automatically request sequential-only processing, you can request it using the SEQONLY parameter on the Override with Database File (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.

Note: Sequential-only processing should not be used with a keyed sequence access path file unless the physical data is in the same order as the access path. SEQONLY(*YES) processing might cause poor application performance until the physical data is reorganized into the access path's order.

Open considerations for sequential-only processing:

Here are the considerations 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 (read) 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.
- Sequential-only processing can be specified with commitment control only if the member is opened for output-only.

- If sequential-only processing is being used for files opened with commitment control and a rollback operation is performed for the job, the records that are in the job's storage area during the rollback operation are not written to the system storage area and never appear in the journal for the commitment control transaction. If no records were ever written to the system storage area prior to a rollback operation being performed for a particular commitment control transaction, the entire commitment control transaction is not reflected in the journal.
- 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 number of records in a block will be changed to one if all of the following conditions are true:
  - The member was opened for output-only processing.
  - No override operations are in effect that have specified sequential-only processing.
  - The file being opened is a file that cannot be extended because its increment number of records was set to zero.
  - The number of bytes available in the file is less than the number of bytes that fit into a block of records.

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 SELECT statement, then SEQONLY(*YES) processing is automatically performed.

  Related concepts

  "Using Open Query File (OPNQRYF) command" on page 123
  By using the Open Query File (OPNQRYF) command, you can open a file to a set of database records that satisfies a database query request.

Input/output considerations for sequential-only processing:

Here are the considerations 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.

  Note: Changes made after records are read into the input buffer are not reflected in the input buffer.
- 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. This journal processing occurs before the records are added to the database.

  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 can occur when records are moved from the buffer to the database. For example, assume that 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.
- The number of records in a block will be changed to one if all of the following conditions are true:
  - The member was opened for output-only processing or sequential-only processing.
  - No override operations are in effect that have specified sequential-only processing.
  - The file being opened is being extended because the increment number of records was set to zero.
  - The number of bytes available in the file is less than the number of bytes that fit into a block of records.

Close considerations for sequential-only processing:

Here are the considerations for closing files when sequential-only processing is specified.

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.

Summary of runtime considerations for processing database files

These tables show whether you can specify a particular file processing option on a CL command and in a high-level language.

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, SHARE, WAITRCD, and INHWRT are ignored by the OPNQRYF command.

The parameters in the following tables are arranged in the order of importance. If a new parameter is related to an existing parameter, it is added next to the existing parameter.

Table 39. Database processing options specified on CL commands

Description	Parameter	Command				
		CRTPF, CRTLF	CHGPF, CHGLF	OPNDBF	OPNQRYF	OVRDBF
File name	FILE	Yes	Yes[1]	Yes	Yes	Yes

Table 39. Database processing options specified on CL commands (continued)

Description	Parameter	Command				
		CRTPF, CRTLF	**CHGPF, CHGLF**	**OPNDBF**	**OPNQRYF**	**OVRDBF**
Library name	LIB	Yes	Yes[2]	Yes	Yes	Yes
Member name	MBR	Yes	No	Yes	Yes	Yes
Member processing options	OPTION	No	No	Yes	Yes	No
Record format lock state	RCDFMTLCK	No	No	No	No	Yes
Starting file position after open	POSITION	No	No	No	No	Yes
Program performs only sequential processing	SEQONLY	No	No	Yes	Yes	Yes
Ignore keyed sequence access path	ACCPTH	No	No	Yes	No	No
Time to wait for file locks	WAITFILE	Yes	Yes	No	No	Yes
Time to wait for record locks	WAITRCD	Yes	Yes	No	No	Yes
Prevent overrides	SECURE	No	No	No	No	Yes
Number of records to be transferred from auxiliary to main storage	NBRRCDS	No	No	No	No	Yes
Share open data path with other programs	SHARE	Yes[3]	Yes[4]	No	No	Yes
Format selector	FMTSLR	Yes[5]	Yes[5]	No	No	Yes
Force ratio	FRCRATIO	Yes	Yes	No	No	Yes
Inhibit write	INHWRT	No	No	No	No	Yes
Level check record formats	LVLCHK	Yes	Yes	No	No	Yes
Expiration date checking	EXPCHK	No	No	No	No	Yes
Expiration date	EXPDATE	Yes[6]	Yes[6]	No	No	Yes

Table 39. Database processing options specified on CL commands (continued)

Description	Parameter	Command				
		CRTPF, CRTLF	CHGPF, CHGLF	OPNDBF	OPNQRYF	OVRDBF
Force access path	FRCACCPTH	Yes	Yes	No	No	No
Commitment control	COMMIT	No	No	Yes	Yes	No
End-of-file delay	EOFDLY	No	No	No	No	Yes
Duplicate key check	DUPKEYCHK	No	No	Yes	Yes	No
Reuse deleted record space	REUSEDLT	Yes[6]	Yes[6]	No	No	No
Coded character set identifier	CCSID	Yes[6]	Yes[6]	No	No	No
Sort sequence	SRTSEQ	Yes	Yes[7]	No	Yes	No
Language identifier	LANGID	Yes	Yes[7]	No	Yes	No

[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] Share open data path with other programs: It applies only to members that are added on the create file request.

[4] Share open data path with other programs: It applies only to the current members of the file on the change file request.

[5] Format selector: It is used on the CRTLF and CHGLF commands only.

[6] Expiration date, reuse deleted records, and coded character set identifier: They are used on the CRTPF and CHGPF commands only.

[7] Sort sequence and language identifier: They cannot be specified on the CHGLF command. They can be specified on the CHGPF command only when they are used in conjunction with the source file.

Table 40. Database processing options specified in high-level language programs

Description	RPG/400	ILE RPG	COBOL/400	ILE COBOL	PL/I
File name	Yes	Yes	Yes	Yes	Yes
Library name	No	Yes	No	No	Yes
Member name	No	Yes	No	No	Yes
Record length	Yes	Yes	Yes	Yes	Yes
Member processing options	Yes	Yes	Yes	Yes	Yes
Record format lock state	No	No	No	No	Yes

Table 40. Database processing options specified in high-level language programs (continued)

Description	RPG/400	ILE RPG	COBOL/400	ILE COBOL	PL/I
Record formats the program will use	Yes	Yes	No	No	No
Clear physical file member of records	No	No	Yes	Yes	No
Program performs only sequential processing	Yes	Yes	Yes	Yes	Yes
Ignore keyed sequence access path	Yes	Yes	Yes	Yes	Yes
Share open data path with other programs	No	No	No	No	Yes
Level check record formats	Yes	Yes	Yes	Yes	Yes
Commitment control	Yes	Yes	Yes	Yes	Yes
Duplicate key check	No	No	Yes	Yes	No
Note: Control language (CL) programs can also specify many of these parameters. See Table 39 on page 117 for more information about the database processing options that can be specified on CL commands.					

Storage pool paging option effect on database performance

The paging option of shared pools can have a significant impact on the performance of reading and changing database files.

- A paging option of *FIXED causes the program to minimize the amount of memory it uses by:
  - Transferring data from auxiliary storage to main memory in smaller blocks
  - Writing file changes (updates to existing records or newly added records) to auxiliary storage frequently

  This option allows the system to perform much like it did before the paging option was added.

- A paging option of *CALC might improve how the program performs when it reads and updates database files. In cases where there is sufficient memory available within a shared pool, the program might:
  - Transfer larger blocks of data to memory from auxiliary storage.
  - Write changed data to auxiliary storage less frequently.

  The paging operations done on database files vary dynamically based on file use and memory availability. Frequently referenced files are more likely to remain resident than those less often accessed. The memory is used somewhat like a cache for popular data. The overall number of I/O operations might be reduced using the *CALC paging option.

  Related concepts

  Performance

Opening a database file

You can use the Open Query File (OPNQRYF) and Open Database File (OPNDBF) commands to open database file members in a program.

Opening a database file member

You can open a database file member with statements in your high-level language program. You can also use the control language (CL) open commands: Open Database File (OPNDBF) and Open Query File (OPNQRYF).

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.

If you specify a member name, files that have the correct file name but do not contain the member name are ignored. If you have multiple database files named FILEA in different libraries, the member that is opened is the first one in the library list that matches the request. For example, LIB1, LIB2, and LIB3 are in your library list and all three contain a file named FILEA. Only FILEA in LIB3 has a member named MBRA that is to be opened. Member MRBA in FILEA in LIB3 is opened; the other FILEAs are ignored.

After finding the member, the system connects your program to the database file. This allows your program to perform input/output operations on the file. For more information about opening files in your high-level language program, see the appropriate high-level language topic collection.

You can also open a database file member with the OPNDBF command and the OPNQRYF command. 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 your program. Then, your program can use the information supplied by the OPNQRYF command to process only the data it needs.

> **Related concepts**
> Control language
> **Related reference**
> Open Database File (OPNDF) command
> Open Query File (OPNQRYF) command

Using Open Database File (OPNDBF) command

Usually when you use the Open Database File (OPNDBF) command, you use the default values for the command parameters. Sometimes you might want to specify particular values for some of the parameters.

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 sequence access path, then you can specify ACCPTH(*ARRIVAL) on the OPNDBF parameter. Ignoring the keyed sequence access path can improve your job's performance.

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.

DUPKEYCHK parameter

Specify whether 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 the COBOL or ILE C/C++ language, or if your COBOL or ILE C/C++ program does not use the duplicate-key feedback information that is returned.

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 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 control language (CL) commands to process the file. For example, the Close File (CLOF) command uses the identifier to specify which file is to be closed.

OPNSCOPE parameter

This parameter specifies the scoping of the open data path (ODP). Specify *ACTGRPDFN if the request is from the default activation group, and the ODP is to be scoped to the call level of the program issuing the command. If the request is from any other activation group, the ODP is scoped to that activation group. Specify *ACTGRP if the ODP is to be scoped to the activation group of the program issuing the command. Specify *JOB if the ODP is to be scoped to the job. If you specify this parameter and the TYPE parameter you get an error message.

OPTION parameter

Specify the *INP option if your application programs use 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 use 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.

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. Sequential-only processing should not be used with a keyed sequence access path file unless the physical data is in access path order.

TYPE parameter

Specify what you want to happen when exceptions that are not monitored occur in your application program. If you specify *NORMAL, one of the following operations can happen:

- Your program can issue a Reclaim Resources (RCLRSC) command to close the files opened at a higher level in the call stack than the program issuing the RCLRSC command.
- 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. TYPE(*NORMAL) causes files to be closed if both of the following situations occur:

- Your program receives an error message.
- The files are opened at a higher level in the call stack.

TYPE(*PERM) allows the files to remain open even if an error message is received. Do not specify this parameter if you specified the OPNSCOPE parameter.

Using Open Query File (OPNQRYF) command

By using the Open Query File (OPNQRYF) command, you can open a file to a set of database records that satisfies a database query request.

The OPNQRYF command allows you to perform many data processing functions on database files. Essentially, it acts as a filter between the processing program and the database records. The database file can be a physical or logical file. Unlike the Create Physical File (CRTPF) or Create Logical File (CRTLF) command, the OPNQRYF command creates only a temporary file for processing the data; it does not create a permanent file.

The OPNQRYF command has functions similar to those in data description specifications (DDS) and to the CRTPF and CRTLF commands. DDS requires source statements and a separate step to create the file. The OPNQRYF command 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, IBM Query for i5/OS can be used to perform some of the functions that 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 SELECT statements. For example, the FILE parameter is similar to the SQL FROM statement, the QRYSLT parameter is similar to the SQL WHERE statement, the GRPFLD parameter is similar to the SQL GROUP BY statement, and the GRPSLT parameter is similar to the SQL HAVING statement.

The following list shows the major functions supplied by the OPNQRYF command:
- Dynamic record selection
- Dynamic keyed sequence access path
- Key fields from different physical files
- Dynamic join
- Missing records in secondary join files
- Unique-key processing
- Mapped field definitions
- Group processing
- Final total-only processing
- Performance optimization
- Open query identifier (ID)
- Sort sequence processing

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.

Related concepts

SQL programming

Control language

Related reference

Open Query File (OPNQRYF) command

Creating queries:

You can use the Open Query File (OPNQRYF) command to create a query over database records. Alternatively, you can create a query by running SQL scripts in System i Navigator.

Related concepts

Querying your database by running SQL scripts

Related reference

Open Query File (OPNQRYF) command

Creating an open query file using an existing record format:

The Open Query File (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.

Assume that 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 following chart is an example of using the OPNQRYF command to select and sequence records:

RBAFO538-0
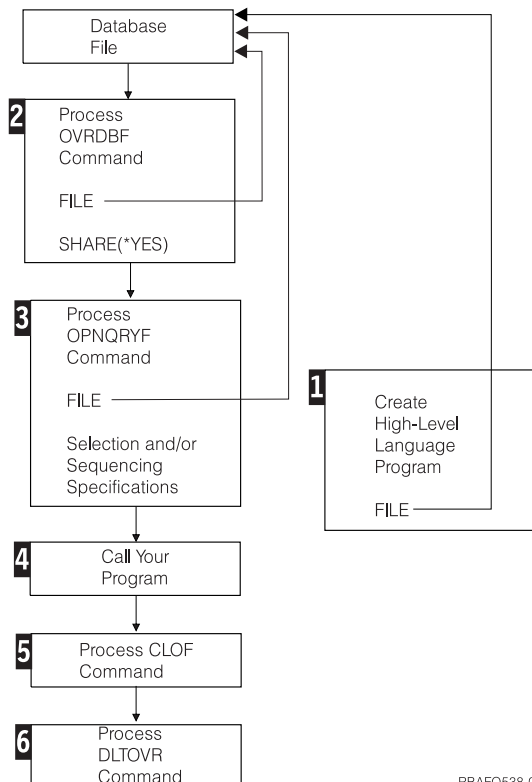
1   Create the high-level language program to process the database file as you would any normal program using externally described data. Only one format can be used, and it must exist in the file.

2   Run the Override with Database file (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 these 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.

3   Run the OPNQRYF command, specifying the database file, member, format names, any selection options, any sequencing options, and the scope of influence for the opened file.

4   Call the high-level language program you created in step 1. Besides using a high-level language, the Copy from Query File (CPYFRMQRYF) command can also be used to process the file created by the OPNQRYF command. Other control language (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 Delete Override (DLTOVR) command. It might not always be necessary to delete the override, but the command is shown in all the examples for consistency.

Related concepts

"Files shared in a job" on page 188
To use the open data path that is built by the Open Query File (OPNQRYF) command, your program must share the query file.

Creating an open query file using 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 separate from the one you are going to process and 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.

The following chart shows the process flow:

Database
File

3 Process
OVRDBF
Command

FILE

TOFILE

SHARE(*YES)

4 Process
OPNQRYF
Command

FILE

FORMAT

Mapped Field
Definitions

1 Create File
with Different
Format

2 Create
High-Level
Language
Program

FILE

5 Call Your
Program

6 Process CLOF
Command

7 Process
DLTOVR
Command
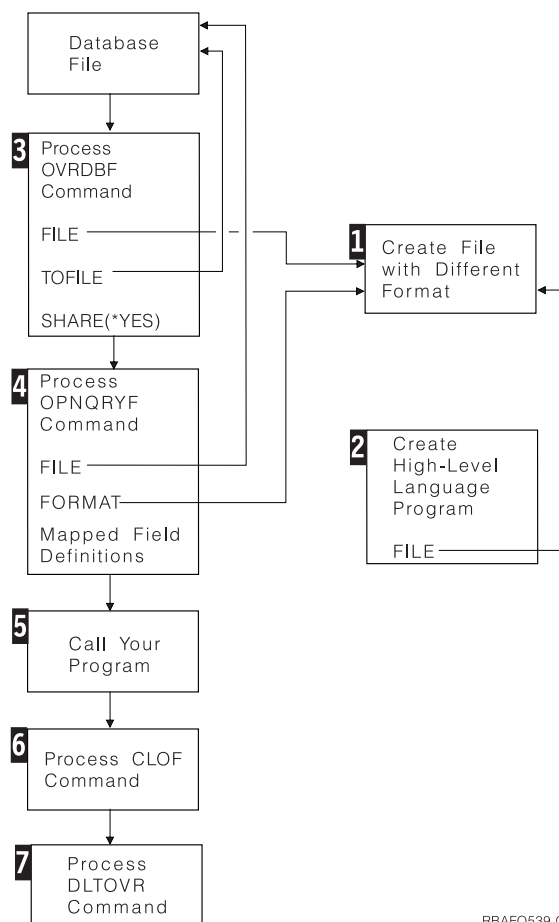
RBAFO539-0

1   Specify the data description specifications (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 can be a logical file and it can be indexed. It can 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 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 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 options, sequencing options, and the scope of influence for the opened file.

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 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, or you can use the Copy from Query File (CPYFRMQRYF) command. The Copy File
(CPYF) command cannot be used. You can then follow step 5 with the CPYF command or Query.

Dynamically selecting records:

The Open Query File (OPNQRYF) command supports dynamic record selection. That is, you can request
a subset of records in a file without using data description specifications (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 examples that follow show how to select records using the OPNQRYF command. In all these
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.)

> **Related concepts**
>
> "Usage notes for the Open Query File (OPNQRYF) command" on page 160
> You need to be aware of these considerations when using the Open Query File (OPNQRYF) command.
>
> **Related reference**
>
> Open Query File (OPNQRYF) command

Example 1: Dynamically selecting records:

This example shows how to select records with a specific value using the Open Query File (OPNQRYF)
command.

Assume that 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 single quotation marks.

Specify the following parameters:
```
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 single quotation marks (' ').
2. When specifying field names in the OPNQRYF command, the names in the record are not enclosed in quotation marks.
3. Character literals must be enclosed by quotation marks (" "). (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')
```

   Note: Numeric constants are *not* enclosed by quotation marks.
5. When comparing a field value to a control language (CL) variable, use apostrophes as follows (only character CL variables can be used):
   - If doing selection against a character, date, time, or timestamp field, specify:

```
OPNQRYF    FILE(FILEA) QRYSLT('"' *CAT &CHAR *CAT '" *EQ FIELDA')
```

     or, in reverse order:

```
OPNQRYF    FILE(FILEA) QRYSLT('FIELDA *EQ "' *CAT &CHAR *CAT '"')
```

     Note: Single quotation marks (' ') and quotation marks (" ") enclose the CL variables and *CAT operators.
   - If doing selection against a numeric field, specify:

```
OPNQRYF    FILE(FILEA) QRYSLT(&CHARNUM *CAT ' *EQ NUM')
```

     or, in reverse order:

```
OPNQRYF    FILE(FILEA) QRYSLT('NUM *EQ ' *CAT &CHARNUM);
```

     Note: Single quotation marks enclose the field and operator only.

When comparing two fields or constants, the data types must be compatible. The following table describes the valid comparisons.

Table 41. Valid data type comparisons for the OPNQRYF command

	Any numeric	Character	Date[1]	Time[1]	Timestamp[1]
Any numeric	Valid	Not valid	Not valid	Not valid	Not valid
Character	Not valid	Valid	Valid[2]	Valid[2]	Valid[2]
Date[1]	Not valid	Valid[2]	Valid	Not valid	Not valid
Time[1]	Not valid	Valid[2]	Not valid	Valid	Not valid
Timestamp[1]	Not valid	Valid[2]	Not valid	Not valid	Valid

[1] Date, time, and timestamp data types can be represented by fields and expressions, but not constants; however, character constants can represent date, time, or timestamp values.

[2] The character field or constant must represent a valid date value if compared to a date data type, a valid time value if compared to a time data type, or a valid timestamp value if compared to a timestamp data type.

The performance of record selection can be greatly enhanced if a file on the system uses the field being selected in a keyed sequence 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 might 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.

Related concepts

"Double-byte character set considerations" on page 285
A double-byte character set (DBCS) is a character set that represents each character with 2 bytes. The database on the i5/OS operating system supports DBCS.

"Open Query File (OPNQRYF) command: Performance considerations" on page 156
Here are the tips and techniques for optimizing the performance of the Open Query File (OPNQRYF) command.

Example 2: Dynamically selecting records:

This example shows how to select records with a specific date value using the Open Query File (OPNQRYF) command.

Assume that you want to process all records in which the *Date* field in the record is the same as the current date. Also assume that the *Date* field is in the same format as the system date. In a control language (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 single quotation marks. The whole expression is enclosed in single quotation marks. The CAT operators and CL variable are enclosed in both single quotation marks and quotation marks.

It is important to know whether the data in the database is defined as character, date, time, timestamp, or numeric. In the preceding example, the *Date* field is assumed to be character.

If the *DATE* field is defined as date data type, the preceding example can be specified as:

```
OVRDBF FILE(FILEA) SHARE(*YES)
OPNQRYF FILE(FILEA) QRYSLT('%CURDATE *EQ DATE')
CALL PGM(PGMB)
CLOF OPENID(FILEA)
DLTOVR FILE(FILEA)
```

Note: The date field does not have to have the same format as the system date.
You can also specify the example as:

```
DCL VAR(&CVTDAT); TYPE(*CHAR) LEN(6)
DCL VAR(&CURDAT); TYPE(*CHAR) LEN(8)
RTVSYSVAL SYSVAL(QDATE) RTNVAR(&CVTDAT);
CVTDAT DATE(&CVTDAT); TOVAR(&CURDAT); TOSEP(/)
OVRDBF FILE(FILEA) SHARE(*YES)
OPNQRYF FILE(FILEA)
       QRYSLT('"' *CAT &CURDAT *CAT '" *EQ DATE')
CALL PGM(PGMB)
CLOF OPNID (FILEA)
DLTOVR FILE(FILEA)
```

This is where *DATE* has a date data type in FILEA, the job default date format is MMDDYY, and the job default date separator is the slash (/).

Note: For any character representation of a date in one of the following formats, MMDDYY, DDMMYY, YYMMDD, or Julian, the job default date format and separator must be the same to be recognized.

If, instead, you were using a constant, the QRYSLT would be specified as follows:
```
QRYSLT('"12/31/87" *EQ DATE')
```

The job default date format must be MMDDYY and the job default separator must be the slash (/).

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 that the variable is in character form. Normally, this means that before the 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 (see the difference from the preceding examples):
```
QRYSLT(&CHARVAR  *CAT ' *GT DATE')
```

If, instead, you were using a constant, the QRYSLT statement would be specified as follows:
```
QRYSLT('123187 *GT DATE')
```

Example 3: Dynamically selecting records:

This example shows how to select records in a range of values using the Open Query File (OPNQRYF) command.

Assume that you have a *Date* field specified in the character format YYMMDD and with the "." separator, and you want to process all records for 1988. You can specify:
```
OVRDBF     FILE(FILEA) SHARE(*YES)
OPNQRYF    FILE(FILEA) QRYSLT('DATE *EQ %RANGE("88.01.01" +
              "88.12.31") ')
CALL       PGM(PGMC)
CLOF       OPNID(FILEA)
DLTOVR     FILE(FILEA)
```

This example also works if the *DATE* field has a date data type, the job default date format is YYMMDD, and the job default date separator is the period (.).

Note: For any character representation of a date in one of the following formats, MMDDYY, DDMMYY, YYMMDD, or Julian, the job default date format and separator must be the same to be recognized.

If the ranges are variables defined as character data types and the DATE field is defined as a character data type, specify the QRYSLT parameter as follows:
```
QRYSLT('DATE *EQ %RANGE("' *CAT &LORNG *CAT '"' *BCAT '"'  +
  *CAT &HIRNG *CAT '")')
```

However, if the DATE field is defined as a numeric data type, specify the QRYSLT parameter as follows:
```
QRYSLT('DATE *EQ %RANGE(' *CAT &LORNG *BCAT &HIRNG *CAT ')')
```

Note: *BCAT can be used if the QRYSLT parameter is in a control language (CL) program, but it is not allowed in an interactive command.

Example 4: Dynamically selecting records:

This example shows how to select records using the contains function of the Open Query File (OPNQRYF) command.

Assume that 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 as follows:

```
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 is in lowercase or mixed case, you can 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 can use any translation table to perform the translation.) Therefore, you can specify as follows:

```
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: Dynamically selecting records:

This example shows how to select records by specifying multiple fields on the Open Query File (OPNQRYF) command.

Assume that you want to process all records in which either the *Amt* field is equal to zero, or the *Lstdat* field (YYMMDD order in character format) is equal to or less than 88-12-31. You can specify:

```
OVRDBF    FILE(FILEA) SHARE(*YES)
OPNQRYF   FILE(FILEA) QRYSLT('AMT *EQ 0 *OR LSTDAT +
                *LE "88-12-31" ')
CALL      PGM(PGMC)
CLOF      OPNID(FILEA)
DLTOVR    FILE(FILEA)
```

This example also works if the *LSTDAT* field has a date data type. The *LSTDAT* field can be in any valid date format; however, the job default date format must be YYMMDD and the job default date separator must be the dash (–).

Note: For any character representation of a date in one of the following formats, MMDDYY, DDMMYY, YYMMDD, or Julian, the job default date format and separator must be the same to be recognized.

If variables are used, the QRYSLT parameter is typed as follows:

```
QRYSLT('AMT *EQ ' *CAT &VARAMT *CAT ' *OR +
   LSTDAT *LE "' *CAT &VARDAT *CAT '"')
```

or, typed in reverse order:

```
QRYSLT('"' *CAT &VARDAT *CAT '" *GT LSTDAT *OR ' +
   *CAT &VARAMT *CAT ' *EQ AMT')
```

Note: The &VARAMT variable must be defined as a character type. If the variable is passed to your control language (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: Dynamically selecting records:

This example shows how to use 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 that you want to prompt the user for some selection values and then display one or more pages of records. At the end of the first request for records, the user might want to specify other selection values and display those records. You can do this by following these steps:

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 control language (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 CLOF command or the Reclaim Resources (RCLRSC) command to close the file, then delete the OVRDBF 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: Dynamically selecting records:

This example shows how to map fields for packed numeric data fields using the Open Query File (OPNQRYF) command.

Assume that 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)
```

In this example, if DATE was a date data type, it can be specified as follows:

```
OPNQRYF FILE(FILEA) +
QRYSLT ('YEAR *EQ 88') +
MAPFLD((YEAR '%YEAR(DATE)'))
```

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 can 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: Dynamically selecting records:

This example shows how to select records using the wildcard function of the Open Query File (OPNQRYF) command.

Assume that 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, quotation marks surround the literal to make its definition the same as the *Char6* field. The wildcard function is not supported for DATE, TIME, or TIMESTAMP data types.

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 this 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 matches 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 this 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 is 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 wildcard functions as shown in this 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 that you have a *Name* field that contains:
- JOHNS
- JOHNS SMITH
- JOHNSON
- JOHNSTON

You only gets the first record if you specify:

```
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*")')
```

Related concepts

"Double-byte character set considerations" on page 285
A double-byte character set (DBCS) is a character set that represents each character with 2 bytes. The database on the i5/OS operating system supports DBCS.

Control language

Example 9: Dynamically selecting records:

This example shows how to specify complex selection statements when you select records using the Open Query File (OPNQRYF) command.

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 Control language (CL) topic. 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 shown in this 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 Control language (CL) topic, instead of the abbreviated form (for example, you can use = instead of *EQ), as shown in this 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 like this:
```
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

Example 10: Dynamically selecting records:

This example shows how to use coded character set identifiers (CCSIDs) when you select records using the Open Query File (OPNQRYF) command.

Each character and DBCS field in all database files is tagged with a CCSID. This CCSID allows you to further define the data stored in the file so that any comparison, join, or display of the fields is performed in a meaningful way. For example, if you compare FIELD1 in FILE1 where FIELD1 has a CCSID of 37 (USA) to FIELD2 in FILE2 where FILED2 has a CCSID of 273 (Austria, Germany), appropriate mapping occurs to make the comparison meaningful.

```
OPNQRYF FILE(FILEA FILEB) FORMAT(RESULTF) +
        JFLD((FILEA/NAME FILEB/CUSTOMER))
```

If field NAME has a CCSID of 37 and field CUSTOMER has a CCSID of 273, the mapping of either NAME or CUSTOMER is performed during processing of the OPNQRYF command so that the join of the two fields provides a meaningful result.

Normally, constants defined in the MAPFLD, QRYSLT, and GRPSLT parameters are tagged with the CCSID defined to the current job. This suggests that when two users with different job CCSIDs run the same OPNQRYF command (or a program containing an OPNQRYF command) and the OPNQRYF command has constants defined in it, the users can get different results because the CCSID tagged to the constants might cause the constants to be treated differently.

You can tag a constant with a specific CCSID by using the MAPFLD parameter. By specifying a MAPFLD whose definition consists only of a constant and then specifying a CCSID for the MAPFLD, the constant becomes tagged with the CCSID specified in the MAPFLD parameter. For example:

```
OPNQRYF FILE(FILEA) FORMAT(RESULTF) QRYSLT('NAME *EQ MAP1') +
        MAPFLD((MAP1 '"Smith"' *CHAR 5 *N 37))
```

The constant "Smith" is tagged with the CCSID 37 regardless of the job CCSID of the user issuing the OPNQRYF command. In this example, all users get the same result records (although the result records would be mapped to the user's job CCSID). Conversely, if the query is specified as:

```
OPNQRYF FILE(FILEA) FORMAT(RESULTF) QRYSLT('NAME *EQ "Smith"')
```

The results of the query might differ, depending on the job CCSID of the user issuing the OPNQRYF command.

> **Related concepts**
>
> i5/OS globalization

Example 11: Dynamically selecting records:

This example shows how to use a sort sequence and a language identifier when you select records using the Open Query File (OPNQRYF) command.

To see how to use a sort sequence, run the examples in this topic against the STAFF file shown in Table 42.

Table 42. The STAFF file

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
20	Pernal	20	Sales	8	18171.25	612.45
30	Merenghi	38	MGR	5	17506.75	0
40	OBrien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	0
60	Quigley	38	SALES	00	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	0	13504.60	128.20

Table 42. The STAFF file (continued)

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
90	Koonitz	42	sales	6	18001.75	1386.70
100	Plotz	42	mgr	6	18352.80	0

In the examples, the results are shown for a particular statement using each of the following sort sequences:

- *HEX sort sequence.
- Shared-weight sort sequence for language identifier ENU.
- Unique-weight sort sequence for language identifier ENU.

Note: ENU is chosen as a language identifier by specifying either SRTSEQ(*LANGIDUNQ) or SRTSEQ(*LANGIDSHR), and LANGID(ENU) in the OPNQRYF command.

The following command selects records with the value MGR in the JOB field:

```
OPNQRYF FILE(STAFF) QRYSLT('JOB *EQ "MGR"')
```

Table 43 shows the record selection with the *HEX sort sequence. The records that match the record selection criteria for the JOB field are selected exactly as specified in the QRYSLT statement; only the uppercase MGR is selected.

*Table 43. Using the *HEX sort sequence. OPNQRYF FILE(STAFF) QRYSLT('JOB *EQ "MGR"') SRTSEQ(*HEX)*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

Table 44 shows the record selection with the shared-weight sort sequence. The records that match the record selection criteria for the JOB field are selected by treating uppercase and lowercase letters the same. With this sort sequence, mgr, Mgr, and MGR values are selected.

*Table 44. Using the shared-weight sort sequence. OPNQRYF FILE(STAFF) QRYSLT('JOB *EQ "MGR"') SRTSEQ(LANGIDSHR) LANGID(ENU)*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0
100	Plotz	42	mgr	6	18352.80	0

Table 45 shows the record selection with the unique-weight sort sequence. The records that match the record selection criteria for the JOB field are selected by treating uppercase and lowercase letters as unique. With this sort sequence, the mgr, Mgr, and MGR values are all different. The MGR value is selected.

*Table 45. Using the unique-weight sort sequence. OPNQRYF FILE(STAFF) QRYSLT('JOB *EQ "MGR"') SRTSEQ(LANGIDUNQ) LANGID(ENU)*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

Arranging records:

By using the Open Query File (OPNQRYF) command, you can arrange query records by the value of one or more key fields in various ways and without using data description specifications (DDS).

Specifying dynamic keyed sequence access paths:

The Open Query File (OPNQRYF) command supports dynamic keyed sequence access paths. That is, you can specify a keyed sequence access path without using data description specifications (DDS).

If an access path that can be shared already exists, the system can share it. If a new access path is required, it is built before any records are passed to the program.

Related concepts

"Specifying key fields from different physical files" on page 139
The Open Query File (OPNQRYF) command allows you to specify a processing sequence for a join logical file in which the keys can be in different physical files (DDS restricts the keys to the primary file).

Example 1: Specifying dynamic keyed sequence access paths:

This example shows how to arrange records using one key field.

Assume that 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: Specifying dynamic keyed sequence access paths:

This example shows how to arrange 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.)

Example 3: Specifying dynamic keyed sequence access paths:

This example shows how to arrange records using a unique-weight sort sequence.

To process the records by the JOB field values with a unique-weight sort sequence using the STAFF file in "Example 11: Dynamically selecting records" on page 136, specify:

```
OPNQRYF FILE(STAFF) KEYFLD(JOB) SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

This query results in a JOB field in the following sequence:
- Clerk

- mgr
- Mgr
- Mgr
- MGR
- sales
- Sales
- Sales
- Sales
- SALES

Example 4: Specifying dynamic keyed sequence access paths:

This example shows how to arrange records using a shared-weight sort sequence.

To process the records by the JOB field values with a unique-weight sort sequence using the STAFF file in "Example 11: Dynamically selecting records" on page 136, specify:

```
OPNQRYF FILE(STAFF) KEYFLD(JOB) SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

The results from this query are similar to the results in Example 3. The *mgr* and *sales* entries can be in any sequence because the uppercase and lowercase letters are treated as equals. That is, the shared-weight sort sequence treats mgr, Mgr, and MGR as equal values. Likewise, sales, Sales, and SALES are treated as equal values.

Specifying key fields from different physical files:

The Open Query File (OPNQRYF) command allows you to specify a processing sequence for a join logical file in which the keys can be in different physical files (DDS restricts the keys to the primary file).

The specification is identical to specifying dynamic keyed sequence access paths. The access path is specified by 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 sequence 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: Specifying key fields from different physical files

This example shows how to use a field in a secondary file as a key field.

Assume that 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 that 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)
```

> **Related concepts**
>
> "Specifying dynamic keyed sequence access paths" on page 138
> The Open Query File (OPNQRYF) command supports dynamic keyed sequence access paths. That is, you can specify a keyed sequence access path without using data description specifications (DDS).

Unique-key processing:

The Open Query File (OPNQRYF) command supports unique-key processing. It allows you to process only the first record of a group.

A 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 have unique keys.

When you use unique-key processing, you can only read the file sequentially. The key fields are sorted according to the specified sort sequence (SRTSEQ) and language identifier (LANGID).

If you specify unique-key processing, and the file actually has duplicate keys, you receive only a single record for each group of records with the same key value.

> **Related reference**
>
> "Example 3: Specifying dynamic keyed sequence access paths" on page 138
> This example shows how to arrange records using a unique-weight sort sequence.
>
> "Example 4: Specifying dynamic keyed sequence access paths" on page 139
> This example shows how to arrange records using a shared-weight sort sequence.

Example 1: Unique-key processing:

This example shows how to read only unique-key records.

Assume that 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: Unique-key processing:

This example shows how to read records using only some of the key fields.

Assume that you want to process the same file with the sequence *Slsman, Cust, Date*, but you want only one record per *Slsman* and *Cust* field. Assume that the file contains the following records.

Slsman	Cust	Date	Record #
01	5000	880109	1
01	5000	880115	2

Slsman	Cust	Date	Record #
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

Note: Null values are treated as equal, so only the first null value would be returned.

Random processing:

In addition to sequential processing, you can use the OPNQRYF command for random processing (for example, the RPG language operation CHAIN or the COBOL language operation READ). However, if you use the grouping or unique-key function, you cannot process the file randomly.

Considerations for arranging records:

Here are the considerations for arranging records using the Open Query File (OPNQRYF) command.

The default processing for the 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 OPNQRYF command.

When you specify the KEYFLD(*FILE) parameter option for the OPNQRYF command, and a sort sequence other than *HEX has been specified for the query with the job default or the OPNQRYF SRTSEQ parameter, you can receive your records in an order that does not reflect the true file order. If the file is keyed, the query's sort sequence is applied to the key fields of the file and informational message CPI431F is sent. The file's sort sequence and alternative collating sequence table are ignored for the ordering, if they exist. This allows users to indicate which fields to apply a sort sequence to without having to list all the field names. If a sort sequence is not specified for the query (for example, *HEX), ordering is done as it was prior to Version 2 Release 3.

Formatting records:

The Open Query File (OPNQRYF) command allows you to specify a record format for the query records. You can also define fields that are mapped from existing fields through numeric and character operations.

Defining fields mapped from existing fields:

The Open Query File (OPNQRYF) command supports mapped field definitions. That is, you can define fields that are mapped or derived from other fields.

Mapped field definitions:
- Allow you to create internal fields that specify selection values, as shown in "Example 7: Dynamically selecting records" on page 132.
- Allow you to avoid confusion when the same field name occurs in multiple files, as shown in "Example 1: Dynamically joining database files" on page 150.
- 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: Defining fields mapped from existing fields:

This example shows the use of mapped fields.

Assume that 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 mapped *Exten* field. You want FILEA to be processed, and you have already created FILEAA. 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 want. The *Brfdsc* field is a brief description of the *Descrp* field (it uses the first 10 characters).

Assume that 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: Defining fields mapped from existing fields:

This example shows the use of built-in functions.

Assume that you want to calculate a mathematical function that is the sine of the *Fldm* field in FILEA. First, you can create a file (assume that it is called FILEAA) with a record format that contains 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 Open Query File (OPNQRYF) command converts its internal definition of the sine value (in binary floating point) to the definition of the *Sinm* field. This technique can be used to avoid certain high-level language restrictions on the use of binary floating-point fields. For example, if you defined the *Sinm* field as a packed decimal field, PGMF can be written in any high-level language, even though the value was built using a binary floating-point field.

There are many other functions besides sine that can be used.

> **Related reference**
>
> "Built-in functions" on page 165
> These built-in functions are supported for an expression that is used to define a derived field on the MAPFLD parameter or for a complex selection operand specified on the QRYSLT or GRPSLT parameter.
>
> "Restricted built-in functions" on page 179
> Some built-in functions are restricted in the way certain relational operators are specified on the QRYSLT and GRPSLT parameters.

Example 3: Defining fields mapped from existing fields:

This example shows the use of mapped fields and built-in functions.

Assume, in "Example 2: Defining fields mapped from existing fields," 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 content of the *Fldx* field. You can use the MAPFLD parameter to change the content 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 sort sequence and language identifier can affect the results of the %MIN and %MAX built-in functions. For example, the uppercase and lowercase versions of letters can be equal or unequal depending on the selected sort sequence and language identifier.

Note: The translated field value is used to determine the minimum and maximum, but the untranslated value is returned in the result record.

The example described uses FILEA as an input file. You can also update data using the Open Query File (OPNQRYF) command. However, if you use a mapped field definition to change a field, updates to the field are ignored.

> **Related reference**
>
> "Built-in functions" on page 165
> These built-in functions are supported for an expression that is used to define a derived field on the MAPFLD parameter or for a complex selection operand specified on the QRYSLT or GRPSLT parameter.
>
> "Restricted built-in functions" on page 179
> Some built-in functions are restricted in the way certain relational operators are specified on the QRYSLT and GRPSLT parameters.

Considerations for specifying record formats:

Here are the considerations for using the FORMAT parameter on the Open Query File (OPNQRYF) command.

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 guidelines and rules:

- 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).
- 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 that 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.

  Related reference

  "Example 1: Dynamically joining database files" on page 150
  This example shows how to dynamically join database files without DDS.

Grouping records:

By using the Open Query File (OPNQRYF) command, you can group records by like values of one or more fields and calculate aggregate functions, such as the minimum field value and average field value, for each group.

Summarizing data from database file records (grouping):

The group processing function of the Open Query File (OPNQRYF) command allows you to summarize data from existing database records.

You can specify:
- The grouping fields
- Selection values both before and after grouping
- A keyed sequence 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
- The sort sequence and language identifier that supply the weights by which the field values are grouped

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: Summarizing data from database file records (grouping):

This example shows how to use the group processing function to summarize data from existing database records.

Assume that 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 might need to define the *Amtsum* field as 7 digits. Any arithmetic overflow causes your program to end abnormally.

Assume that 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)
```

Your program retrieves the following records.

Cust	Count	Amtsum	Amtavg	Amtmax
001	3	1500.00	500.00	700.00
002	1	1200.00	1200.00	1200.00

Cust	Count	Amtsum	Amtavg	Amtmax
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 might not appear in ascending sequence. To ensure a specific sequence, you should specify the KEYFLD parameter.

Assume that 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')
```

Your program retrieves the following records.

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 that 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: Fields used for selection do not have to appear in the format processed by the program.

Your program retrieves the following records.

Cust	Count	Amtsum	Amtavg	Amtmax
001	2	800.00	400.00	500.00
002	1	1200.00	1200.00	1200.00

Note: The values for CUST 001 changed because the selection took place before the grouping took place.

Assume that 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))
```

Your program retrieves the following records.

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:

The Open Query File (OPNQRYF) command supports final-total-only processing. It is a special form of grouping in which you do not specify grouping fields. The output is only one record.

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: Final total-only processing:

This example shows simple total processing.

Assume that 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 can 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: Final total-only processing:

This example shows total-only processing with record selection.

Assume that 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: Final total-only processing:

This example shows total-only processing using a new record format.

Assume that you want to process the new file/format with a control language (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       &TOTAMTA *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 immediate message.

Grouping field references:

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 be either grouping fields (specified on the GRPFLD parameter) or mapped fields (specified on the MAPFLD parameter).

Mapped fields are defined by 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

Joining records:

The Open Query File (OPNQRYF) command supports a dynamic join operation without data description specifications (DDS). You can join logical files as well as physical files, and you can specify keys from any of the join files. The command also allows missing records in secondary join files.

Dynamically joining database files:

The dynamic join function of the Open Query File (OPNQRYF) command allows you to join files without having to first specify data description specifications (DDS) and then create a join logical file.

You must use the FORMAT parameter on the 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 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) parameter is used in all of the following examples. The default for the JORDER parameter 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.

The join criterion, like the record selection criterion, is affected by the sort sequence (SRTSEQ) and the language identifier (LANGID) specified.

Related reference

This example shows how to use a sort sequence and a language identifier when you select records using the Open Query File (OPNQRYF) command.

Example 1: Dynamically joining database files:

This example shows how to dynamically join database files without DDS.

Assume that you want to join FILEA and FILEB and 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 on 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 file contains the record format to be specified on the FORMAT parameter of the 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 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 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 can be achieved by specifying:

```
JFLD((CUST FILEB/CUST)) +
MAPFLD((CUST 'FILEA/CUST'))
```

Any other uses of the same field name in the 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 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 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 generally tries to use the file with the smallest number of records selected as the primary file. However, the system also tries to avoid building a temporary file.

You can force the system to follow the file sequence of the join as you have specified it in the FILE parameter on the 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 can cause different results.

Example 2: Dynamically joining database files:

This example shows how to read only those records that have a match in secondary files.

Assume that 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 that 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 sequence access path does not have to exist for these files. However, if a keyed sequence access path does exist, performance might 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 preceding order. 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 parameter 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: Dynamically joining database files:

This example shows how to use 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 that FILEA has character fields *Year, Month*, and *Day* and needs to be joined to FILEB which has the *Date* field in YYMMDD format. Assume that you have defined file JOINAB with the required 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 use 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 when you use the Open Query File (OPNQRYF) command.

Example: Handling missing records in secondary join file

This example shows how to read records from the primary file that do not have a match in the secondary file.

In "Example 1: Dynamically joining database files" on page 150, 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 this 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 can 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 FILE parameter of the OPNQRYF command. The Close File (CLOF) command also names FILEB. The JFLD and MAPFLD parameters are also changed. The program is created using FILEB.

Optimizing performance:

By using the Open Query File (OPNQRYF) command, you can specify how the system performs the selection and processing to improve performance.

Controlling how the system runs the Open Query File (OPNQRYF) command:

The performance optimization function of the Open Query File (OPNQRYF) command 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 the OPNQRYF command 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 command to process the data.

For most batch type functions, you are usually only interested in the total time of both steps mentioned in the preceding paragraph. Therefore, the default for the OPNQRYF command is OPTIMIZE(*ALLIO). This means that the OPNQRYF command considers the total time it takes for both steps.

If you use the OPNQRYF command in an interactive environment, you might not be interested in processing the entire file. You might want the first screen full of records to be displayed as quickly as possible. For this reason, you want the first step to avoid building an access path, if possible. You can specify OPTIMIZE(*FIRSTIO) in such a situation.

If you want to process the same results of the OPNQRYF command with multiple programs, you might want the first step to make an efficient open data path (ODP). That is, you 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 parameter on the OPNQRYF command requires 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.

Related concepts

Database performance and query optimization

Example 1: Controlling how the system runs the Open Query File (OPNQRYF) command:

This example shows how to optimize 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: Controlling how the system runs the Open Query File (OPNQRYF) command:

This example shows how to minimize the number of records that multiple programs 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 can 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)
```

Runtime messages for the Open Query File (OPNQRYF) command:

When the Open Query File (OPNQRYF) command is run, messages are sent to inform the interactive user of the status of the OPNQRYF request.

For example, if a keyed access path is built by the OPNQRYF command to satisfy the request, a message is sent to the user. The following messages might be sent when the OPNQRYF command is run.

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...`
CPI4307	`Query running.  Building hash table from file &1 in &2.`
CPI4011	`Query running.  Number of records processed...`

To stop these status messages from appearing, you can see the discussion about message handling in the Control language (CL) topic.

When your job is running under debug (by using the Start Debug (STRDBG) command), or requested with query options file option of DEBUG_MESSAGES *YES, messages are sent to your job log. These messages describe the implementation method that is used to process the OPNQRYF request. These messages provide information about the optimization processing that occurred. You can use these messages as a tool for tuning the OPNQRYF request to achieve the best performance. Listed here are these messages:

CPI4321

> Access path built for file...

CPI4322

> Access path built from keyed file...

CPI4324

> Temporary file built from file...

CPI4325

> Temporary file built for query

CPI4326

> File processed in join position...

CPI4327

> File processed in join position 1.

CPI4328

> Access path of file that is used...

CPI4329

> Arrival sequence that is used for file...

CPI432A

> Query optimizer timed out...

CPI432C

> All access paths considered for file...

CPI432E

> Selection fields mapped to different attributes...

CPI432F

> Access path suggestion for file...

CPI433B

> Unable to update query options file.

CPI4330

> &6 tasks used for parallel &10 scan of file &1.

CPI4332

> &6 tasks used for parallel index that is created over file...

CPI4333

> Hashing algorithm used to process join.

CPI4338

> &1 access paths used for bitmap processing of file...

CPI4339

> Query options retrieved file &2 in library &1.

CPI4341

> Performing distributed query.

CPI4342

> Performing distributed join for query.

CPI4345

Temporary distributed result file &4 built...

CPI4346

Optimizer debug messages for query join step &1 of &2 follow:

CPI4347

Query is processing in multiple steps.

Most of the messages provide a reason why the particular option was performed. The second level text on each message gives an extended description of why the option was chosen. Some messages provide suggestions to help improve the performance of the OPNQRYF request.

> **Related concepts**
>
> Control language

Open Query File (OPNQRYF) command: Performance considerations:

Here are the tips and techniques for optimizing the performance of the Open Query File (OPNQRYF) command.

The best performance can occur when the OPNQRYF command uses an existing keyed sequence 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 (key positioning selection) rather than read the records and select at run time (dynamic selection).

The OPNQRYF command cannot use an existing index when any of the following conditions are true:
- The key field in the access path is derived from a substring function.
- The key field in the access path is derived from a concatenation function.
- Both listed here are true of the sort sequence table associated with the query (specified on the SRTSEQ parameter):
  - It is a shared-weight sequence table.
  - It does not match the sequence table associated with the access path (a sort sequence table or an alternate collating sequence table).
- Both listed here are true of the sort sequence table associated with the query (specified on the SRTSEQ parameter):
  - It is a unique-weight sequence table.
  - It does not match the sequence table associated with the access path (a sort sequence table or an alternate collating sequence table) when either:
    - Ordering is specified (KEYFLD parameter).
    - Record selection exists (QRYSLT parameter) that does not use *EQ, *NE, *CT, %WLDCRD, or %VALUES.
    - Join selection exists (JFLD parameter) that does not use *EQ or *NE operators.

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 sequence access path. Your program still sees the same records. The OPNQRYF processing can only make this type of decision if an access path exists on the *Code* field. In general, if your request includes approximately 20% or more of the number of records in the file, the OPNQRYF processing tends to ignore the existing access paths and read the file in arrival sequence.

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 sequence 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 sequence 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 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 and is greater than 20% of the records in the file. While it is generally faster to build the keyed sequence 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. You can use the ALWCPYDTA(*OPTIMIZE) parameter of the Open Query File (OPNQRYF) command to allow the system to use a full record sort if this is the fastest method of processing records.

If you do not intend to read all of the query records and if the OPTIMIZE parameter is *FIRSTIO or *MINWAIT, you can specify a number to indicate how many records you intend to retrieve. If the number of records is considerably less than the total number the query is expected to return, the system might select a faster access method.

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 of the OPNQRYF command, the system must create a temporary file. The rules for when a temporary file is created are complex, but the following cases are typical in which this occurs:
- When you specify a dynamic join, and the KEYFLD parameter describes key fields from different physical files.
- When you specify a dynamic join and the GRPFLD parameter describes fields from different physical files.
- When you specify both the GRPFLD and KEYFLD parameters but they are not the same.
- When the fields specified on the KEYFLD parameter total more than 2000 bytes in length.
- When you specify a dynamic join and *MINWAIT for the OPTIMIZE parameter.
- When you specify a dynamic join using a join logical file and the join type (JDFTVAL) of the join logical file does not match the join type of the dynamic join.
- When you specify a logical file and the format for the logical file refers to more than one physical file.
- When you specify an SQL view, the system might require a temporary file to contain the results of the view.
- When the ALWCPYDTA(*OPTIMIZE) parameter is specified and using a temporary result would improve the performance of the query.

When a dynamic join occurs (JDFTVAL(*NO)), the OPNQRYF command attempts to improve performance by reordering the files and joining the file with the smallest number of selected records to the file with the largest number of selected records. To prevent the OPNQRYF command from reordering the files, specify JORDER(*FILE). This forces the OPNQRYF command to join the files in the order specified on the FILE parameter.

> **Related concepts**
>
> Database performance and query optimization

Open Query File (OPNQRYF) command: Performance considerations for sort sequence tables:

Here are the tips and techniques for optimizing the performance of sort sequence tables.

Grouping, joining, and selection: Open Query File (OPNQRYF) command performance:

When using an existing index, the optimizer ensures that the attributes of the selection, join, and grouping fields match the attributes of the keys in the existing index.

Also, the sort sequence table associated with the query must match the sequence table (a sort sequence table or an alternate collating sequence table) associated with the key field of the existing index. If the sequence tables do not match, the existing index cannot be used.

However, if the sort sequence table associated with the query is a unique-weight sequence table (including *HEX), some additional optimization is possible. The optimizer acts as though no sort sequence table is specified for any grouping fields or any selection or join predicates that use the following operators or functions:

- *EQ
- *NE
- *CT
- %WLDCRD
- %VALUES

The advantage is that the optimizer is free to use any existing access path where the keys match the field and the access path either:

- Does not contain a sequence table.
- Contains a unique-weight sequence table (the table does not have to match the unique-weight sort sequence table associated with the query).

Ordering: Open Query File (OPNQRYF) command performance:

For ordering fields, the optimizer is not free to use any existing access path. The sort sequence tables associated with the index and the query must match unless the optimizer chooses to do a sort to satisfy the ordering request.

When a sort is used, the translation is performed during the sort, leaving the optimizer free to use any existing access path that meets the selection criteria.

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 is 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.

Handling errors:

This topic provides solutions to the common problems that you might encounter when using the Open Query File (OPNQRYF) command.

Typical errors when using the Open Query File (OPNQRYF) command:

You must specify several functions correctly for the Open Query File (OPNQRYF) command and your program to get 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 can look at both options if you are having problems.

Listed here are the most common problems you might encounter when using the OPNQRYF command and the ways 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. You have run the OPNQRYF command with the OPNSCOPE(*ACTGRPDFN) or TYPE(*NORMAL) parameter option from a program that was running in the default activation group at a higher level in the call stack than the program that is getting an error message or that is running the Reclaim Resources (RCLRSC) command. This closes the open query file because it was opened from a program at a higher level in the call stack than the program that ran the RCLRSC command. If the open query file was closed, you must run the OPNQRYF command again. Note that when using the OPNQRYF command with the TYPE(*NORMAL) parameter option on releases prior to Version 2 Release 3, the open query file is closed even if it was opened from the same program that reclaims the resources.
- 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 reasons:
  - 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 OVRDBF command, used with the TOFILE parameter, describes the first file on the FILE parameter of the 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 statement.

Handling divide-by-zero errors:

Dividing by zero is considered an error by the Open Query File (OPNQRYF) command. However, you can get a zero result and avoid a divide-by-zero error.

Record selection is normally done before field mapping errors occur. Therefore, a record can be omitted that would have caused a divide-by-zero error and in this case, processing by the OPNQRYF command would continue. If you want a zero answer, here is a solution that is practical for typical commercial data.

Assume that 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) .nnnn1)
```

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 can 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 ensures 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.

Usage notes for the Open Query File (OPNQRYF) command:

You need to be aware of these considerations when using the Open Query File (OPNQRYF) command.

Command elements:

Here are the rules for field names and expressions specified on the Open Query File (OPNQRYF) command parameters, as well as a wide range of built-in functions that the OPNQRYF command supports.

Field names:

Field names specified on the Open Query File (OPNQRYF) command parameters must follow these rules.

The field name used as the first part of an element in the list specified on the MAPFLD parameter must be a simple name, and the field names in the record format identified on the FORMAT parameter are always treated as simple names. Any other field name specified on an OPNQRYF command parameter (QRYSLT, KEYFLD, JFLD, GRPFLD, GRPSLT, or the field-definition expression part of the MAPFLD parameter) is a qualified field name, specified as follows:

field-name
> Specify a simple field name that identifies a field that is defined on the MAPFLD parameter, or with a field name that is unique among all field names from all record formats included in the list specified on the FILE parameter. This form is not allowed if there is no MAPFLD parameter definition for the specified field name and the FILE parameter includes more than one record format that contains a field with the specified name, even if the same file and record format is specified more than once in the list on the FILE parameter.
>
> For example, AMOUNT is valid if the field named AMOUNT is defined on the MAPFLD parameter. It is also valid if AMOUNT is not defined on the MAPFLD parameter, as long as there is only one field named AMOUNT in any record format specified on the FILE parameter.

file-name/field-name
> Specify a field name that is qualified with the simple name of the file specified on the FILE parameter whose record format contains the field, but only if the simple file name is unique among all file names specified on the FILE parameter. This form is not allowed if the same simple file name is specified more than once in the list specified for the FILE parameter, even if different library, member, or record format names are used.
>
> For example, WHS01/PARTNBR is valid if there is a field named PARTNBR in the record format for file WHS01, and file name WHS01 is only specified once on the FILE parameter.

file-nbr/field-name
> Specify a simple field name that is qualified with the number of the element in the FILE parameter list for the record format that contains the field. The file-nbr qualifier must be specified without leading zeros. This form is only required if the same simple file name is specified more than once in the list specified on the FILE parameter.
>
> For example, 2/BALDUE is valid if the second file record format in the list specified on the FILE parameter contains a field named BALDUE.

*MAPFLD/field-name
> Specify a simple field name that is qualified with the special value *MAPFLD if the field is defined on the MAPFLD parameter. When the field is defined, this form has the same meaning as specifying the simple field name with no qualifier. If the field is not defined on the MAPFLD parameter, *MAPFLD cannot be specified.
>
> For example, *MAPFLD/AVGBAL is valid if the AVGBAL field is specified as the first part of one of the mapped field list elements specified on the MAPFLD parameter.

Expressions:

The expressions specified on the Open Query File (OPNQRYF) command parameters must follow these conventions.

Expressions specified on the QRYSLT, GRPSLT, and MAPFLD parameters are similar to expressions specified on other control language (CL) command parameters. Logical, relational, numeric, and string operations are performed by using combinations of field values and constants. Symbolic and named operators are supported, as well as many built-in functions, and parentheses are used to control the order of evaluation.

There are also differences in the expressions specified on OPNQRYF parameters and on other CL command parameters. Listed here are the ways that expressions on the QRYSLT, GRPSLT, and MAPFLD parameters differ from normal CL expressions:

- The expression string must be enclosed in apostrophes if it contains embedded blanks or special characters.
- The following differences affect numeric and string literals:
  - Character string constants are quoted by using single quotation marks or quotation marks.
  - The leading and trailing zeros of a numeric constant are significant parts of its attributes.
  - Floating-point constants (including the special values *INF and *NEGINF) are used in expressions.
- The following differences contrast CL variables with database fields:
  - No prefixed ampersand (&) is used in database field names.
  - Qualified field names are supported.
  - No 'logical' field type exists for database fields.
  - Many additional data types are supported for database fields.
- The following CL operators are not supported on the OPNQRYF command:
  - *BCAT or | >
  - *TCAT or | <
- The following additional operators are supported beyond CL support:
  - // for remainder
  - ** for exponentiation
  - *CT for 'contains' (character scan)
  - *XOR or && for 'logical exclusive or'
- The following differences affect built-in function support:
  - The %SWITCH built-in function is not supported.
  - Many additional built-in functions are supported.
  - Nested built-in functions and expressions for built-in function arguments (such as '%LOG(%SIN(x))') generally are allowed.
  - To support expressions as built-in function arguments, any argument that is a signed numeric value or an expression (for example, '%MIN(3 (-2) x (y+4))') must be enclosed in parentheses.

The following table shows the priority of all operators that are used for expressions on the QRYSLT, GRPSLT, or MAPFLD parameters. Only operators listed for priorities 1 through 5, excluding the *NOT and operators, are allowed in an expression specified on the MAPFLD parameter:

Priority	Operators
1	+, - (when used for signed numeric values), *NOT, ¬
2	**
3	*, / ,// (a / must have a space before the / and/or after the /)
4	+, - (when used between two operands)
5	*CAT, | |
6	*GT, *LT, *EQ, *GE, *LE, *NE, *NG, *NL, *CT, >, <, =, >=, <=, ¬=, ¬>, ¬<
7	*AND, &
8	*OR, *XOR, |, &&

Except for operators ¬ and *NOT, the operators for priorities 1 through 4 are numeric operators, which require numeric operands. The operators for priority 5 are string operators, which require operands to be either character or DBCS strings. Priority 6 operators are called relational operators, which require at least

one operand that is a field name or a numeric or string expression (not a constant). The operators for priorities 7 and 8, plus the ¬ and *NOT operators (priority 1), are logical operators. The operands in a logical expression are relations (constructed by using a relational operator with appropriate operands) and other logical expressions.

The operands in a string expression, including string operands for a built-in function, are a combination of character fields and DBCS fields and constants. If both operands of such an expression are DBCS-only fields or constants, the final result from evaluation of the expression is a DBCS-only field value. If the operands are a combination of DBCS or character fields or constants, the result is a DBCS-open field value. When DBCS fields are concatenated, the extraneous shift-in and shift-out characters between the fields are removed.

The result produced by a + or - sign prefixed operator has the same attributes as the operand, unless the operand of a - sign prefixed operator is a *BIN2, in which case the result is a *BIN4. The result of an ** operator (exponentiation) is a double-precision binary floating-point number (*FLT8), or a decimal floating-point number (34 digits) if either operand is a decimal floating-point number.

For other numeric operators that require two operands, if either operand is a binary floating-point number, the result is a double-precision binary floating-point number (*FLT8); if either operand is a decimal floating-point number, the result is a decimal floating-point number. See the following table for rules that involve decimal floating-point operands.

One operand	The other operand	Result
Decimal floating-point(n)[1]	*BIN2	Decimal floating-point(n)[1]
Decimal floating-point(n)[1]	*BIN4	Decimal floating-point(n)[1]
Decimal floating-point(n)[1]	*BIN8	Decimal floating-point(34)
Decimal floating-point(n)[1]	Decimal($d<=16, f$)[2]	Decimal floating-point(n)[1]
Decimal floating-point(n)[1]	Decimal($d>16, f$)[2]	Decimal floating-point(34)
Decimal floating-point(n)[1]	*FLT4	Decimal floating-point(n)[1]
Decimal floating-point(n)[1]	*FLT8	Decimal floating-point(n)[1]
Decimal floating-point(n)[1]	Decimal floating-point(m)[1]	Decimal floating-point($\max(n, m)$)[1]

[1] n or m is the number of significant digits of the decimal floating-point operand.

[2] d is the total number of digits and f is the number of fractional digits of the decimal operand.

If both operands are fixed-point numbers, the system uses the information in the following table to determine the number of total and fractional digits required to produce a packed decimal (*DEC) result.

Operation	Result (total digits)	Result (fractional digits)
+	MAX(d1-f1, d2-f2)+MAX(f1,f2)+1	MAX(f1, f2)
−	MAX (d1-f1, d2-f2)+MAX (f1,f2)+1	MAX(f1, f2)
*	d1+d2	f1+f2
/	31	31-(d1-f1+f2)
//	MIN(d1-f1,d2-f2)+MAX(f1,f2)	MAX(f1,f2)
Legend:		
d1	Total digits in operand 1	
f1	Fractional digits in operand 1	
d2	Total digits in operand 2	
f2	Fractional digits in operand 2	

If both operands are zero-precision binary fields and/or integer constants, the result is a *BIN4, unless the operator is a "/". In that case, the result is the same as for a fixed-point result. If the total number of digits required exceeds 31, the number of fraction digits is reduced enough to enable calculation of the result with a total of 31 digits. If some fraction digits are dropped and the attributes of the end result of the computation (the attributes specified on the MAPFLD parameter for the field) require greater precision than that of the intermediate result, a warning message is sent to indicate that some precision was lost in evaluating the expression.

When a numeric or string expression is specified on the MAPFLD parameter, the attributes of the final result are used in one of the two ways. They are either used directly for the field value (if field-type *CALC is specified and the field is not contained in the prototype record format identified on the FORMAT parameter), or the final result is changed to match the attributes specified on the MAPFLD parameter or contained in the field definition in the record format identified by the FORMAT parameter.

Both operands of a relational operator can be constants. The fields, constants, or expressions specified as operands on the left and right side of a relational operator must be of the same type, either numeric or string. Any combination of character and DBCS field operands are allowed except that a character field cannot be related to a DBCS-only field.

There are two types of DBCS constants: DBCS-only and DBCS-open. A DBCS-only constant has only DBCS data between its single quotation marks. This data must be enclosed in SO/SI characters. A DBCS-open constant has a mixture of DBCS and alphameric data. An SO character (HEX 0E) indicates the start of a group of DBCS characters and an SI character (HEX 0F) follows the last double-byte character of the group.

If a numeric or string expression appears as a complex selection operand on the QRYSLT or GRPSLT parameter, the attributes of the final result of the expression used for the selection operand are changed to match the other relational operand.

It is not necessary for operands of a relational operator to have identical attributes. If the operands do not have identical attributes, the system changes them to identical attributes (except for the *CT operator, where the character string operands might be of different lengths), before performing the operation. This change uses packed decimal format if both operands are fixed-point numeric operands, or floating-point format if either operand is a floating-point number. The changes for fixed-point numeric operands align their decimal points and pad them with zeros. Numeric type changes might truncate fractional digits if more than 31 total digits are required for fixed-point numbers, or might drop some of the least significant digits if more than 15 total digits are required for binary floating-point numbers. Character operands are changed by padding the shorter operand with blanks.

The *CT operator performs a scan of the character field or string expression (except for expressions made up of a single character string literal) that must be specified as the left side of the relation, in order to determine if it contains the character string, field, or expression value specified as the right side of the relation. The second operand (the search value) must be no longer than the first operand (the base string).

If the string is found, the relation is satisfied and the result is a logical value of 'true'; otherwise, the result is a logical 'false' value. The following example illustrates this process:
* Field BASEFLD contains the value 'THIS IS A TEST'.
* Field TESTFLD contains the value 'TE'.

Expression	Result
'BASEFLD *CT ''IS A'''	True
'BASEFLD *CT TESTFLD	True

Expression	Result
'BASEFLD *CT ''X'''	False
'BASEFLD *CT TESTFLD | | ''Z'''	False
'BASEFLD | | ''ABC'' *CT ''TAB'''	True

Zero-length literal support changes the results of a comparison when used as the compare argument of the *CT operator. A *zero-length literal* is denoted as a quoted string with nothing, not even a blank, between the quotation marks ('''').

Consider this statement:
```
QRYSLT('field *CT ""')
```

With zero-length literal support, the statement returns records that contain anything. It is, in essence, a wildcard comparison for any number of characters followed by any number of characters. It is equivalent to the following expression:
```
'field = %WLDCRD("**")'
```

Before zero-length literal support was introduced, the argument ('''') was interpreted as a single-byte blank. The statement returned records that contained a single blank somewhere in the field. It was, in essence, a wildcard comparison for any number of characters, followed by a blank, followed by any number of characters. It was equivalent to the following expression:
```
'field = %WLDCRD("* *")'
```

To get the same result with the *CT operator, you must code the QRYSLT parameter to explicitly look for the blank:
```
QRYSLT('field *CT " "')
```

Built-in functions:

These built-in functions are supported for an expression that is used to define a derived field on the MAPFLD parameter or for a complex selection operand specified on the QRYSLT or GRPSLT parameter.

A numeric argument is a numeric field, a numeric constant or a numeric expression. A string argument is a character field, a character string literal, or a string expression. Unless otherwise noted, all built-in functions allow expressions, including other built-in functions, to be used as arguments.

For a field that appears in the record format identified by the FORMAT parameter, and that is also defined by an expression on the MAPFLD parameter, the expression result is calculated by using the attributes described in the following paragraphs. Then the resultant value is mapped to match the attributes of the field.

%ABSVAL (numeric-argument)
> %ABSVAL accepts a numeric argument and returns the absolute value of the argument. The returned value has the same attributes as the argument, unless the argument is a *BIN2, in which case the returned value is a *BIN4.

> The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a packed decimal number (*DEC) with 8 digits and 0 precision (date duration), 6 digits and 0 precision (time duration), or 20 digits and 6 precision (timestamp duration).

%ACOS (numeric-argument)
> %ACOS accepts a numeric argument and returns the arc cosine of the argument, in radians. %ACOS and %COS are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%AND (string-argument ...)

%AND accepts two or more character or hexadecimal string arguments and returns a string that is the bit-wise 'AND' (logical and) of the arguments. This function takes the first argument string, ANDs it with the next string, and continues to AND each successive argument with the previous result. If an argument is encountered that is shorter than the previous result, it is padded on the right with blanks. The returned value is a string of type *HEX with the same length as the longest argument. If any of the arguments are variable-length, the maximum length is used as the length of the argument.

%ANTILOG (numeric-argument)

%ANTILOG accepts a numeric argument and returns the antilogarithm (base 10) of the argument. %ANTILOG and %LOG are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8), or a decimal floating-point number (34 digits) if the argument is decimal floating-point.

%ASIN (numeric-argument)

%ASIN accepts a numeric argument and returns the arc sine of the argument, in radians. %ASIN and %SIN are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%ATAN (numeric-argument)

%ATAN accepts a numeric argument and returns the arc tangent of the argument, in radians. %ATAN and %TAN are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%ATANH (numeric-argument)

%ATANH accepts a numeric argument and returns the hyperbolic arc tangent of the argument, in radians. %ATANH and %TANH are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%AVG (numeric-argument)

%AVG accepts a numeric argument and returns the average value of its argument for the group of records defined on the GRPFLD parameter. The argument must be a field name or an expression (not a literal).

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. If no records are selected, the result is the null value. Otherwise,

- If the argument is fixed-point, the result is a packed decimal number (*DEC) with 31 total digits and the same number of integer digits as the argument.

- If the argument is binary floating-point, the result is a double-precision binary floating-point number (*FLT8).
- If the argument is decimal floating-point, the result is a decimal floating-point number (34 digits).
- If the argument is date duration, time duration, or timestamp duration, the returned value is a packed decimal number (*DEC) with 31 digits and 0 precision (date duration), 31 digits and 0 precision (time duration), or 31 digits and 6 precision (timestamp duration).

%AVG is an aggregate function that is used for a nongrouping field in a query that uses the grouping function.

%CHAR (date/time-argument date/time-format)

%CHAR accepts a date/time argument and date/time format and returns the character representation of the argument using the specified format. The date/time argument can be a date, time, or timestamp field. The returned value is of type *CHAR and is tagged with the CCSID of the current job.

The date/time format is optional. If specified, it must be one of the following formats:

EUR　European format

ISO　International Standards Organization format

JIS　Japanese Industrial Standard format

USA　United States format

If the format is not specified, the job default format is used.

Example:
```
OPNQRYF
  FILE(library/file)
    GRPFLD(charfld)
    GRPSLT('charfld = %CHAR(timefld "USA")')
```

%COS (numeric-argument)

%COS accepts a numeric argument and returns the cosine of the argument. The argument must be specified in radians. %COS and %ACOS are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%COSH (numeric-argument)

%COSH accepts a numeric argument and returns the hyperbolic cosine of the argument. The argument must be specified in radians.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%COT (numeric-argument)

%COT accepts a numeric argument and returns the cotangent of the argument. The argument must be specified in radians.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%COUNT

%COUNT does not support any arguments. It returns the count of the number of records contained in the group of records defined on the GRPFLD parameter. The returned value is a 4-byte binary number (*BIN4) with 10 total decimal digits and no fraction digits. %COUNT is an aggregate function that applies only to a query that uses the grouping function.

%CURDATE

%CURDATE does not support any arguments. It obtains the current date based on a reading of the time-of-day clock. The returned value is of type *DATE. The format and separator are derived from the job attributes.

%CURSERVER

%CURSERVER does not support any arguments. If only non-distributed files are specified then it obtains the current server name (or RDB name) of the local system. If distributed files are specified then it obtains the current server name (or RDB name) of the COORDINATOR node. The returned value is of type variable-length character (*VCHAR) with a maximum length of 18.

%CURTIME

%CURTIME does not support any arguments. It obtains the current time based on a reading of the time-of-day clock. The returned value is of type *TIME. The format and separator are derived from the job attributes.

%CURTIMESTP

%CURTIMESTP does not support any arguments. It obtains the current timestamp based on a reading of the time-of-day clock. The returned value is of type *TIMESTP. The format and separator will be derived from the job attributes.

%CURTIMEZONE

%CURTIMEZONE does not support any arguments. It obtains the current time zone. The returned value is a packed decimal number (*DEC) with 6 digits and 0 precision.

%DATE (date/time-argument)

%DATE accepts a date/time argument and returns a date. The date/time argument can be a date or timestamp field, a character or hexadecimal field containing the external form of a date, a date literal, or a numeric field or literal value in the range 1 - 3,652,059. The returned value is of type *DATE.

Example:
```
OPNQRYF
    FILE(library/file)
    QRYSLT(('%DATE(tstampfld) =
        "1989-10-23"'))
```

%DAY (date/time-argument)

%DAY accepts a date/time argument and returns the day part of the value. The date/time argument can be a date or timestamp field, a date duration or timestamp duration (field or literal), or a numeric field or literal. The returned value is of type *BIN4.

A numeric field argument must be defined as packed decimal (*DEC) with 8 digits and 0 precision for date duration or packed decimal (*DEC) with 20 digits and 6 precision for timestamp duration. A numeric constant argument must have 8 digits followed by a decimal point, or 14 digits followed by a decimal point and 6 digits.

%DAYS (date/time-argument)

%DAYS accepts a date/time argument and returns an integer representation of the date. The date/time argument can be a date or timestamp field, a character or hexadecimal field containing the external form of a date, or a date literal. The returned value is of type *BIN4.

%DIGITS (numeric-argument)

%DIGITS accepts a numeric argument and returns a character representation of its numeric value, not including the sign or a decimal point. The result is tagged with the CCSID of the current job. For example, %DIGITS (-1.5) returns the character string 15. The numeric argument must not be a floating point number.

%DURDAY (integer-argument)
%DURDAY accepts an integer argument and returns a labeled duration of days. The integer argument for this function can be a numeric expression, a field, or a literal.

This built-in function is allowed to stand by itself in the *mapped-field-definition* of the MAPFLD parameter, and is allowed as part of an arithmetic (addition or subtraction) expression with a date or timestamp field on the QRYSLT, GRPSLT, or MAPFLD parameters.

%DURHOUR (integer-argument)
%DURHOUR accepts an integer argument and returns a labeled duration of hours. The integer argument for this function can be a numeric expression, a field, or a literal.

This built-in function is allowed to stand by itself in the *mapped-field-definition* on the MAPFLD parameter, and is allowed as part of an arithmetic (addition or subtraction) expression with a time or timestamp field on the QRYSLT, GRPSLT, or MAPFLD parameters.

%DURMICSEC (integer-argument)
%DURMICSEC accepts an integer argument and returns a labeled duration of microseconds. The integer argument for this function can be a numeric expression, a field, or a literal.

This built-in function is allowed to stand by itself in the *mapped-field-definition* on the MAPFLD parameter, and is allowed as part of an arithmetic (addition or subtraction) expression with a timestamp field on the QRYSLT, GRPSLT, or MAPFLD parameters.

%DURMINUTE (integer-argument)
%DURMINUTE accepts an integer argument and returns a labeled duration of minutes. The integer argument for this function can be a numeric expression, a field, or a literal.

This built-in function is allowed to stand by itself in the *mapped-field-definition* on the MAPFLD parameter, and is allowed as part of an arithmetic (addition or subtraction) expression with a time or timestamp field on the QRYSLT, GRPSLT, or MAPFLD parameters.

%DURMONTH (integer-argument)
%DURMONTH accepts an integer argument and returns a labeled duration of months. The integer argument for this function can be a numeric expression, a field, or a literal.

This built-in function is allowed to stand by itself in the *mapped-field-definition* on the MAPFLD parameter, and is allowed as part of an arithmetic (addition or subtraction) expression with a date or timestamp field on the QRYSLT, GRPSLT, or MAPFLD parameters.

%DURSEC (integer-argument)
%DURSEC accepts an integer argument and returns a labeled duration of seconds. The integer argument for this function can be a numeric expression, a field, or a literal.

This built-in function is allowed to stand by itself in the *mapped-field-definition* on the MAPFLD parameter, and is allowed as part of an arithmetic (addition or subtraction) expression with a time or timestamp field on the QRYSLT, GRPSLT, or MAPFLD parameters.

%DURYEAR (integer-argument)
%DURYEAR accepts an integer argument and returns a labeled duration of years. The integer argument for this function can be a numeric expression, a field, or a literal.

This built-in function is allowed to stand by itself in the *mapped-field-definition* value on the MAPFLD parameter, and is allowed as part of an arithmetic (addition or subtraction) expression with a date or timestamp field on the QRYSLT, GRPSLT, or MAPFLD parameters.

Example:

```
OPNQRYF
   FILE((library/file))
   QRYSLT('startfld > %CURDATE + oneyear *AND
           endfld   < %CURDATE + %DURYEAR(2)')
   MAPFLD((oneyear '%DURYEAR(1)'))
```

%EXP (numeric-argument)

%EXP accepts a numeric argument and returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument. %EXP and %LN are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8), or a decimal floating-point number (34 digits) if the argument is decimal floating-point.

%HASH (expression-argument)

%HASH accepts a valid expression and returns a 4-byte binary number (*BIN4) with 10 total decimal digits and no fraction digits. The returned value will be the partition number of the record selected.

A valid expression cannot include aggregate functions such as %COUNT, %AVG, %MIN, %MAX, %SUM, and %STDDEV as operands to %HASH.

Use the %HASH function to determine what the partitions are if the partitioning key is composed of EMPNO and LASTNAME. The following example returns the partition number for every row in EMPLOYEE.

Example:
```
OPNQRYF
     FILE((CORPDATA/EMPLOYEE))
     FORMAT(FNAME)
     MAPFLD((HASH '%HASH((1/EMPNO) (1/LN))'))
```

%HEX (hexadecimal-argument)

%HEX accepts an argument and returns the hexadecimal equivalent of the argument's value. The hexadecimal argument can be of any type. The returned value is of type *CHAR, and is tagged with the CCSID of the current job.

%HOUR (date/time-argument)

%HOUR accepts a date/time argument and returns the hour part of the value. The date/time argument can be a time or timestamp field, a time duration or timestamp duration (either field or literal), or a numeric field or literal. The returned value is of type *BIN4.

A numeric field argument must be defined as packed decimal (*DEC) with 6 digits and 0 precision for time duration or packed decimal (*DEC) with 20 digits and 6 precision for timestamp duration. A numeric constant argument must have 6 digits followed by a decimal point, or 14 digits followed by a decimal point and 6 digits.

Example:
```
Example:
  OPNQRYF
     FILE(library/file)
     QRYSLT(('%HOUR(timefld2) = 12'))
```

%LEN (length-argument)

%LEN accepts one argument and returns the number of bytes used to represent the value unless the value is a graphic field type. If the value is a graphic field type, the number of graphic characters is returned. The length argument can be of any type. The returned value is of type *BIN4.

Example:

```
OPNQRYF
    FILE(library/file)
    QRYSLT('%LEN(varlenfld) <= 30')
```

Argument Type	Result Length in Bytes
Character	1-32766
Hex	1-32766
DBCS-only	4-32766
DBCS-either	4-32766
DBCS-open	4-32766
DBCS-graphic	1-16383
Variable Character	0-32740
Variable Hex	0-32740
Variable DBCS-only	0-32740
Variable DBCS-either	0-32740
Variable DBCS-open	0-32740
Variable DBCS-graphic	0-16370
Date	4
Time	3
Timestamp	10
Binary *BIN4	2
Binary *BIN8	4
Binary floating point *FLT4	4
Binary floating point *FLT8	8
Decimal floating point (16 digits)	8
Decimal floating point (34 digits)	16
Packed decimal (p,s)	INTEGER(p/2)+1, (1-16)
Zoned decimal (p,s)	p (1-31)

p=precision, s=scale

String notes: The %LEN function returns the length of the value as it is stored in the data space.

- For fixed-length fields, the length is always the same as the declared size of the field, not the length of the actual data in the field.

- For variable-length fields, the length is the length of the actual data in the field, including trailing blanks.

For example, assume FIXED10 is a *CHAR(10) field, and VAR10 is a *VCHAR(10) field. The following example shows results of the %LEN function:

%LEN Statement	Field Data	Result
%LEN(fixed10)	'1234567890'	10
%LEN(fixed10)	'12345'	10
%LEN(var10)	'1234567890'	10
%LEN(var10)	'12345'	5
%LEN(var10)	'12345 '	7
%LEN(var10)	''	0

%LN (numeric-argument)

%LN accepts a numeric argument and returns the natural logarithm of the argument. %LN and %EXP are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8), or a decimal floating-point number (34 digits) if the argument is decimal floating-point.

%LOG (numeric-argument)

%LOG accepts a numeric argument and returns the common logarithm (base 10) of the argument. %LOG and %ANTILOG are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values.

The returned value is a double-precision binary floating-point number (*FLT8), or a decimal floating-point number (34 digits) if the argument is decimal floating-point.

%MAX (numeric-or-string-or-date/time-argument ...)

%MAX accepts one or more character-string, DBCS-string, numeric, or date/time arguments, and returns the largest value from the list. Date/time arguments are arguments of type *DATE, *TIME, or *TIMESTP, or arguments that are date, time, or timestamp durations. String arguments must be no longer than 256 bytes.

If only one argument is specified, this function returns the maximum value of its argument for the group of records defined on the GRPFLD parameter, and the returned value has the same attributes as the argument. If no records are selected, the result is the null value. If the single argument is a date duration, time duration, or timestamp duration, then the returned value is a packed decimal number (*DEC) with 8 digits and 0 precision (date duration), 6 digits and 0 precision (time duration), or 20 digits and 6 precision (timestamp duration). When a single argument is used, it must be a field name or an expression (not a literal). %MAX with only one argument is an aggregate function that is used for a nongrouping field in a query that uses the grouping function.

If multiple arguments are specified, %MAX returns the maximum value of all the arguments. All arguments must be either character-string, DBCS-string, numeric, or date/time values. This function calculates the maximum value of the first two arguments, and then continues to determine the maximum value of the previous result and the next successive argument. The final result is determined according to the following value conversion rules.

If an argument has different attributes than the previous result, the two values are converted to identical attributes and the operation continues. This conversion uses packed decimal if both values are fixed-point numeric values, uses decimal floating point if either value is decimal floating-point, or uses binary floating point if either value is binary floating-point. The conversion for fixed-point numeric values aligns the decimal points and pads the values with zeros. Numeric type changes might truncate fractional digits if more than 31 total digits are required for fixed-point numbers, or drop some of the least significant digits if more than 15 total digits are required for binary floating-point numbers. Character values are changed by padding the shorter field with blanks.

%MICSEC (date/time-argument)

%MICSEC accepts a date/time argument and returns the microsecond part of the value. The date/time argument can be a timestamp (field or literal), a timestamp duration (field or literal), a character field that contains the external form of a timestamp, or a numeric field or literal. The returned value is of type *BIN4. A numeric field argument must be defined as packed decimal (*DEC) with 20 digits and 6 precision for timestamp duration. A numeric constant argument must be 14 digits followed by a decimal point and 6 digits.

%MIN (numeric-or-string-or-date/time-argument ...)

%MIN accepts one or more character-string, DBCS-string, numeric, or date/time arguments, and returns the smallest value from the list. Date/time arguments are arguments of type *DATE, *TIME, or *TIMESTP, or arguments that are date, time, or timestamp durations. String arguments must be no longer than 256 bytes.

If only one argument is specified, this function returns the minimum value of its argument for the group of records defined on the GRPFLD parameter, and the returned value has the same attributes as the argument. If no records are selected, the result is the null value. If the single argument is a date duration, time duration, or timestamp duration, then the returned value is a packed decimal number (*DEC) with 8 digits and 0 precision (date duration), 6 digits and 0 precision (time duration), or 20 digits and 6 precision (timestamp duration). When a single argument is used, it must be a field name or an expression (not a literal). %MIN with only one argument is an aggregate function that is used for a nongrouping field in a query that uses the grouping function.

If multiple arguments are specified, %MIN returns the minimum value of all the arguments. All arguments must be either character-string, DBCS-string, numeric, or date/time values. This function calculates the minimum value of the first two arguments, and then continues to determine the minimum value of the previous result and the next successive argument. The final result is determined by the value change rules described below.

If an argument has different attributes than the previous one, the two values are changed to identical attributes and the operation continues. This change uses packed decimal numbers if both values are fixed-point numeric values, uses decimal floating-point numbers if either value is a decimal floating-point number, or uses binary floating-point numbers if either value is a binary floating-point number. The change for fixed-point numeric values aligns the decimal points and pads with zeros. Numeric type change might truncate fractional digits if more than 31 total digits are required for fixed-point numbers, or might drop some of the least significant digits if more than 15 total digits are required for binary floating-point numbers. Character values are changed by padding the shorter field with blanks.

%MINUTE (date/time-argument)

%MINUTE accepts a date/time argument and returns the minute part of the value. The date/time argument can be a time or timestamp field, a time duration or timestamp duration (either field or literal), or a numeric field or literal. The returned value is of type *BIN4.

A numeric field argument must be defined as packed decimal (*DEC) with 6 digits and 0 precision for time duration or packed decimal (*DEC) with 20 digits and 6 precision for timestamp duration. A numeric constant argument must have 6 digits followed by a decimal point, or 14 digits followed by a decimal point and 6 digits.

%MONTH (date/time-argument)

%MONTH accepts a date/time argument and returns the month part of the value. The date/time argument can be a date or timestamp field, a date duration or timestamp duration (field or literal), or a numeric field or literal. The returned value is of type *BIN4.

A numeric field argument must be defined as packed decimal (*DEC) with 8 digits and 0 precision for date duration or packed decimal (*DEC) with 20 digits and 6 precision for timestamp duration. A numeric constant argument must have 8 digits followed by a decimal point, or 14 digits followed by a decimal point and 6 digits.

%NODENAME (integer-argument)

%NODENAME accepts an integer-argument which is used to identify a file that has been specified on the FILE parameter. The argument must be greater than 0 and less than or equal to the number of files specified on the file parameter. The %NODENAME function returns the RDB name for the record retrieved. The returned value is of type *VCHAR of length 18.

Find the node name for every record of the EMPLOYEE table.

Example:
```
OPNQRYF
     FILE((CORPDATA/EMPLOYEE))
     FORMAT(FNAME)
     MAPFLD((NODENAME '%NODENAME(1)'))
```

Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO) and determine the node from which each record involved in the join originated.

Example:
```
OPNQRYF
  FILE((CORPDATA/EMPLOYEE) (CORPDATA/DEPARTMENT))
  FORMAT(FNAME)
  JFLD((EMPLOYEE/DEPTNO DEPARTMENT/DEPTNO *EQ))
  MAPFLD((EMPNO 'EMPLOYEE/EMPNO')
         (NODENAME1 '%NODENAME(1)')
         (NODENAME1 '%NODENAME(2)'))
```

Join the EMPLOYEE and DEPARTMENT tables, select all records of the result where the records of the two tables are on the same node.

Example:

```
OPNQRYF
  FILE((CORPDATA/EMPLOYEE) (CORPDATA/DEPARTMENT))
  FORMAT(FNAME)
  JFLD((1/NODENAME1 2/NODENAME2 *EQ))
  MAPFLD((NODENAME1 '%NODENAME(1)')
         (NODENAME2 '%NODENAME(2)'))
```

%NODENUMBER (integer-argument)

%NODENUMBER accepts an integer-argument which is used to identify a file that has been specified on the FILE parameter. The argument must be greater than zero and less than or equal to the number of files specified on the file parameter. The %NODENUMBER function returns a 4-byte binary number (*BIN4) with 10 total decimal digits and no fraction digits. The returned value will be the node number of the record selected.

If the argument identifies a non-distributed file, the value zero is returned.

For OPNQRYF the node number from the secondary file where the outer or exception join is performed will be returned.

If CORPDATA.EMPLOYEE is a distributed file, then the node number for each record and the employee name will returned.

Example:

```
OPNQRYF
    FILE((CORPDATA/EMPLOYEE))
    FORMAT(FNAME)
    MAPFLD((NODENAME '%NODENUMBER(1)')
           (LNAME '1/LASTNAME'))
```

%NONNULL (argument ...)

%NONNULL accepts a list of two or more arguments and returns the first non-null value from the list. The items in the argument list can be fields or literal values of any type. The type of the returned value is that of the item selected from the list.

Example:

```
OPNQRYF
    FILE(library/file)
    QRYSLT('%NONNULL(fld1 fld2 0) > 0')
```

The above example selects records from the file where either field FLD1 or field FLD2 contains a non-null value that is greater than zero. If both FLD1 and FLD2 were null, the %NONNULL function specified in this example would return '0' because of the constant '0' passed as the third argument. If any field is DBCS-graphic, all fields must be DBCS-graphic.

%NOT (string-argument)

%NOT accepts a character or hexadecimal string argument and returns a string that is the bit-wise 'NOT' (logical not) of the argument. The returned value is a string of type *HEX with the same length as the argument.

%OR (string-argument ...)

%OR accepts two or more character-string arguments and returns a string that is the bit-wise 'OR' (logical inclusive or) of the arguments. This function takes the first argument string, ORs it with the next string, and then continues to OR each successive argument with the previous result. If an argument is encountered that is shorter than the previous result, it is padded with blanks. The final result is a string with the same length as the longest argument. If any of the arguments are variable-length, the maximum length is used as the length of the argument.

%PARTITION (integer-argument)

%PARTITION accepts an integer-argument which is used to identify a file that has been specified on the FILE parameter. The argument must be greater than 0 and less than or equal to the number of files specified on the file parameter. The partition function returns a 4-byte binary number (*BIN4) with 10 total decimal digits and no fraction digits. The returned value will be the partition number of the record.

If the argument identifies a non-distributed file then a value of zero will be returned.

Find the PARTITION number for every row of the EMPLOYEE table. This can be used to determine if there is data skew.

Example:
```
OPNQRYF FILE((CORPDATA/EMPLOYEE))
     FORMAT(FNAME)
     MAPFLD((PART1 '%PARTITION(1)'))
```

Select the employee number (EMPNO) from the EMPLOYEE table for all records where the partition number is equal to 100.

Example:
```
OPNQRYF
     FILE((EMPLOYEE))
     QRYSLT('%PARTITION(1) *EQ 100')
```

Join the EMPLOYEE and DEPARTMENT tables, select all records of the result where the records of the two tables have the same partition number

Example:
```
OPNQRYF
     FILE((CORPDATA/EMPLOYEE) (CORPDATA/DEPARTMENT))
     FORMAT(FNAME)
     JFLD((1/PART1 2/PART2 *EQ))
     MAPFLD((PART1 '%PARTITION(1)')
     (PART2 '%PARTITION(2)'))
```

%SECOND (date/time-argument)

%SECOND accepts a date/time argument and returns the seconds part of the value. The date/time argument can be a time or timestamp field, a time duration or timestamp duration (either field or literal), or a numeric field or literal. The returned value is of type *BIN4.

A numeric field argument must be defined as packed decimal (*DEC) with 6 digits and 0 precision for time duration or packed decimal (*DEC) with 20 digits and 6 precision for timestamp duration. A numeric constant argument must have 6 digits followed by a decimal point, or 14 digits followed by a decimal point and 6 digits.

%SIN (numeric-argument)

%SIN accepts a numeric argument and returns the sine of the argument. The argument must be specified in radians. %SIN and %ASIN are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%SINH (numeric-argument)

%SINH accepts a numeric argument and returns the hyperbolic sine of the argument. The argument must be specified in radians.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values.

| The returned value is a double-precision binary floating-point number (*FLT8). The numeric
| argument must not be a decimal floating-point number.

%SQRT (numeric-argument)

%SQRT accepts a numeric argument and returns the square root of the argument.

| The following argument types are treated as numeric values: date duration, time duration, and
| timestamp duration. Arguments of these types can be specified either as fields or literal values.
| The returned value is a double-precision binary floating-point number (*FLT8), or a decimal
| floating-point number with the same number of digits as the argument if the argument is decimal
| floating-point.

%SST (string-argument start-position-expression <length-expression>)

%SST and %SUBSTRING accept a character, hexadecimal, DBCS, or graphic string, a starting
position expression, and an optional length expression as arguments. They return a substring of
the string argument that is of the same type and CCSID as the string argument and has length
equal to the value specified by the length-expression.

Single-byte substringing is done when these functions (%SST and %SUBSTRING) are used for
DBCS data. The shift-out and shift-in characters might be lost, which produces unusual results.
The result of the DBCS substring operation is the DBCS-open type.

The string argument can be a fixed- or variable-length character, hexadecimal, DBCS, or graphic
field or an expression which evaluates to a fixed- or variable-length character, hexadecimal,
DBCS, or graphic string.

The values derived from expressions for the second and third arguments must be valid integers.
The second argument must have a value between 1 and the length attribute (or maximum length
of a variable-length field) of the first argument, and the third argument must have a value
between 1 and the length attribute (or maximum length of a variable-length field) of the first
argument.

If an argument is DBCS-graphic, the second and third arguments must also be specified as
DBCS-graphic characters, not bytes.

If an expression is given for the second or third arguments, the expression must be enclosed in
parentheses.

If the expressions evaluate to variable-length results, no validation of the range of these
expressions is guaranteed and errors might occur during input/output processing.

The maximum value allowed for the third argument (length) is 32766 except for DBCS-graphic,
which is 16383. However, if the third operand is represented by an expression, this causes the
result to be variable-length. Thus, the value of the expression cannot exceed 32740 except for
DBCS-graphic, which cannot exceed 16370.

The user can omit the third argument. If the third argument is not specified and the first
argument is:

* fixed-length, the default value for the third argument is LENGTH(argument_1) -
  value_for_argument_2 + 1

* variable-length, the default value for the third argument is the maximum of 0 and
  LENGTH(argument_1) - value_for_argument_2 + 1

* variable-length with a length less than the value for argument_2, the default value for the third
  argument is zero and the result is the empty string.

Example:

```
OPNQRYF
    FILE(library/file)
    QRYSLT('field1 =
    %SST(field2 (numfld1+3)
    (numfld1+numfld2))')
```

%STDDEV (numeric-argument)

%STDDEV accepts a numeric argument and returns the standard deviation of its argument for the group of records defined by the GRPFLD parameter. The argument must be a field name or an expression (not a literal). If no records are selected, the result is the null value. Otherwise, the returned value is a double-precision binary floating-point number (*FLT8), or a decimal floating-point number (34 digits) if the argument is decimal floating-point. %STDDEV is an aggregate function that is used for a nongrouping field in a query that uses the grouping function.

%STRIP (string-argument <strip-character><strip-function>)

%STRIP accepts a character-, DBCS-, or graphic-string argument, an optional strip character, and an optional strip function as arguments. It returns a result string with the strip character removed from the string argument as specified by the strip function.

The string argument can be a literal, a fixed or variable-length character, hexadecimal, DBCS, or graphic field, or an expression which evaluates to a fixed- or variable-length character, hexadecimal, DBCS, or graphic string.

The strip character must be a single character, enclosed in apostrophes, with a data type compatible to the source string. The default is a single SBCS space for character data, DBCS-open, and DBCS-either, a single DBCS space for DBCS-only data, and a single graphic space for graphic data.

The strip function can be one of three functions:

*LEAD

Remove leading strip character(s)

*TRAIL

Remove trailing strip character(s)

*BOTH

Remove both leading and trailing strip character(s)

The default strip function is *BOTH.

Example:

```
OPNQRYF
    FILE(library/file)
    QRYSLT('%STRIP(fld '.' *TRAIL) = 'Mr')
```

%SUBSTRING (string-field-name start-position length)

%SUBSTRING performs the same operation as %SST. See the %SST description for more information.

%SUM (numeric-argument)

%SUM accepts a numeric argument and returns the sum of all the values for its argument in the group of records defined on the GRPFLD parameter and must be enclosed in parentheses. The argument must be a field name or an expression (not a literal).

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. If no records are selected, the result is the null value. Otherwise,

- If the argument is a binary floating-point number, the returned value is a double-precision binary floating-point number (*FLT8).
- If the argument is a decimal floating-point number, the returned value is a decimal floating-point number (34 digits).
- If the argument is a binary number with zero-precision, the returned value is *BIN4.
- If the argument is a binary number with nonzero precision or a fixed-point number, the returned value is a packed decimal number (*DEC) with 31 total digits and as many fractional digits as the argument.

- If the argument is of type date duration, time duration, or timestamp duration, the returned value is a double-precision binary floating-point number (*FLT8).

%SUM is an aggregate function that is used for a nongrouping field in a query that uses the grouping function.

%TAN (numeric-argument)
%TAN accepts a numeric argument and returns the tangent of the argument. The argument must be specified in radians. %TAN and %ATAN are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The return value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%TANH (numeric-argument)
%TAN accepts a numeric argument and returns the hyperbolic tangent of the argument. The argument must be specified in radians. %TANH and %ATANH are inverse operations.

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. Arguments of these types can be specified either as fields or literal values. The returned value is a double-precision binary floating-point number (*FLT8). The numeric argument must not be a decimal floating-point number.

%TIME (date/time-argument)

%TIME accepts a date/time argument and returns a time. The date/time argument can be a time or timestamp field, a character or hexadecimal field containing the external form of a time, or a time literal. The returned value is of type *TIME.

%TIMESTP (date/time-argument date/time-argument)
%TIMESTP accepts one or two date/time arguments and returns a timestamp.
- If only one date/time argument is specified, it must be a timestamp (field or literal), or a character or hexadecimal field containing the external form of a timestamp.
- If both arguments are specified,
    1. The first date/time argument must be a date (field or literal), or a character or hexadecimal field containing the external form of a date.
    2. The second date/time argument must be a time (field or literal), or a character or hexadecimal field containing the external form of a time.

The returned value is of type *TIMESTP.

%USER
%USER does not support any arguments. It returns the user profile name of the job in which the query is running. The returned value is of type variable-length character (*VCHAR) with a maximum length of 18.

Example:
```
OPNQRYF
    FILE(library/file)
    QRYSLT('field = %USER')
```

%VAR (numeric-argument)
%VAR accepts a numeric argument and returns the variance of its argument for the group of records defined by the GRPFLD parameter. The argument must be a field name or an expression (not a literal).

The following argument types are treated as numeric values: date duration, time duration, and timestamp duration. If no records are selected, the result is the null value. Otherwise, the returned value is a double-precision binary floating-point number (*FLT8), or a decimal

floating-point number (34 digits) if the argument is decimal floating-point. %VAR is an aggregate function that is used for a nongrouping field in a query that uses the grouping function.

%XLATE (string-argument qualified-table)

%XLATE accepts a character-string argument and the name of a table object (*TBL), and returns a string that is the value of the first argument translated by using the contents of the table. The returned value is a string with the same length and CCSID as the first argument.

The second argument must be a simple or qualified table object name. If no library name is specified, *LIBL is used to find the table.

%XOR (string-argument...)

%XOR accepts two or more character-string arguments and returns a string that is the bit-wise 'XOR' (logical exclusive or) of the arguments. This function takes the first argument string, XORs it with the next string, and then continues to XOR each successive argument with the previous result. If an argument is encountered that is longer than the previous result, the previous result is padded with blanks before the XOR operation. If any of the arguments is variable-length, the maximum length is used as the length of the argument. The final result is a string of type *HEX with the same length as the longest argument.

%YEAR

%YEAR accepts a date/time argument and returns the year part of the value. The date/time argument can be a date or timestamp field, a date duration or timestamp duration (field or literal), or a numeric field or literal. The returned value is of type *BIN4.

A numeric field argument must be defined as packed decimal (*DEC) with 8 digits and 0 precision for date duration or packed decimal (*DEC) with 20 digits and 6 precision for timestamp duration. A numeric constant argument must have 8 digits followed by a decimal point, or 14 digits followed by a decimal point and 6 digits.

Restricted built-in functions:

Some built-in functions are restricted in the way certain relational operators are specified on the QRYSLT and GRPSLT parameters.

The following built-in function is supported only as the second operand of the 'equal' or 'not-equal' relational operators specified on the QRYSLT and GRPSLT parameters:

%NULL

%NULL accepts no arguments. It is used to select or omit records based on whether a field in the record contains a null value.

Example:

```
OPNQRYF
    FILE(library/file)
    QRYSLT('charfld = %NULL')
```

This query would select all the records where 'charfld' contains the null value.

The following built-in functions are supported only as the second operand of the 'equal' relational operator specified on the QRYSLT and GRPSLT parameters:

%RANGE (low-value high-value)

%RANGE is used to identify the lower and upper boundaries for the value of a field or expression. %RANGE must be specified as the right side of a relation whose operator is equal. The low-value and high-value argument must be field names, character strings, or numeric literals, to match the type of field or expression specified as left side of the relation. For example, to select only records where the numeric field NBRFLD has a value ranging from 10 through 20, specify as follows:

```
'nbrfld = %RANGE(10 20)'
```

If the low-value argument is greater than the high-value argument, the relation produces a logical value of 'false'.

%VALUES (allowed-value...)

%VALUES is used to identify a list of allowed values for a field or expression. %VALUES must be specified as the right side of a relation whose operator is equal. The allowed-value arguments must be character string or numeric literals, to match the type of the field or expression specified as the left side of the relation. For example, to select only records where the second character of field CHARFLD has a value that is one of the values 'A', 'E', 'I', 'O', or 'U', specify as follows:

```
'%SST(charfld 2 1) =   %VALUES(''A'' ''E'' ''I'' ''O'' ''U'')'
```

%WLDCRD ("pattern-string" "wild-characters")

%WLDCRD is used to specify a pattern that performs a wildcard scan of the character or hexadecimal field or string expression (except for expressions made up of a single character-string literal) that must be specified as the left side of the relation. %WLDCRD must be specified as the right side of a relation whose operator is equal. The pattern-string argument must be a character-string, DBCS, or graphic literal, to match the left side of the relation. The wild-characters argument is an optional parameter that specifies what 'wildcard' characters are used in the pattern-string.

If specified for character data only (no DBCS data), the wild-characters argument must be a character-string literal of exactly two characters. The first character is the value that matches any single character in the search string. The second character is the value that matches a substring of any zero or more characters. The two characters must not be the same, but there is no requirement that either character appear in the pattern-string. If the wild-characters argument is omitted, the default is for an underline ('_') to match any single character and an asterisk ('*') to match a substring of any zero or more characters.

If the wild-characters argument is specified for DBCS data only (no character data), the argument must be a double-byte character-string literal of exactly two double-byte characters. The first double-byte character is the value that will match any one double-byte character in the search string. The second double-byte character is the value that will match a substring of any zero or more characters. The two double-byte characters must not be the same, but there is no requirement that either character appear in the pattern string. If the wild-characters argument is omitted, the default is for a DBCS underline to match any one double-byte character and a DBCS asterisk to match a substring of any zero or more double-byte characters.

If the wild-characters argument is specified for both character and DBCS data, in addition to the previous rules, the argument must first contain a single-byte character-string literal (two single-byte characters), then a double-byte character string (two double-byte characters).

In this case, the first character matches any single-byte character in the character string, the second character matches a substring of any number of single-byte or double-byte characters. The first double-byte character matches any double-byte character in the character string. The second double-byte character matches a substring of any number of single-byte or double-byte characters.

The following example selects only records where the character field CHARFLD contains a 'T', followed by any two characters and an 'E', appearing anywhere in the field.

```
'charfld = %WLDCRD(''*T__E*'')'
```

Note: The asterisks at the start and end of the pattern-string are required to allow the 'T' and 'E' to appear somewhere other than the first and last positions in the field:

To select only records where the character field CHARFLD starts with the string 'ABC', followed by one or more other characters and then followed by the string 'XYZ' (but not necessarily at the end of the field), specify the following.

```
'charfld = %WLDCRD(''ABC_*XYZ*'')'
```

To select only records where the second character of field CHARFLD is an asterisk ('*'), the last character is an underline ('_'), and the letter 'M' appears somewhere in between, specify as follows:

```
'charfld = %WLDCRD(''#*.M._'' ''#.'')'
```

Data type considerations:

Here are the rules that govern the comparison and arithmetic of date, time, and timestamp values and the use of large object and DATALINK data types on the Open Query File (OPNQRYF) command.

Comparing date, time, and timestamp using the Open Query File (OPNQRYF) command:

A date, time, or timestamp value can be compared either with another value of the same data type or with a string representation of that data type.

All comparisons are chronological, which means the farther a time is from January 1, 0001, the *greater* the value of that time.

Comparisons involving time values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds are implied.

Comparisons involving timestamp values are chronological without regard to representations that might be considered equivalent. Thus, the following predicate is true:

TIMESTAMP ('1990-02-23-00.00.00') > '1990-02-22-24.00.00'

When a character, DBCS-open, or DBCS-either field or constant is represented as a date, time, or timestamp, the following rules apply:

Date: The length of the field or literal must be at least 8 if the date format is *ISO, *USA, *EUR, *JIS, *YMD, *MDY, or *DMY. If the date format is *JUL (yyddd), the length of the variable must be at least 6 (includes the separator between yy and ddd). The field or literal can be padded with blanks.

Time: For all of the time formats (*USA, *ISO, *EUR, *JIS, *HMS), the length of the field or literal must be at least 4. The field or literal can be padded with blanks.

Timestamp: For the timestamp format (yyyy-mm-dd-hh.mm.ss.uuuuuu), the length of the field or literal must be at least 16. The field or literal can be padded with blanks.

Performing date, time, and timestamp arithmetic using the Open Query File (OPNQRYF) command:

Date, time, and timestamp values can be incremented, decremented, and subtracted. These operations might involve decimal numbers called *durations*.

Durations:

A *duration* is a number that represents an interval of time.

Date duration

A date duration represents a number of years, months, and days, expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format *yyyymmdd*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. The result of subtracting one date value from another, as in the expression HIREDATE - BRTHDATE, is a date duration.

Labeled duration

A labeled duration represents a specific unit of time as expressed by a number (which can be the result of an expression) used as an operand for one of the seven duration built-in functions: %DURYEAR, %DURMONTH, %DURDAY, %DURHOUR, %DURMINUTE, %DURSEC, or %DURMICSEC. The functions are for the duration of year, month, day, hour, minute, second, and microsecond, respectively. The number specified is converted as if it were assigned to a DECIMAL(15,0) number. A labeled duration can only be used as an operand of an arithmetic operator when the other operand is a value of data type *DATE, *TIME, or *TIMESTP. Thus, the expression HIREDATE + %DURMONTH(2) + %DURDAY(14) is valid, whereas the expression HIREDATE + (%DURMONTH(2) + %DURDAY(14)) is not. In both of these expressions, the labeled durations are %DURMONTH(2) and %DURDAY(14).

Time duration

A time duration represents a number of hours, minutes, and seconds, expressed as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format *hhmmss*, where *hh* represents the number of hours, *mm* the number of minutes, and *ss* the number of seconds. The result of subtracting one time value from another is a time duration.

Timestamp duration

A timestamp duration represents a number of years, months, days, hours, minutes, seconds, and microseconds, expressed as a DECIMAL(20,6) number. To be properly interpreted, the number must have the format *yyyymmddhhmmsszzzzzz*, where *yyyy, mm, dd, hh, mm, ss, and zzzzzz* represent, respectively, the number of years, months, days, hours, minutes, seconds, and microseconds. The result of subtracting one timestamp value from another is a timestamp duration.

Rules for date, time, and timestamp arithmetic:

The only arithmetic operations that can be performed on date and time values are addition and subtraction. If a date or time value is the operand of addition, the other operand must be a duration.

The specific rules governing the use of the addition operator with date and time values follow:
* If one operand is a date, the other operand must be a date duration or a labeled duration of years, months, or days.
* If one operand is a time, the other operand must be a time duration or a labeled duration of hours, minutes, or seconds.
* If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.

The rules for the use of the subtraction operator on date and time values are not the same as those for addition because a date or time value cannot be subtracted from a duration, and because the operation of subtracting two date and time values is not the same as the operation of subtracting a duration from a date or time value. The specific rules governing the use of the subtraction operator with date and time values follow:
* If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of years, months, or days.
* If the second operand is a date, the first operand must be a date or a string representation of a date.
* If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of hours, minutes, or seconds.
* If the second operand is a time, the first operand must be a time or string representation of a time.
* If the first operand is a timestamp, the second operand must be a timestamp, a string representation of a timestamp, or a duration.
* If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp.

Subtracting dates:

The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates.

The data type of the result is DECIMAL(8,0). If DATE1 is greater than or equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = DATE1 - DATE2.

```
If %DAY(DATE2) <= %DAY(DATE1) ;
    then %DAY(RESULT) = %DAY(DATE1) - %DAY(DATE2).

If %DAY(DATE2) > %DAY(DATE1) ;
    then %DAY(RESULT) = N + %DAY(DATE1) - %DAY(DATE2) ;
    where N = the last day of %MONTH(DATE2). ;
    %MONTH(DATE2) is then incremented by 1.

If %MONTH(DATE2) <= %MONTH(DATE1) ;
    then %MONTH(RESULT) = %MONTH(DATE1) - %MONTH(DATE2).

If %MONTH(DATE2) > %MONTH(DATE1) ;
    then %MONTH(RESULT) = 12 + %MONTH(DATE1) - %MONTH(DATE2). ;
    %YEAR(DATE2) is then incremented by 1.

%YEAR(RESULT) = %YEAR(DATE1) - %YEAR(DATE2).
```

For example, the result of %DATE('3/15/2000') - '12/31/1999' is 215 (or, a duration of 0 years, 2 months, and 15 days).

Incrementing and decrementing dates:

The result of adding a duration to a date or subtracting a duration from a date is itself a date.

(For the purposes of this operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001, and December 31, 9999, inclusive. If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a year that is not a leap year. In this case, the day is changed to 28.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would not be valid (September 31, for example). In this case, the day is set to the last day of the month.

Adding or subtracting a duration of days, of course, affects the day portion of the date, and potentially the month and year.

Date durations, whether positive or negative, can also be added to and subtracted from dates. As with labeled durations, the result is a valid date.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1 + X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression: DATE1 + %DURYEAR(%YEAR(X)) + %DURMONTH(%MONTH(X)) + %DURDAY(%DAY(X))

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1 - X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression: DATE1 - %DURDAY(%DAY(X)) - %DURMONTH(%MONTH(X)) - %DURYEAR(%YEAR(X))

When adding durations to dates, adding one month to a given date gives the same date one month later *unless* that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

Note: If one or more months are added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Subtracting times:

The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times.

The data type of the result is DECIMAL(6,0). If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1. If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = TIME1 - TIME2.

If %SECOND(TIME2) <= %SECOND(TIME1) ;
    then %SECOND(RESULT) = %SECOND(TIME1) - %SECOND(TIME2).

If %SECOND(TIME2) > %SECOND(TIME1) ;
    then %SECOND(RESULT) = 60 + %SECOND(TIME1) - %SECOND(TIME2). ;
    %MINUTE(TIME2) is then incremented by 1.

If %MINUTE(TIME2) <= %MINUTE(TIME1) ;
    then %MINUTE(RESULT) = %MINUTE(TIME1) - %MINUTE(TIME2).

If %MINUTE(TIME2) > %MINUTE(TIME1) ;
    then %MINUTE(RESULT) = 60 + %MINUTE(TIME1) - %MINUTE(TIME2). ;
    %HOUR(TIME2) is then incremented by 1.

%HOUR(RESULT) = %HOUR(TIME1) - %HOUR(TIME2).

For example, the result of %TIME('11:02:26') - '00:32:56' is 102930 (a duration of 10 hours, 29 minutes, and 30 seconds).

Incrementing and decrementing times:

The result of adding a duration to a time or subtracting a duration from a time is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time.

If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

Adding or subtracting a duration of seconds, of course, affects the seconds portion of the time, and potentially the minutes and hours.

Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order.TIME1 + X, where X is a DECIMAL(6,0) number, is equivalent to the expression: TIME1 + %DURHOUR(%HOUR(X)) + %DURMINUTE(%MINUTE(X)) + %DURSEC(%SECOND(X))

Subtracting timestamps:

The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and microseconds between the two timestamps.

The data type of the result is DECIMAL(20,6). If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = TS1 - TS2:

```
If %MICSEC(TS2) <= %MICSEC(TS1) ;
    then %MICSEC(RESULT) = %MICSEC(TS1) - ;
    %MICSEC(TS2).

If %MICSEC(TS2) > %MICSEC(TS1) ;
    then %MICSEC(RESULT) = 1000000 + ;
    %MICSEC(TS1) - %MICSEC(TS2) ;
    and %SECOND(TS2) is incremented by 1.
```

The seconds and minutes part of the timestamps are subtracted as specified in the rules for subtracting times:

```
If %HOUR(TS2) <= %HOUR(TS1) ;
    then %HOUR(RESULT) = %HOUR(TS1) - %HOUR(TS2).

If %HOUR(TS2) > %HOUR(TS1) ;
    then %HOUR(RESULT) = 24 + %HOUR(TS1) - %HOUR(TS2) ;
    and %DAY(TS2) is incremented by 1.
```

The date part of the timestamp is subtracted as specified in the rules for subtracting dates.

Incrementing and decrementing timestamps:

The result of adding a duration to a timestamp or subtracting a duration from a timestamp is itself a timestamp.

Date and time arithmetic is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. Microseconds overflow into seconds.

BLOB, CLOB, and DBCLOB considerations:

Fields of large object data types (BLOB, CLOB, or DBCLOB) cannot be directly accessed from an open query file. Also, they are not allowed on some parameters of the Open Query File (OPNQRYF) command.

Fields that contain any of the large object data types (BLOB, CLOB, or DBCLOB) can only be read by using the Copy From Query File (CPYFRMQRYF) command or SQL. The CPYFRMQRYF command must be used to access large object fields from an open query file. A field of data type BLOB, CLOB, or DBCLOB cannot be specified on these OPNQRYF parameters: KEYFLD, UNIQUEKEY, JFLD, and GRPFLD.

DATALINK considerations:

Fields of type DATALINK might not appear in selection, grouping, ordering, or join operations. If a DATALINK field appears in that format, it will be returned in its unprocessed form as it exists in the data space.

DDM file considerations:

The Open Query File (OPNQRYF) command can process distributed data management (DDM) files. However, there are some restrictions.

All DDM files identified on the FILE parameter must exist on the same IBM System i or System/38 target system. The Open Query File (OPNQRYF) command that 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 System i platforms).

Input and output considerations:

Here are the input and output considerations for using the OPTION parameter of the Open Query File (OPNQRYF) command and for field usage in an open query file.

Using the Open Query File (OPNQRYF) command for more than just input:

You can use the OPTION parameter of the Open Query File (OPNQRYF) command to specify the type of processing for your query file.

The default is OPTION(*INP); that is, the file is opened for input only. You can also use other OPTION values of 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 parameter, 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, distinct processing, or a user-defined table function is specified in the definition of the view. If the query optimizer needs to create a temporary file to implement the query, then the use of the file is restricted to input only.

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 that 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 that 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.

Field usage attribute:

Here are the rules that govern the field usage attribute (input-only or both input and output) in an open query file.

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 Open Query File (OPNQRYF) command. 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 later in this topic. 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 conditions are true:

- Input-only is not required because of any of the conditions previously described in this topic.
- 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 files, members, 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.

Open data path considerations:

An *open data path (ODP)* is a control block created when a file is opened. An ODP contains information about the merged file attributes and information returned by input or output operations.

The file, library, and file member names used by ODP are the same as the first file and file member names specified on the FILE parameter, unless an override forces the use of a different file or file member name. The record format name of the open query file is the same as that specified on the FORMAT parameter.

The Open Query File (OPNQRYF) command always opens a file with an ODP that is shared, as if SHARE(*YES) were specified for the file. If the file, library, or file member name specified in the high-level language (HLL) program differs from the name of the open query file, an override command must be used to specify the correct file, library, and member names to allow the HLL program to share the open query file ODP. If the first, or the only, member queried has an attribute of SHARE(*NO), SHARE(*YES) must be specified in an override to enable an HLL program to share the query file ODP.

If the OPNQRYF command is scoped to the job, any subsequent open, other than a query open, of the same file can share the ODP whether scoped to an activation group or the job. If the OPNQRYF command is scoped to an activation group, any subsequent open, other than a query open, of the same file can share the ODP if it is also scoped to the same activation group.

Files shared in a job:

To use the open data path that is 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, it does a full open of the file that 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), 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.
- Your program must be running in the same activation group to which the query open data path (ODP) is scoped. If the query ODP is scoped to the job, your program can run in any activation group within the job.

The OPNQRYF command never shares an existing open data path in the job or activation group. A request to open a query file fails with an error message if the open data path has the same library, file, and member name that is in the open request, and if either of the following is true:
- OPNSCOPE(*ACTGRPDFN) or OPNSCOPE(*ACTGRP) is specified for the OPNQRYF command, and the open data path is scoped to the same activation group or job from which the OPNQRYF command is run.
- OPNSCOPE(*JOB) is specified for the OPNQRYF command, and the open data path is scoped to the same job from which the OPNQRYF command is run.

Subsequent shared opens adhere to the same open options (such as SEQONLY) that were in effect when the OPNQRYF command was run.

Related concepts

"Sharing database files in the same job or activation group" on page 109
By default, the database management system allows one file to be read and changed by many users at the same time. You can also share a file in the same job or activation group by specifying the SHARE parameter.

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).

Copying from an open query file:

You can use the Copy from Query File (CPYFRMQRYF) command to copy from an open query file to another file or to print a formatted listing of the records.

Any open query file, except those using distributed data management (DDM) files, specified with the input, update, or all operation value on the FILE parameter of the Open Query File (OPNQRYF) command can be copied using the CPYFRMQRYF command. The CPYFRMQRYF command cannot be used to copy to logical files.

Although the CPYFRMQRYF command uses the open data path of the open query file, it does not open the file. Consequently, you do not have to specify SHARE(*YES) for the database file you are copying.

Related concepts

Database file management

Example 1: Copying from an open query file:

This example shows how to build a file with a subset of records using the Open Query File (OPNQRYF) and Copy from Query File (CPYFRMQRYF) commands.

Assume that you want to create a file from the CUSTOMER/ADDRESS file that contains only records where the value of the STATE field is Texas. You can specify as follows:

```
OPNQRYF FILE(CUSTOMER/ADDRESS) QRYSLT('STATE *EQ "TEXAS"')
CPYFRMQRYF FROMOPNID(ADDRESS) TOFILE(TEXAS/ADDRESS) CRTFILE(*YES)
```

Example 2: Copying from an open query file:

This example shows how to print records based on selection using the Open Query File (OPNQRYF) and Copy from Query File (CPYFRMQRYF) commands.

Assume that you want to print all records from FILEA where the value of the CITY field is Chicago. You can specify as follows:

```
OPNQRYF FILE(FILEA) QRYSLT('CITY *EQ "CHICAGO"')
CPYFRMQRYF FROMOPNID(FILEA) TOFILE(*PRINT)
```

Example 3: Copying from an open query file:

This example shows how to copy a subset of records to a diskette using the Open Query File (OPNQRYF) and Copy from Query File (CPYFRMQRYF) commands.

Assume that you want to copy all records from FILEB where the value of FIELDB is 10 to a diskette. You can specify:

```
OPNQRYF FILE(FILEB) QRYSLT('FIELDB *EQ "10"') OPNID(MYID)
CPYFRMQRYF FROMOPNID(MYID) TOFILE(DISK1)
```

Example 4: Copying from an open query file:

This example shows how to create a copy of the output of a dynamic join using the Open Query File (OPNQRYF) and Copy from Query File (CPYFRMQRYF) commands.

Assume that you want to create a physical file that has the format and data of the join of FILEA and FILEB, and that 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. To join the files and save a copy of the results in a new physical file MYLIB/FILEC, you can specify:

```
OPNQRYF FILE(FILEA FILEB) FORMAT(JOINAB) +
  JFLD((FILEA/CUST FILEB/CUST)) +
  MAPFLD((CUST 'FILEA/CUST')) OPNID(QRYFILE)
CPYFRMQRYF  FROMOPNID(QRYFILE) TOFILE(MYLIB/FILEC) CRTFILE(*YES)
```

The file MYLIB/FILEC will be created by the CPYFRMQRYF command. The file will have file attributes like those of FILEA although some file attributes might be changed. The format of the file will be like JOINAB. The file will contain the data from the join of FILEA and FILEB using the Cust field. File FILEC in library MYLIB can be processed like any other physical file with control language (CL) commands, such as the Display Physical File Member (DSPPFM) command and utilities, such as Query.

> **Related concepts**
> Database file management

Override considerations:

Overrides can change the name of a file, library, and member that should be processed by an open query file. However, any parameters other than TOFILE, MBR, LVLCHK, INHWRT, or SEQONLY that are specified on an Override with Database File (OVRDBF) command are ignored by the Open Query File (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.

Program considerations:

You need to consider these rules and techniques when running the Open Query File (OPNQRYF) command from a command line or when writing CL or other high-level language programs that use the OPNQRYF command.

- If you run the OPNQRYF command from a command entry line with the OPNSCOPE(*ACTGRPDFN) or TYPE(*NORMAL) parameter option, error messages that occur after the OPNQRYF command successfully runs will not close the file. Such messages would have closed the file prior to Version 2 Release 3 when TYPE(*NORMAL) was used. The system automatically runs the Reclaim Resources (RCLRSC) command if an error message occurs, except for message CPF0001, which is sent when the system detects an error in the command. However, the RCLRSC command only closes files opened from the default activation group at a higher level in the call stack than the level at which the RCLRSC command was run.

- After running a program that uses the OPNQRYF 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.

Considerations for writing a high-level language program:

Here are the considerations for writing a high-level language program that processes a database file using the Open Query File (OPNQRYF) command.

If you omit the FORMAT parameter, your high-level language program is coded as if you were 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 contained 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 were 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.

> **Related tasks**
>
> "Creating an open query file using an existing record format" on page 124
> The Open Query File (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.

CL program coding with the Open Query File (OPNQRYF) command:

When you use the Open Query File (OPNQRYF) command, follow these rules to prevent coding errors.
- Specify selection fields from a database file without an ampersand (&). Fields declared in the control language (CL) program with DCL or DCLF require the ampersand.
- Enclose fields defined in the CL program with DCL or DCLF within single quotation marks ('&testfld', for example).
- Enclose all parameter comparisons within quotation marks when compared to character fields, single quotation marks when compared to numeric fields.

In the following example, the fields INVCUS and INVPRD are defined as character data:

```
QRYSLT('INVCUS *EQ "' *CAT &K1CUST *CAT '" *AND +
        INVPRD *GE "' *CAT &LPRD *CAT '" *AND +
        INVPRD *LE "' *CAT &HPRD *CAT '"')
```

If the fields are defined numeric data, the QRYSLT parameter can look like this:

```
QRYSLT('INVCUS *EQ ' *CAT &K1CUST *CAT ' *AND +
        INVPRD *GE ' *CAT &LPRD *CAT ' *AND +
        INVPRD *LE ' *CAT &HPRD *CAT ' ')
```

Basic database file operations in programs

Your program can perform read, update, write, and delete operations on database files.

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 then sets the file position to the record just read, so that another read operation requesting the next sequential record can 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 file positioning operation of a high-level language (for example, SETLL in the RPG language or START in the COBOL language). A program can also change the file position by using the Position Database File (POSDBF) command.

Note: File positioning by means of the OVRDBF command does not occur until the next time the file is opened. Because a file can be opened only once within a control language (CL) program, this command cannot be used within a single CL program to affect what is read through the Receive File (RCVF) command.

At end of file, after the last read, the file member is positioned to *START or *END file position, depending on whether the program was reading forward or backward through the file. The following diagram shows *START and *END file positions. *START is at the top, with three records following, and *END at the bottom.
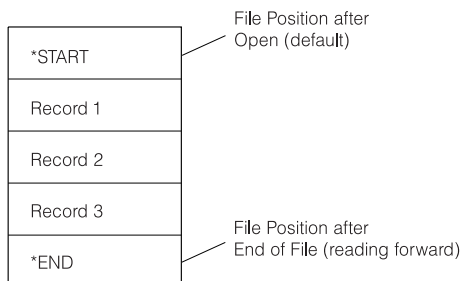
RBAFO540-0

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 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.

You can also use the 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.

> **Related concepts**
>
> Control language
>
> "Waiting for more records when end of file is reached" on page 196
> 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.
>
> **Related reference**
>
> Override with Database File (OVRDBF) command

Reading database records

The system provides various ways to read database records.

Note: Some high-level languages do not support all of the read operations available on the system. See your high-level language topic collection for more information about reading database records.

Reading database records using an arrival sequence access path:

These read operations are allowed if the file is defined with an arrival sequence access path, or if the file is defined with a keyed sequence access path, but you choose to ignore the keyed sequence access path.

Note: The system performs the read operations based on the operations that you specify using your high-level language. Your high-level language might not allow all of the following read operations. See your high-level language topic collection to determine which operations are allowed.

> **Related concepts**
>
> "Ignoring the keyed sequence access path" on page 102
> If a key field is defined for a database file, the system automatically uses the keyed sequence access path. However, sometimes you can use the ACCPTH parameter to ignore the keyed sequence access path for better performance.

Reading next operation:

This operation positions the file to the next record that is not deleted in the arrival sequence access path and gets that record.

Deleted records between the current position in the file and the next active record are skipped. (The READ statement in the RPG language and the READ NEXT statement in the COBOL language are examples of this operation.)

Reading previous operation:

This operation positions the file to the previous active record in the arrival sequence access path and gets that record.

Deleted records between the current file position and the previous active record are skipped. (The READP statement in the RPG language and the READ PRIOR statement in the COBOL language are examples of this operation.)

Reading first operation:

This operation positions the file to the first active record in the arrival sequence access path and gets that record.

Reading last operation:

This operation positions the file to the last active record in the arrival sequence access path and gets that record.

Reading same operation:

This operation gets the record that is identified by the current position in the file. The file position is not changed.

Reading by relative record number operation:

This operation positions the file to the record that is identified by the relative record number in the arrival sequence access path and gets that record.

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 a specified number of records. (The CHAIN statement in the RPG language and the READ statement in the COBOL language are examples of this operation.) Special consideration should be given to creating or changing a file to reuse deleted records if the file is processed by relative record processing.

> **Related concepts**
>
> "Reusing deleted records" on page 101
> Sometimes you might want to reuse deleted records for your database files. In this case, you can use the REUSEDLT parameter.

Reading database records using a keyed sequence access path:

These read 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: The system performs the read operations based on the statements that you specify using your high-level language. Your high-level language might not allow all of the following operations. See your high-level language topic collection to determine which operations are allowed by the language.

Reading next operation:

This operation gets the next record on the keyed sequence access path.

If a record format name is specified, this operation gets the next record in the keyed sequence access path that matches the record format. The current position in the file is used to locate the next record. (The READ statement in the RPG language and the READ NEXT statement in the COBOL language are examples of this operation.)

Reading previous operation:

This operation gets the previous record on the keyed sequence access path.

If a record format name is specified, this operation gets the previous record in the keyed sequence access path that matches the record format. The current position in the file is used to locate the previous record. (The READP statement in the RPG language and the READ PRIOR statement in the COBOL language are examples of this operation.)

Reading first operation:

This operation gets the first record on the keyed sequence access path.

If a record format name is specified, this operation gets the first record on the access path with the specified format name.

Reading last operation:

This operation gets the last record on the keyed sequence access path.

If a record format name is specified, this operation gets the last record on the access path with the specified format name.

Reading same operation:

This operation gets the record that is identified by the current file position. The position in the file is not changed.

Reading by key operation:

This operation 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 sequence 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 the RPG language and the READ statement in the COBOL language are examples of this operation.)

Reading by relative record number operation:

For a keyed sequence 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 sequence 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 sequence 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 sequence 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 the RPG language and the READ statement in the COBOL language are examples of this operation.)

Reading when logical file shares an access path with more keys operation:

When the First-In First-Out (FIFO), Last-In First-Out (LIFO), or First-Changed First-Out (FCFO) keyword is not specified in the data description specifications (DDS) for a logical file, the logical file can implicitly share an access path that has more keys than the logical file being created.

This sharing of a partial set of keys from an existing access path can lead to perceived problems for database read operations that use these partially shared keyed sequence access paths. The problems might appear to be:

- Records that should be read, are never returned to your program
- Records are returned to your program multiple times

What is actually happening is that your program or another currently active program is updating the physical file fields that are keys within the partially shared keyed sequence access path, but that are not actual keys for the logical file that is being used by your program (the fields being updated are beyond the number of keys known to the logical file being used by your program). The updating of the actual key fields for a logical file by your program or another program has always yielded the above results. The difference with partially shared keyed sequence access paths is that the updating of the physical file fields that are keys beyond the number of keys known to the logical file can cause the same consequences.

If these consequences caused by partially shared keyed sequence access paths are not acceptable, the FIFO, LIFO, or FCFO keyword can be added to the DDS for the logical file, and the logical file created again.

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, after 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, after 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 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 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 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 example shows an end-of-file delay operation:

```
User Program
  ┌──────────────┐
  │ OVRDBF       │
  │ FILEA        │
  │ EOFDLY (60)  │
  └──────┬───────┘
         │
  ┌──────▼───────┐
  │ Read         │
  │ FILEA        │
  └──────┬───────┘
         │
    No  ╱ EOF ╲  Yes
       ╲      ╱
  ┌─────────┐  ┌────────────┐
  │ Process │  │ EOF        │
  │         │  │ Processing │
  └─────────┘  └────────────┘

Data Management
  ┌──────────────┐
  │ Data         │
  │ Management   │
  │ Read to      │
  │ FILEA        │
  └──────┬───────┘
    No  ╱ EOF ╲  Yes
       ╲      ╱
  ┌───────────┐
  │ Return    │
  │ Record    │
  │ to Program│
  └───────────┘
         No  ╱ EOFDLY ╲  Yes
            ╲         ╱
  ┌───────────────┐   ╭────────╮
  │ Return EOF    │   │ Wait   │
  │ Condition to  │   │ EOFDLY │
  │ Program       │   │ Time   │
  └───────────────┘   ╰────────╯
                      RBAFO541-0
```
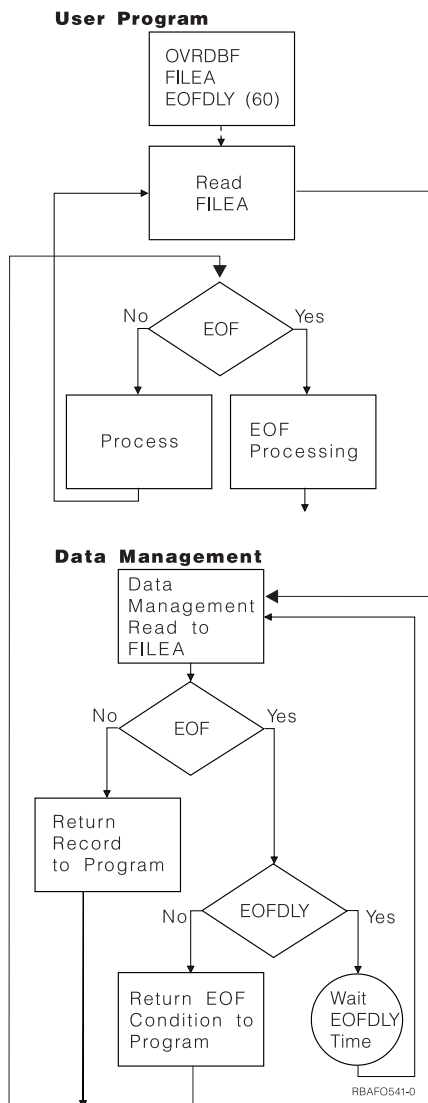
The actual processing of the EOFDLY parameter is more complex than shown because it is possible to force a true end-of-file if OPTION(*CNTRLD) on the ENDJOB command is used with a long delay time.

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.

Related concepts

Work management

Releasing locked records:

The system automatically releases a locked record when the record is updated or deleted, or when you read another record in the file. However, you might 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 topic collection for more information about releasing record locks.

Note: The rules for locking are different if your job is running under commitment control.

> **Related concepts**
> 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 the RPG language and the REWRITE statement in the COBOL language are examples of this operation. Before you update a database record, you must first read and lock the record. You lock the record by specifying the update option on any of the read operations listed under "Reading database records using an arrival sequence access path" on page 193 or "Reading database records using a keyed sequence access path" on page 194.

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 and if your job is running under a commitment control level of *ALL or *CS (cursor stability), then you must wait until the record is released or the time specified by the WAITRCD parameter 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. If the job that is reading records is not running under a commitment control level of *ALL or *CS, the job is able to read a record that is locked for update.

If the file you are updating has an update trigger associated with it, the trigger program is called before or after updating the record.

If the files being updated are associated with referential constraints, the update operation can be affected.

> **Related concepts**
> Commitment control
>
> "Triggering automatic events in your database" on page 267
> A *trigger* is a set of actions that run automatically when a specified change or read operation is performed on a specified database file. You can define a set of trigger actions in any high-level language that is supported on the i5/OS operating system.
>
> "Ensuring data integrity with referential constraints" on page 255
> You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

Adding database records

The write operation allows you to add a record to a physical file member.

The WRITE statement in the RPG language and the WRITE statement in the COBOL language 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. If a multiple-format logical file is used, 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 when the file organization is defined as RELATIVE). For more information about writing records over deleted record positions, see your high-level language topic collection.

If the physical file to which records are added reuses deleted records, the system tries to insert the records into slots that held deleted records. Before you create or change a file to reuse deleted records, you should review the restrictions and tips for use to determine whether the file is a candidate for reuse of deleted record space.

If you are adding new records to a file member that has a keyed access path, the new record appears in the keyed sequence 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.

If the file to which you are adding a record has an insert trigger associated with it, the trigger program is called before or after inserting the record.

If the files you are adding to are associated with referential constraints, record insertion can be affected.

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.

> **Related concepts**
>
> "Reusing deleted records" on page 101
> Sometimes you might want to reuse deleted records for your database files. In this case, you can use the REUSEDLT parameter.
>
> "Triggering automatic events in your database" on page 267
> A *trigger* is a set of actions that run automatically when a specified change or read operation is performed on a specified database file. You can define a set of trigger actions in any high-level language that is supported on the i5/OS operating system.
>
> "Ensuring data integrity with referential constraints" on page 255
> You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

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 control language (CL) program or a high-level language program. It must be used if all of the following conditions 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.

RBAFO542-0
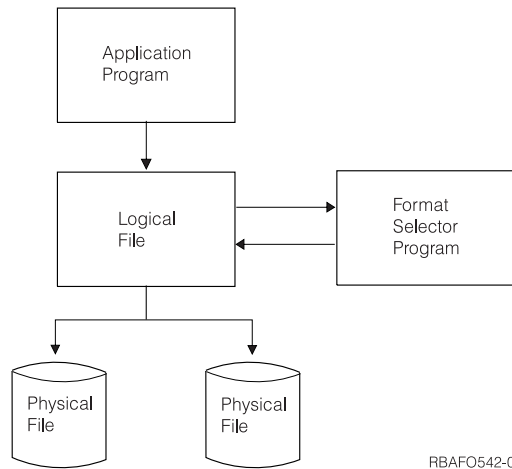
The following example shows the programming statements for a format selector program written in the RPG/400 language:

```
CL0N01N02N03Factor1+++OpcdeFactor2+++ResultLenDHHiLoEqComments+++...
+++*
C           *ENTRY    PLIST
C                     PARM           RECORD 80
C* The length of field RECORD must equal the length of
C* the longest record expected.
C                     PARM           FORMAT 10
C                     MOVELRECORD    BYTE    1
C          BYTE       IFEQ 'A'
C                     MOVEL'HDR'     FORMAT
C                     ELSE
C                     MOVEL'DTL'     FORMAT
C                     END
```

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 update operations only. For update operations, 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.

Related concepts

"Controlling how records are added to a logical file with multiple formats" on page 43
To add a record to a multiple-format logical file, you need to identify the member of the based-on physical file to which you want the record to be written.

Related tasks

"Creating a logical file" on page 39
You can create a logical file using data description specifications (DDS).

Using the force-end-of-data operation:

The force-end-of-data (FEOD) operation allows you to force all changes that were made to a file by your program to auxiliary storage. It also allows you to position the read operation to either the beginning or the end of a file if the file is open for input operations.

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.

*START sets the beginning or starting position in the database file member currently open to just before the first record in the member (the first sequential read operation reads the first record in the current member). If MBR(*ALL) processing is in effect for the Override with Database File (OVRDBF) command, a read previous operation gets the last record in the previous member. If a read previous operation is done and the previous member does not exist, the end of file message (CPF5001) is sent. *END sets the position in the database file member currently open to just after the last record in the member (a read previous operation reads the last record in the current member). If MBR(*ALL) processing is in effect for the OVRDBF command, a read next operation gets the first record in the next member. If a read next operation is done and the next member does not exist, the end of file message (CPF5001) occurs.

If the file has a delete trigger, the force-end-of-data operation is not allowed. If the file is part of a referential parent relationship, the FEOD operation is not allowed.

See your high-level language topic collection for more information about the FEOD operation (some high-level languages do not support the FEOD operation).

Related concepts

"Triggering automatic events in your database" on page 267
A *trigger* is a set of actions that run automatically when a specified change or read operation is performed on a specified database file. You can define a set of trigger actions in any high-level language that is supported on the i5/OS operating system.

"Ensuring data integrity with referential constraints" on page 255
You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

Deleting database records

The delete operation allows you to delete an existing database record.

The DELET statement in the RPG language and the DELETE statement in the COBOL language are examples of this operation. To delete a database record, you must first read and lock the record. You lock the record by specifying the update option on any of the read operations listed under "Reading database records using an arrival sequence access path" on page 193 or "Reading database records using a keyed sequence access path" on page 194. 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 specify which record you want deleted on the delete statement. For example, the RPG language 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 sequence 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.
- 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.

Note: The system tries to reuse deleted record space automatically if the file has the reuse deleted record space attribute specified.

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 topic collection 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 Initialize Physical File Member (INZPFM) command. The INZPFM command can initialize the entire physical file member to deleted records.

If the file from which you are deleting has a delete trigger associated with it, the trigger program is called before or after deleting the record.

If the file is part of a referential constraint relationship, record deletion might be affected.

Related concepts

"Reorganizing a physical file member" on page 210
You can reorganize a physical file member to change the manner in which records are stored on the i5/OS operating system.

"Reusing deleted records" on page 101
Sometimes you might want to reuse deleted records for your database files. In this case, you can use the REUSEDLT parameter.

"Initializing data in a physical file member" on page 209
To use relative record processing in a program, a 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 you initialize these records.

"Triggering automatic events in your database" on page 267
A *trigger* is a set of actions that run automatically when a specified change or read operation is performed on a specified database file. You can define a set of trigger actions in any high-level language that is supported on the i5/OS operating system.

"Ensuring data integrity with referential constraints" on page 255
You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

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 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)

To close a database file in a program, use one of the following methods:
- High-level language close statements

  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 topic collection.
- Close File (CLOF) command

  You can use the CLOF command to close database files that were opened using either the Open Database File (OPNDBF) or Open Query File (OPNQRYF) command.
- 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 ODP for that file. You can use the RCLRSC command to allow a calling program to close the files of a called program. (For example, if the called program returns to the calling program without closing its files, the calling program can then close the files of the called program.) However, the normal way of closing files in a program is with the high-level language close operation or through the CLOF command. For more information about resource reclamation in the integrated language environment, see the ILE Concepts book.

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.

When a process is trying to lock a file that is held by another process, the Close database file exit program is called. This exit is called in the process that is holding the lock.

Related concepts

"Sharing database files in the same job or activation group" on page 109
By default, the database management system allows one file to be read and changed by many users at
the same time. You can also share a file in the same job or activation group by specifying the SHARE
parameter.
Control language

ILE Concepts PDF

Related reference

Close File (CLOF) command

Reclaim Resources (RCLRSC) command

Monitoring database file errors in a program

When your applications perform operations on database files, you need to monitor messages about file
errors so as to take appropriate actions to prevent the errors.

Each high-level language (HLL) provides its own procedure for monitoring these messages, and you
should see the documentation for the HLL you are using to implement error message monitoring.

One or more of the following events occur when error conditions are detected during processing of a
database file:
- Messages can be sent to the program message queue for the program processing the file.
- An inquiry message can be sent to the system operator message queue.
- File errors and diagnostic information can appear to your program as return codes and status
  information in the file feedback area.

For example, the COBOL language sets a return code in the file status field if it is defined in the
program.

System handling of error messages

If you do not monitor messages about database file errors, the system handles the errors.

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 actions.

Effect of error messages on file positioning

If a message is sent to your program when your program is processing a database file member, the
position in the file is not lost.

The position 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.

Determining which messages you want to monitor

If your programming language allows you to monitor error messages, you can choose which messages
you want to monitor.

Here are some of the error messages that you can monitor.

Table 46. Sample of monitorable error messages

Message identifier	Description
CPF5001	End of file reached

Table 46. Sample of monitorable error messages (continued)

Message identifier	Description
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
CPF502B	Error in trigger program
CPF502D	Referential constraint violation
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
CPF503A	Referential constraint violation
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	Unique access path problem prevents access to member
CPF5097	Key mapping error

You can display the full description of these messages using the Display Message Description (DSPMSGD) command.

> **Related concepts**
>
> Monitoring for messages in a CL program or procedure
>
> Control language
>
> **Related reference**
>
> Display Message Description (DSPMSGD) command

Managing database files

You can manage and maintain control over database files in various ways.

Basic operations for managing database files

The basic operations for managing database files are copying a file and moving a file.

Copying a file

You can copy a file using System i Navigator or the Copy File (CPYF) command.

Copying a file using System i Navigator

Copying a file (table) to a different library (schema) creates two instances of the same file. To copy a file to a different library, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the table that you want to copy.
4. Click **Tables**.

5. Right-click the table that you want to copy, and click **Copy**.
6. Right-click the schema to which you want to copy the table, and click **Paste**.

Distributed data management (DDM) is used by System i Navigator to actually copy the table. The copy operation is performed by using DDM over TCP/IP. TCP/IP must be enabled between the source and target systems in a way as they are known to System i Access.

Copying a file using the Copy File (CPYF) command

The Copy File (CPYF) command copies all or part of a database or an external device file to a database or an external device file.

> **Related concepts**
> Control language
> **Related reference**
> Copy File (CPYF) command

Moving a file

You can move a database file from one library to another using System i Navigator or the Move Object (MOVOBJ) command.

Moving a file using System i Navigator

To move a file (table) to a different library (schema), follow these steps:
1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the table that you want to move.
4. Click **Tables**.
5. Right-click the table that you want to move, and click **Cut**.
6. Right-click the schema to which you want to move the table, and click **Paste**. Or drag the table to another schema on the same system or a different system.

Note: Moving a table to a new location does not always remove it from the source system. For example, if you have read authority but not delete authority to the source table, the table is moved to the target system; however, it is not deleted from the source system, causing two instances of the table to exist.

Distributed data management (DDM) is used by System i Navigator to actually move the table. The move operation is performed by using DDM over TCP/IP. TCP/IP must be enabled between the source and target systems in a way as they are known to System i Access.

Moving a file using the Move Object (MOVOBJ) command

The Move Object (MOVOBJ) command removes an object from its currently assigned library and places it in a different library. The type of the object moved is specified in the OBJTYPE parameter.

> **Related concepts**
> Distributed database programming
> Control language
> **Related reference**
> Move Object (MOVOBJ) command

Managing database members

You can perform various operations on database file members, such as adding and removing members and changing member attributes.

Before you perform any input or output operations on a file, the file must have at least one member. As a general rule, a database file has 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 a database file assume that the member being used is the first member in the file, and because most files only have one member, usually you do not 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 allows you to make changes to file definitions. You can use control language (CL) commands to perform most of these operations.

> **Related concepts**
>
> Control language

Adding members:

You can use several methods to add members to a database file.

- Automatically. When a file is created using the Create Physical File (CRTPF) or Create Logical File (CRTLF) command, 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) command.
- The Copy File (CPYF) command. If the member you are copying does not exist in the file being copied to, the member is added to the file by the CPYF command.

> **Related reference**
>
> Create Physical File (CRTPF) command
>
> Create Logical File (CRTLF) command
>
> Add Physical File Member (ADDPFM) command
>
> Copy File (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: SRCTYPE (the member's source type), 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 can use the SIZE parameter on the CHGPF command.

Related reference

Change Physical File Member (CHGPFM) command

Change Logical File Member (CHGLFM) command

Change Physical File (CHGPF) command

Change Logical File (CHGLF) command

Renaming members:

You can use the Rename Member (RNMM) command to change the name of an existing member in a physical or logical file; however, the file name is not changed.

Related reference

Rename Member (RNMM) command

Removing members:

You can use the Remove Member (RMVM) command to remove a member and its contents from a database file.

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.

Related reference

Remove Member (RMVM) command

Physical file member operations

Some member operations are unique to physical file members.

If the physical file members are associated with referential constraints, the operations on these members might be affected.

Related concepts

"Ensuring data integrity with referential constraints" on page 255

You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

Initializing data in a physical file member:

To use relative record processing in a program, a 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 you initialize these records.

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 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.

Variable-length character fields have a zero-length default value. The default value for null-capable fields is the null value. The default value for dates, times, and timestamps is the current date, time, or timestamp if no default value is defined. Program-described files have a default value of all blanks.

Note: You can initialize one 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 are 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.

Related concepts

"Deleting database records" on page 203
The delete operation allows you to delete an existing database record.

Related reference

Initialize Physical File Member (INZPFM) command

Clearing data from a physical file member:

You can remove data from a physical file member using the Clear Physical File Member (CLRPFM) command. After the clear operation is complete, the member description remains, but the data is removed.

Related reference

Clear Physical File Member (CLRPFM) command

Reorganizing a physical file member:

You can reorganize a physical file member to change the manner in which records are stored on the i5/OS operating system.

Reorganizing a table using System i Navigator:

Reorganizing a table restores the table to its ideal physical organization. That is, the rows of the table are laid out on pages and ordered by their key values in some frequently used indexes. You can reorganize a table by compressing out deleted records, by key, or by a selected index from System i Navigator.

To reorganize a table using System i Navigator, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the table that you want to reorganize.
4. Click **Tables**.
5. Right-click the table that you want to reorganize, and click **Data** → **Reorganize**.
6. To specify how to reorganize the rows in the table, select one of the following options.

- By compressing out deleted rows without preserving the arrival row sequence: specify that valid rows at the end of the table are moved to deleted rows until no deleted rows remain.
- By compressing out deleted rows and preserving the arrival row sequence: specify that all valid rows after the first deleted row in the table are moved forward in the table to compress out any deleted rows.
- By table key: specify that the rows of the table are rearranged by the key values of the table's access path. The table must have a primary key or must be a keyed physical file.
- By a selected index from library: specify that the rows of the table are rearranged by the key values of an index or keyed logical file that is built over the specified table. You can select only an existing index. Your list of indexes is determined by the library you select.

7. Optional: To control the performance and concurrency of the reorganize operation, select any of the following options:

- Specify which partition of a partitioned file (or which member of a multiple member physical file) should be reorganized.
- Specify whether the reorganize operation can be suspended and subsequently restarted.

  If you do not specify that the reorganize operation can be suspended, the table will be allocated exclusively for the duration of the reorganize operation and can only be suspended by ending the job immediately.

  If you specify that the reorganize can be suspended:

  – The file must be journaled since the rows are moved under commitment control to ensure that no rows are lost if the reorganize operation is suspended.

  – You can also specify whether other users can read the table or change the table during the reorganize operation. Locks are acquired for short periods of time on rows that are moved during the reorganize. If concurrent jobs also acquire locks on rows, record lock time-outs might occur. You can change the record lock wait time for the file, or you can use the Override with Database File (OVRDBF) command to specify an appropriate record wait time.

- Specify how indexes are maintained:

  – If you specify that the reorganize operation can be suspended, you can also specify that all indexes should be maintained during the reorganize operation. No index rebuilds are necessary.

  – Otherwise, you can specify that indexes be rebuilt synchronously or asynchronously. You can see the progress of asynchronously built indexes by using the Edit Rebuild of Access Paths (EDTRBDAP) command.

Related concepts

"Locking records" on page 105
DB2 for i5/OS has built-in integrity for records.

Reorganizing a physical file member using the Reorganize Physical File Member (RGZPFM) command:

You can specify several key options when you reorganize a physical file member using the Reorganize Physical File Member (RGZPFM) command.

You can use the RGZPFM command to:

- Remove deleted records to make the space occupied by them available.
- Reorganize the records of a physical file member in the order in which you normally access them sequentially, thereby minimizing the time required to retrieve records. You can do this by using the KEYFILE parameter. This might be advantageous for members that are primarily accessed in an order other than arrival sequence. You can reorganize a physical file member using either of the following key fields:

  – Key fields of the physical file member

  – Key fields of a logical file member that is based on the physical file

Note: Key fields are defined at the file level.

- Reorganize a source file member, insert new source sequence numbers, and reset the source date fields (using the SRCOPT and SRCSEQ parameters on the RGZPFM command).
- If you specify that the reorganize operation cannot be canceled, reclaim space in the variable portion of the member that was previously used by variable-length fields in the physical file format and that has become fragmented.

  Related reference

  Reorganize Physical File Member (RGZPFM) command

  "Usage notes: Reorganizing a physical file member"
  Here is a list of considerations for reorganizing a physical file member.

Example: Reorganizing a physical file member:

The example shows how to reorganize a physical file member with the Reorganize Physical File Member (RGZPFM) command.

For example, the following RGZPFM command reorganizes the first member of a physical file using an access path of a logical file member:

```
RGZPFM  FILE(DSTPRODLB/ORDHDRP)
        KEYFILE(DSTPRODLB/ORDFILL   ORDFILL)
```

Physical file member ORDHDRP has an arrival sequence access path. You reorganize it using the access path of logical file member ORDFILL. Assume that the key field is the *Order* field. The following tables show how records are arranged.

This table shows the original physical file member ORDHDRP. Record 3 is deleted before the RGZPFM command is run.

Relative record number	Cust	Order	Ordate. . .
1	41394	41882	072480. . .
2	28674	32133	060280. . .
3	deleted	record	
4	56325	38694	062780. . .

This table shows the physical file member ORDHDRP after you reorganize it 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. . .

Usage notes: Reorganizing a physical file member:

Here is a list of considerations for reorganizing a physical file member.

- If a physical file member has an arrival sequence access path and you reorganize it using a keyed sequence access path, the arrival sequence access path is changed. That is, the records in the member are physically placed in the order of the keyed sequence access path. 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.
- Reorganizing a physical file member compresses deleted records, which changes subsequent relative record numbers.

- Because access paths with the following DDS keywords specified depend on the physical sequence of records in the physical file member, the sequence of the records with duplicate key fields might change after you reorganize a physical file member using a keyed sequence access path:
  - First-Changed First-Out (FCFO)
  - First-In First-Out (FIFO)
  - Last-In First-Out (LIFO)

  The sequence of the records with duplicate key fields are maintained only for the access path specified in the KEYFILE parameter. If the access path specified in the KEYFILE parameter has an LIFO DDS keyword, the duplicate key fields are maintained only if you specify that the reorganize operation can be canceled (suspended).
- If you specify that the reorganize operation cannot be canceled and you end the job running the Reorganize Physical File Member (RGZPFM) command, all the access paths over the physical file member might have to be rebuilt. If you specify that the reorganize operation can be canceled, and you cancel the RGZPFM command, only those access paths that are not maintained during the reorganize operation might have to be rebuilt.
- If you use the RGZPFM command twice in a row, you might notice that the total size of the member after the first time differs from the total size after the second. This is because the amount of space allocated for the reorganized member is only an estimate that allows extra space for future insertion. After records are reorganized the first time, the space allocated is calculated exactly.

Reorganization options:

You can specify whether the reorganize operation on a physical file member can or cannot be canceled. Consider the features of each option when deciding which option to use.
- **ALWCANCEL(*NO)**: This is the traditional type of reorganize operation. A full copy of the data might be made, so you need up to two times the amount of space. This option cannot be canceled (suspended) and cannot fully run in parallel. It requires exclusive use of the physical file member.
- **ALWCANCEL(*YES)**: The data rows are moved within the physical file member so that a full copy of the data is not required. The physical file must be journaled, however, so storage is necessary for the journal entries. You can use the journal receiver threshold to minimize the amount of storage used in a specific journal receiver.

  This option can be canceled (suspended) and restarted. It can run in parallel if the DB2 Symmetric Multiprocessing option is installed. To control the amount of resources used by the reorganize operation, you might want to change the query attributes by using the Change Query Attributes (CHGQRYA) CL command or by using the Change Query Attributes function from System i Navigator.

  This option requires exclusive use for only a few seconds after the reorganize operation is complete to return storage to the system. If the exclusive lock cannot be acquired, a warning message is sent to the job log indicating that space cannot be recovered. To recover the space, you can start the reorganization again when no concurrent users are accessing the file member. The reorganize operation then immediately attempts to recover the space before starting the reorganization. If concurrent data changes have occurred since the initial reorganize operation, only a portion of the space might be recovered.

  If LOCK(*EXCLRD) or LOCK(*SHRUPD) is specified, the result of the reorganize operation is not guaranteed to be exact, because concurrent users might be locking rows or changing rows in the file member.

The type of reorganize operation that you decide to use depends on several factors. For example, is your goal to recover space, or is the sequence of the rows important? Is it important that the reorganize operation can be canceled (suspended)? Is it important to allow concurrent access to the physical file member? Use the following table to determine which option is most appropriate based on these factors. The shaded entries (which are also identified by an asterisk) are the characteristics of a key file option that make its choice particularly desirable.

	ALWCANCEL(*NO)		ALWCANCEL(*YES)		
	KEYFILE(*NONE)	KEYFILE(*FILE or keyfile)	KEYFILE (*RPLDLTRCD)	KEYFILE(*NONE)	KEYFILE(*FILE or keyfile)
Cancel and restart	No	No	Yes*	Yes*	Yes*
Concurrent access	No	No	Yes*	Yes*	Yes*
Parallel processing	Only index rebuilds	Only index rebuilds	Data movement and index rebuilds*	Data movement and index rebuilds*	Data movement and index rebuilds*
Non-parallel performance	Very fast	Fast	Very fast*	Slower*	Slowest*
Temporary storage	Double data storage	Double data storage	Journal receiver storage*	Journal receiver storage*	Journal receiver storage*
LIFO KEYFILE index processing	N/A	Duplicates reversed	N/A*	N/A*	Duplicate ordering preserved*
Index processing (non-KEYFILE)	Synchronous or asynchronous rebuilds*	Synchronous or asynchronous rebuilds*	Maintain indexes or synchronous or asynchronous rebuilds*	Maintain indexes or synchronous or asynchronous rebuilds*	Maintain indexes or synchronous or asynchronous rebuilds*
Final row position exact	Yes*	Yes*	Only if LOCK(*EXCL) and not restarted	Only if LOCK(*EXCL) and not restarted	Only if LOCK(*EXCL) and not restarted
Amount of CPU and I/O used	Smallest*	Next smallest*	Smallest	More	Most
Variable lengths segment reorganize	Good*	Good*	Worse	Worse	Worse
Allows referential integrity parents and FILE LINK CONTROL DataLinks	Yes*	Yes*	No	No	No
Allows QTEMP and database cross-reference files	Yes*	Yes*	No	No	No
Replication cost	Minimal (one journal entry)*	Minimal (one journal entry)*	More (journal entries for all rows moved)	Most (journal entries for all rows moved)	Most (journal entries for all rows moved)

Suspending or canceling a reorganize operation:

Sometimes you might need to suspend or cancel a reorganize operation on a physical file member.

If you specify that the reorganize operation cannot be canceled and one of the following situations occur while the Reorganize Physical File Member (RGZPFM) command is running, the records might not be reorganized at all:

- 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.

In addition, when the RGZPFM command is running, records associated with large objects (LOBs) might not be reorganized and a deleted record might remain in the member.

If you specify that the reorganize operation can be canceled and one of the preceding situations occurs, or you cancel the reorganize operation, the operation might be only partially completed. If you issue the

same reorganize operation later, the reorganize operation might continue from where it was interrupted. However, if significant changes have occurred since the reorganize operation was canceled, the reorganize operation will not be continued and will start over.

The status of the member being reorganized also depends on how much the system was able to do before the reorganization is ended and what you have specified in the SRCOPT parameter.

If the SRCOPT parameter was specified, any of the following statuses might happen to the member:
- It is completely reorganized. A completion message is sent to your job log indicating the reorganize operation is completely successful.
- It is not reorganized at all, or only partially reorganized. A message is sent to your job log indicating that the reorganize operation is not successful. If this occurs, run the Reorganize Physical File Member (RGZPFM) command again.
- It is reorganized, but only some of the sequence numbers are changed. A completion message is sent to your job log indicating that the member is reorganized, but all the sequence numbers are not changed. If this occurs, issue the RGZPFM command again with KEYFILE(*NONE) specified.

If the SRCOPT parameter was not specified, the member is either completely reorganized or not reorganized at all. You can display the contents of the member using either the Display Physical File Member (DSPPFM) command or the Quick View from System i Navigator to determine how much of the member, if any, has been reorganized. You can run the RGZPFM command again if necessary.

To reduce the number of deleted records that exist in a physical file member, you can create or change the file to reuse deleted record space.

> **Related concepts**
> "Reusing deleted records" on page 101
> Sometimes you might want to reuse deleted records for your database files. In this case, you can use the REUSEDLT parameter.

Displaying records in a physical file member:

You can display records in a physical file member using the Display Physical File Member (DSPPFM) command.

The DSPPFM command can be used to display the data in the physical file member by arrival sequence. The DSPPFM command can be used for:
- Problem analysis
- Debugging
- Record inquiry

You can display source files or data files, whether 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 language installed, you can use the Start SQL (STRSQL) command to interactively select and display records.

> **Related reference**
> Display Physical File Member (DSPPFM) command

Using database attribute and cross-reference information

The i5/OS integrated database provides file attribute and cross-reference information.

Some of the cross-reference information include:
- 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
- Constraints associated with a file
- Key fields for a file

Each of the commands described in these topics 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.

You can retrieve information about a member of a database file for use in your applications with the Retrieve Member Description (RTVMBRD) command.

> **Related concepts**
> Control language
>
> **Related reference**
> Retrieve Member Description (RTVMBRD) command

Displaying information about database files

By using System i Navigator or CL commands, you can display various types of information about database files.

Displaying attributes of a file using System i Navigator:

You can use System i Navigator to display the attributes of a database file or a device file.

To display database file attributes, follow these steps:
1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the table, view, or index for which you want to display information.
4. Click **Tables**, **Views**, or **Indexes**.
5. Right-click the table, view, or index for which you want to display information, and click **Description**.

Displaying attributes of a file using the Display File Description (DSPFD) command:

You can use the Display File Description (DSPFD) command to display the attributes of a database file. The information can be displayed, printed, or written to a database output file (OUTFILE).

The information supplied by the DSPFD 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)
- File constraints (*CST)
- Triggers (*TRG)

  Related reference

  Display File Description (DSPFD) command

Displaying the description 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).

  Related reference

  Display File Field Description (DSPFFD) command

Displaying the relationships between files on the system:

You can use the Display Database Relations (DSPDBR) command to display the relationships between database files on the system. The information can be displayed, printed, or written to a database output file (OUTFILE).

You can use the 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.
- A list of physical files that are dependent files in a referential constraint relationship with this file.

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 Control language (CL) topic 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 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 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
Constraint	C	The physical file is dependent on the data in another physical file with which it is associated via a constraint.
Data	D	The file or member is dependent on the data in a member of another file.
Access path sharing	I	The file member is sharing an access path.
Access path owner	O	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	V	The SQL view or member is dependent on another SQL view.

Related concepts

Control language

Related reference

Display Database Relations (DSPDBR) command

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 and 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 DSPPGMREF command.

The following table shows the objects for which the high-level languages and utilities save information.

Language or utility	Files	Programs	Data areas	See Notes
CL	Yes	Yes	Yes	1
COBOL/400	Yes	Yes	No	2
CSP	Yes	Yes	No	3
DFU	Yes	N/A	N/A	
FORTRAN/400	No	No	N/A	
ILE C/C++	No	No	N/A	
ILE COBOL	Yes	Yes	Yes	2
ILE RPG	Yes	Yes	Yes	
PL/I	Yes	Yes	N/A	2
RPG/400	Yes	Yes	Yes	4
SQL	Yes	N/A	N/A	

Language or utility	Files	Programs	Data areas	See Notes

Notes:

1. 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 control language (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 Control language (CL) topic.

2. 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).

3. CSP programs also save information for an object of type *MSGF, *CSPMAP, and *CSPTBL.

4. 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 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.

> **Related concepts**
> Control language
> **Related reference**
> Display Program Reference (DSPPGMREF) command

Displaying the system cross-reference files:

You can use the database files that are managed by the system to determine basic attribute and database file requirements. To display the fields in these files, use the Display File Field Description (DSPFFD) command.

The system manages eight database files that contain the following information:
- Basic database file attribute information (QSYS/QADBXREF)
- Cross-reference information (QSYS/QADBFDEP) about all the database files on the system (except those database files that are in the QTEMP library)
- Database file field information (QSYS/QADBIFLD)
- Database file key field information (QSYS/QADBKFLD)
- Referential constraint file information (QSYS/QADBFCST)
- Referential constraint field information (QSYS/QADBCCST)
- SQL package information (QSYS/QADBPKG)
- Remote database directory information (QSYS/QADBXRDBD)

Note: The authority to use these files is restricted to the security officer. However, all users have the authority to view the data by using one of (or the only) logical file(s) built over each file. The authorities for these files cannot be changed because they are always open.

> **Related reference**
> Display File Field Description (DSPFFD) command

Writing the output from a command directly to a database file

You can store the output from many control language (CL) commands in an output physical file by specifying the OUTFILE parameter on the commands. You can then 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 file. 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.

See the Control language (CL) topic for a list of commands that allow output files and the names of the model files supplied for those commands.

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.

> **Related concepts**
> Control language
> **Related reference**
> Display File Field Description (DSPFFD) command
> Display Journal (DSPJRN) command
> Display Problems (DSPPRB) command

Example: A command output file:

This example shows how to write the output from a control language (CL) command directly to a database file.

You can use 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 example shows the DDS for such a logical file. Records are selected based on the file name.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
    A* Logical file DBROUTL for query
    A
    A          R DBROUTL                  PFILE(DBROUT)
    A          S WHFNAM                   VALUES('ORDHDRL' 'ORDFILL')
    A
```

Output files for the Display File Description (DSPFD) command:

The Display File Description (DSPFD) command provides unique output files, depending on the parameters specified.

To collect access path information about all files in the LIBA library, you can 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 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 as follows:

```
DSPFFD   QSYS/QAFDACCP
```

> **Related concepts**
> Control language
> **Related reference**
> Display File Description (DSPFD) command

Output files for the Display Journal (DSPJRN) command:

The Display Journal (DSPJRN) command provides unique output files.

> **Related concepts**
> Control language
> **Related reference**
> Display Journal (DSPJRN) command

Output files for the Display Problems (DSPPRB) command:

The Display Problems (DSPPRB) command provides unique output files, depending on the type of record.

The output files depend on the following types of record:

- Basic problem data record (*BASIC). This includes problem type, status, machine type/model/serial number, product ID, contact information, and tracking data.
- Point of failure, isolation, or answer FRU records (*CAUSE). Answer FRUs are used if they are available. If answer FRUs are not available, isolation FRUs are used if they are available. If answer FRUs and isolation FRUs are not available, then point of failure FRUs are used.
- PTF fix records (*FIX).
- User-entered text (note records) (*USRTXT).
- Supporting data identifier records (*SPTDTA).

The records in all five output files have a problem identifier so that the cause, fix, user text information, and supporting data can be correlated with the basic problem data. Only one type of data can be written to a particular output file. The cause, fix, user text, and supporting data output files can have multiple records for a particular problem.

    Related concepts

    Control language

    Related reference

    Display Problems (DSPPRB) command

Changing database file descriptions and attributes

Sometimes you might change the descriptions or attributes of a database file by adding, changing, or deleting a field.

Effects of changing fields in a file description

The system uses the information in the record format description to determine the level identifier. Changes to the fields in a file description cause the level identifier to change. Changes in key fields or select/omit fields might cause unexpected results in programs using the new access path.

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, the number of decimal positions in the field, and whether the field allows the null value. 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
- CCSID 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 does not cause a level check, the change might 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 might 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 might affect your program. You can 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 source entry utility (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) command 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.

> **Related reference**
>
> Display File Field Description (DSPFFD) command
>
> Display File Description (DSPFD) command
>
> Change Physical File (CHGPF) command
>
> Change Logical File (CHGLF) command
>
> Override with Database File (OVRDBF) command

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. If the level identifier changes, you can either compile the program again or avoid recompiling using a logical file.

For example, the level identifier changes 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 program again that uses the physical file. After the program is recompiled, it uses 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. To avoid compiling your program again, follow these steps:

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 perform one of the following operations:

   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 "Database file processing: Runtime considerations" on page 99).

      `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 tables illustrate the relationship of the record formats used in the three files.

Table 47. FILEA (old physical file)

FLDA	FLDB	FLDC	FLDD

In FILEB, FLDB1 was added to the record format.

Table 48. FILEB (new physical file)

		FLDB1		

FILEC shares the record format of FILEA. FLDB1 is not used in the record format for the logical file.

Table 49. FILEC (logical file)

FLDA	FLDB	FLDC	FLDD

When you make changes to a physical file, which causes you to create the file again, all logical files referring to it must first be deleted before you can delete and create a new physical file. After the physical file is re-created, you can re-create or restore the logical files referring to it.

Example 1: Changing a physical file description and attributes:

This example shows how to 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 the 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 new logical file name should be the same as the old logical file name. You can use the FORMAT keyword to use the same record formats as in the current logical file if no changes need to be made to the record formats. You can also use the Create Duplicate Object (CRTDUPOBJ) command to create another logical file from the old logical file FILELFC 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: Changing a physical file description and attributes:

This example shows how to create new versions of files in the same library.

To create new versions of files in the same library, follow these steps:

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 old and new 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 assume that 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.

Related concepts

Control language

Changing a logical file description and attributes

As a general rule, when your changes to a logical file cause a change to the level identifier (for example, adding a new field, deleting a field, or changing the length of a field), you need to recompile the program that uses the logical file.

Sometimes you can make changes to a logical file, which changes the level identifier and does 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 are forced to turn off level checking to run your program that uses 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.

Related reference

Change Logical File (CHGLF) command

Create Logical File (CRTLF) command

Recovering and restoring your database

You can use several i5/OS save and restore functions to recover your database after the system loses data.

Related concepts

"Writing data and access paths to auxiliary storage" on page 104

Normally, DB2 for i5/OS determines when to write changed data and access paths from main storage to auxiliary storage. However, you can control when database changes are written to auxiliary storage.

Backup and recovery

Recovering data in a database file

The i5/OS operating system uses journaling and commitment control to help you recover data in a database file.

Managing journals:

Journal management allows you to record all the data changes that occur to one or more database files. You can then use the journal for recovery.

If a database file is destroyed or becomes unusable and you are using journals, you can reconstruct most of the activity for the file. The journal also allows you to remove revisions made to the file.

Related concepts

Journal management

Journals:

When a change is made to a file and you are using journals, the system records the change in a journal receiver and writes the receiver to auxiliary storage before the change is recorded in the file. Therefore, the journal receiver always has the latest database information.

Journal entries record activity for a specific record or for the file as a whole. Each entry includes bytes of control information that identify the source of the activity (such as user, job, program, time, and date). For changes that affect a single record, record images are included after the control information. The record image before the change can also be included. You can control whether to create a journal both before and after record images or just after record images by specifying the IMAGES parameter on the Start Journal Physical File (STRJRNPF) command.

All journaled database files are automatically synchronized with the journal when the system is started (IPL time). If the system session ends abnormally, some database changes might be in the journal, but some of these changes might not be reflected in the database files. If that is the case, the system automatically updates the database files from the journal.

Journals make saving database files an easier and faster task. For example, instead of saving an entire file every day, save the journal receiver that contains the changes to that file. You can 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.

> **Related concepts**
>
> Journal management

Working with journals:

You can work with journals using CL commands or System i Navigator functions.

The following CL commands can be used to work with journals:

- To recover a damaged or unusable database file member that contains journaled changes, use the Apply Journaled Changes (APYJRNCHG) and Remove Journaled Changes (RMVJRNCHG) commands.
- To apply the changes that were recorded in a journal receiver to the designated physical file member, use the APYJRNCHG command. However, depending on the type of damage to the physical file and the amount of activity since the file was last saved, removing changes from the file using the RMVJRNCHG command can be easier.
- To convert journal entries to a database file, use the Display Journal (DSPJRN) command. Use this file for activity reports, audit trails, security, and program debugging.

> **Related concepts**
>
> Journal management
>
> Control language
>
> **Related reference**
>
> Apply Journaled Changes (APYJRNCHG) command
>
> Removed Journaled Changes (RMVJRNCHG) command
>
> Display Journal (DSPJRN) command

Creating a journal using System i Navigator:

Creating a journal causes a new instance of a journal on your system. You must start journaling the table to a journal before it begins journaling information.

To create a journal using System i Navigator, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas**.
4. Right-click the schema in which you want to create the journal, and click **New** → **Journal**.
5. In the New Journal window, specify a journal name in the **Name** field.
6. Optional: In the **Description** field, specify a description for the journal.

7. Select a library in which to store the journal receivers.
8. Click **OK** to create the journal.

You can now create the journal using the default values. You can edit the journal default values by clicking **Advanced**. To create the journal using the default values, click **OK**.

To edit the journal default values, first create the journal and then follow these steps:
1. In the New Journal window, click **Advanced**.
2. In the **Journal message queue** field, select the journal message queue. The default is the System Operator message queue. You can specify another message queue. However, if the message queue that you specify is not available when the message is sent, the message is sent to the System Operator message queue.
3. Specify the library in which to store the journal message queue.
4. Optional: In the **Description** field, edit or specify a description.
5. Select a **Receivers managed by** option. Your choice is **System** or **User**.
6. If you do not want to record the job, program, and user profile information, select **Minimize fixed portion of entries**. This minimizes the size of each journal entry but limits the selection criteria that can be used on other journal commands.
7. If you want to automatically remove only the internal journal entries required for system restart recovery, select **Remove internal entries**. Removing these entries reduces the size of the journal receiver.
8. If you want to edit the default values of the new journal receiver, click **New Receiver**. A new journal receiver is created at the same time as the journal.
9. After you have completed the **Advanced** options, click **OK** to return to the New Journal window.
10. Click **OK** to create the journal.

   Related concepts
   "Creating a journal receiver using System i Navigator"
   A *journal receiver* is a file that contains the information that a journal is recording. When you create a journal, the receiver is automatically created. However, if you want to manually swap receivers, you must first create a journal receiver.

Creating a journal receiver using System i Navigator:

A *journal receiver* is a file that contains the information that a journal is recording. When you create a journal, the receiver is automatically created. However, if you want to manually swap receivers, you must first create a journal receiver.

To create a journal receiver using System i Navigator, follow these steps:
1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the journal for which you want to add a receiver.
4. Right-click the journal for which you want to add a receiver, and click **Properties**.
5. In the Journal Properties window, click **Receivers**.
6. In the Receivers for Journal window, click **New**.
7. In the New Journal Receiver window, specify a name (limited to 10 characters), a library to contain the receiver, a description, and a storage space threshold.
8. Click **OK** to close the New Journal Receiver window, the Receivers for Journal window, and the Journal Properties window.

While you are creating a journal, you can also create a journal receiver by following these steps:
1. In the Advanced Journal Attributes window, click **New Receiver**.

2. In the New Journal Receiver window, specify a name (limited to 10 characters), a library to contain the receiver, a description, and a storage space threshold.

3. Click **OK** to close the New Journal Receiver window.

4. Click **OK** to close the Advanced Journal Attributes window. If you do not specify values for the journal receiver, it will be created with the default values.

   Related concepts

   "Creating a journal using System i Navigator" on page 227
   Creating a journal causes a new instance of a journal on your system. You must start journaling the table to a journal before it begins journaling information.

Values for new journals and journal receivers:

The journal and journal receiver are created or changed with the values that you specify on the Advanced Journal Attributes or the Journal Properties window. If you do not specify any values, the journal and journal receiver are created with the default values.

For journals:
- The journal is created in the library in focus.
- The storage space for the journal is allocated from the same auxiliary storage pool (ASP) as the storage space of the journal's library's ASP. This value cannot be changed.
- The message queue associated with the journal is the System Operator message queue.
- The swapping of journal receivers is set so that the system automatically does the swapping.
- The journal receivers are not automatically deleted by the system during swap processing.
- The fixed portion of journal entries are not minimized, but the internal journal entries are removed.
- The public authority for the journal is taken from the default public authority for the library.
- No default text description is created for the journal.

For journal receivers:
- The storage space for the journal receiver is allocated from the same auxiliary storage pool (ASP) as the storage space of the journal receiver's library's ASP. This value cannot be changed.
- The storage space threshold for the journal receiver is set at 500 megabyte (MB).
- The public authority for the journal receiver is taken from the default public authority for the library.
- A default text description is created for the journal receiver.

Adding a remote journal using System i Navigator:

Remote journals allow you to replicate journal information to a separate system. A remote journal is associated with an existing journal. The journal on the source system can be either a local journal or another remote journal.

To add a remote journal using System i Navigator, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.

2. Expand the database that you want to work with.

3. Expand **Schemas** and the schema that contains the journal to which you want to add a remote journal.

4. Click **Journals**.

5. Right-click the journal to which you want to add a remote journal, and click **Properties**.

6. In the Journal Properties window, click **Remote Journals**.

7. In the Remote Journals window, to add (that is, to associate) a remote journal to this journal, click **Add**.

a. In the Add a Remote Journal window, click the arrow in the **Relational database name** field to display a list of relational database (RDB) directory entries.

b. Select the journal type (Type 1 or Type 2).

c. To associate the remote journal receivers on the target system with a library different from that used on the source system, select **Redirect receiver**.

d. In the **Target receiver schema** field, specify the library on the target system where the remote journal receivers are to be located.

8. Click **OK** to close the Add a Remote Journal window.

The remote journal type influences the redirection capabilities, journal receiver restore operations, and remote journal association characteristics.

Limited redirection (Type 1) If the remote journal being added is Type 1, the journal library name be redirected to a single different library from that of the local journal. All Type 1 remote journals associated with a given local journal must reside in the same library. A Type 1 remote journal cannot be added to a Type 2 remote journal.

Flexible redirection (Type 2) If the remote journal being added is Type 2, you can specify a redirected library that is the same as or different from the redirected library specified on previous additions of Type 2 remote journals. Subsequent additions of Type 2 remote journals can specify a different library redirection from what was specified on any previously added remote journal.

After you have added a remote journal, you must activate it.

Note: This task assumes that you have an RDB directory entry and that you have a user profile on the target system.

Removing a remote journal using System i Navigator:

Removing a remote journal disassociates it from the journal on the source system; it does not delete the journal or the journal receivers. You must deactivate a remote journal before you can remove it.

Activating a remote journal using System i Navigator:

After you add a remote journal, you must activate it.

To activate a remote journal, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.

2. Expand the database that you want to work with.

3. Expand **Schemas** and the schema that contains the journal that has the associated remote journal you want to activate.

4. Click **Journals**.

5. Right-click the journal that has the associated remote journal you want to activate, and click **Properties**.

6. In the Journal Properties window, click **Remote Journals**.

7. In the Remote Journals window, select the remote journal in the list of remote journals, and then click **Activate** to activate the selected remote journal.

8. In the Activate window, select the delivery mode, the starting receiver, and the processing mode for the activate request.

9. Click **OK** to close the Activate window.

Notes:

- Activating a remote journal for the first time creates one or more journal receivers on the target system.
- Activating the remote journal establishes the connection between the source and remote journals so that journal entry replication can begin.
- The remote journal must not be active before you can activate it. Also, the remote journal that you are activating must not be replicating journal entries to other remote journals.

Deactivating a remote journal using System i Navigator:

When you deactivate a remote journal, the collection of data is stopped.

If you are deactivating a synchronously maintained journal, the remote journal function is ended immediately, regardless of whether an immediate or controlled end is requested. If the remote journal is in the catch-up phase of processing, the remote journal function is ended immediately, regardless of whether an immediate or controlled end is requested. (Catch-up processing refers to the process of replicating journal entries that existed in the journal receivers of the source journal before the remote journal was activated.) Remote journals are in catch-up if they have a delivery mode of asynchronous pending or synchronous pending.

Controlled
Deactivates the remote journal function in a controlled manner. This means that the system should replicate all journal entries already queued to be sent from the source system to the target system before deactivating the remote journal. Until all queued entries have been replicated, the remote journal will be in a state of inactive pending. While in a state of inactive pending, the Remote Journals window provides inactive pending information. When all queued entries have been sent to the target system, the system sends a message to the journal message queue, indicating that the remote journal function has been ended.

Immediately
Deactivates the remote journal function immediately. This means that the system should not continue to replicate any journal entries that are already queued before deactivating the remote journal.

Displaying journal information for a table using System i Navigator:

You can get journal information for a table from System i Navigator.

To display journal information for a table, follow these steps:
1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the table for which you want to display journal information.
4. Click **Tables**.
5. Right-click the table for which you want to display journal information, and click **Journaling**.
6. If the table has never been journaled, you can specify the journal that you want to use by typing the journal and library names in the appropriate fields, or by clicking the **Browse** button and navigating to the location of the journal that you want to use for the table.
7. To journal before images, select the **Journal image before change** check box.
8. To omit open and close entries from being journaled, clear the **Include open and close entries** check box.

Swapping journal receivers using System i Navigator:

Swapping journal receivers replaces the current journal receiver with a new journal receiver that is automatically created and attached to the journal by the system. You can use two methods to swap receivers. The first method does not change the attributes of the new receiver; the second method does.

If you select the first method, follow these steps. For this option, the system generates a new name when it creates the receiver, and you are not allowed to change the attributes of the new receiver.

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the journal that you want to use.
4. Click **Journals**.
5. Right-click the journal that you want to use, and click **Swap Receivers**.

If you select the second method, follow these steps. For this option, you are allowed to change the attributes of the new receiver.

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the journal that you want to work with.
4. Click **Journals**.
5. Double-click the journal that you want to work with.
6. In the Journal Properties window, click **Receivers**. The Receivers for Journal window displays all of the receivers that are associated with the journal.
7. To add a new receiver, click **New**.
8. Click **OK** to close the New Journal Receiver window and the Receivers for Journal window. After you click **OK** to close the Journal Properties window, the status of the new journal receiver changes to attached.

Starting or ending journaling for libraries:

You can journal a library or a list of libraries and, optionally, journal database files that are created, moved, or restored into the library or the list of libraries.

The following libraries cannot be journaled, where *xxxxx* is a 5-digit number and *yyyy* is a 4-digit number:

- QGPL
- QRCL
- QRCL*xxxxx*
- QRCY*xxxxx*
- QRECOVERY
- QRPLOBJ
- QRPL*xxxxx*
- QSPL
- QSPL*yyyy*
- QSYS
- QSYS*xxxxx*
- QSYS2
- QSYS2*xxxxx*
- QTEMP
- SYSIB*xxxxx*
- SYSIBM
- SYSIBMADM
- SYSPROC
- SYSTOOLS

To start journaling changes to a library, use the Start Journal Library (STRJRNLIB) command. When the STRJRNLIB command is used, the library must exist and you must have the appropriate authority to the journal. Authority to the journal is not required later when files are implicitly journaled.

When you start journaling changes to a library by using the STRJRNLIB command, you can specify whether database files that are created, moved, or restored into the library should inherit the journaling state of the library and which journaling attributes they should have. Based on the inheritance rules that you defined using the INHRULES parameter, files that are created, moved, or restored into the journaled library can automatically start journaling to the same journal as the library.

Journal entries for libraries and for files that were created, moved, or restored into the libraries and that started journaling because of the library journaling inheritance can be applied by the Apply Journaled Changes (APYJRNCHG) command or the Apply Journaled Changes Extend (APYJRNCHGX) command.

Note: If a journaled library contains a data area named QDFTJRN, no files that are created, moved, or restored into the library can inherit the journaling state of the library. Instead, files that are created, moved, or restored into the library automatically start journaling based on the rules defined in the QDFTJRN data area.

To end journaling changes to a library, use the End Journal Library (ENDJRNLIB) command. When journaling is ended, the inheritance rules that were previously defined for the library are not deleted.

Related concepts

Automatically starting journaling

"Implicit physical file journaling" on page 38
A physical file can be journaled automatically when it is created.

Journaling libraries

Related reference

Start Journal Library (STRJRNLIB) command

End Journal Library (ENDJRNLIB) command

Change Journaled Object (CHGJRNOBJ) command

Apply Journaled Changes (APYJRNCHG) command

Apply Journaled Changes Extend (APYJRNCHGX) command

Starting or ending journaling for tables (files) using System i Navigator:

After you create a journal, you must start it for a table (file). If you want to delete a journal, you must end it.

To start or end journaling for a table using System i Navigator, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the journal that you want to edit.
4. Click **Journals**.
5. Right-click the journal that you want to edit, and click **Start or End Table Journaling**. The Start/End Journaling window opens.
   - To start journaling for a table, select the table that you want to journal from the **Tables** list, and then click **Add**. Or you can drag a table from the **Tables** list to the **Tables to journal** list.
   - To end journaling for a table, select the table that you no longer want to journal from the **Tables already journaled** list, and then click **Remove**.
   - To end journaling for all the tables at once, click **Select all** to select all the tables listed in the **Tables already journaled** list, and then click **Remove**.

6. Click **OK** to close the Start/End Journaling window.

You can also start or end journaling for a table by following these steps:

1. Right-click the table for which you want to start or end journaling, and click **Journaling**.
2. To end journaling for the table, click **End**. To start journaling for the table, continue with the following steps:

   a. Select a journal to associate with the table. You can also browse for the journal.

   b. Select a library in which the journal is located. This field is automatically completed when you select a journal from **Browse**.

   c. To journal before images, select the **Journal images before change** check box.

   d. To omit open and close entries from being journaled, clear the **Include open and close entries** check box.

   e. Click **Start** to start journaling for the table.

Ensuring data integrity with commitment control:

Commitment control allows you to define and process a number of changes to database files in a single unit (transaction).

Commitment control ensures that complex application transactions are logically synchronized, even if the job or system ends. Two-phase commitment control ensures that committable resources, such as database files on multiple systems, remain synchronized.

You implement commitment control in your database by executing commit and rollback operations. Using SQL, you use the COMMIT and ROLLBACK statements.

> **Related concepts**
> Commitment control
> **Related reference**
> COMMIT
> ROLLBACK

Transactions:

A transaction is a group of changes that appear as a single change, such as the transfer of funds from a savings account to a checking account.

Transactions can be classified into the following categories:

* 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. These changes represent only part of a logical group of transactions.

Revisions made to files during transaction processing are journaled when using commitment control.

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, the files reflect an incomplete logical transaction. For example, the job might have updated a record in file A, but before it updates a corresponding record in file B, the job ended abnormally. In this case, the logical transaction consists of two updates, but only one update had completed before the job ended abnormally.

Benefits of using commitment control:

Recovering a complex application requires detailed application knowledge. Programs cannot be restarted. In this case, commitment control helps solve these problems.

Sometimes record changes might 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 are accessing the files at the same time. In this case, commitment control can help. Commitment control locks records from other users during a complex transaction. 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, if the system ends abnormally before performing the commit operation, 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 roll back to a clean transaction boundary.

Usage notes: Commitment control:

The commit and rollback operations are available in several programming languages that are supported on the i5/OS operating system, including RPG, COBOL, PL/I, SQL, and CL.

You can open logical files for output under commitment control when underlying physical files are journaled to different journals. Commitment control can also be used in a batch environment.

However, the checks for violations are deferred if a record change affects underlying physical files that are journaled to the same journal. If the record change affects underlying physical files that are not journaled to the same journal, and it causes a duplicate key or referential constraint violation, an error will occur during the input/output operation. For example, assume physical file A with a unique key is journaled to journal X, while physical file B with a unique key is journaled to journal Y. Logical file C is created over physical files A and B and opened under commitment control. A delete operation performed using logical file C removes a record from physical file A with key K. It is possible to add a record back to physical file A with key K before the transaction is committed. However, an attempt to add a record to physical file B with key K, before the transaction is committed, will fail since physical files A and B are journaled to different journals.

Just as it provides assistance in interactive transaction recovery, commitment control can also help in batch job recovery.

> **Related concepts**
> Commitment control

Reducing time in 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. Access path recovery might take a long time; however, you can reduce the recovery time in several ways.

Journaling access paths is often faster than rebuilding access paths. With the System-managed access path protection (SMAPP) support, you do not have to use the journaling commands, such as the Start Journal Access Path (STRJRNAP) command, to get the benefits of access path journaling. SMAPP support recovers access paths after an abnormal system end rather than rebuilding them during IPL.

Saving access paths:

You can reduce the time of access path recovery by saving access paths. To save access paths, you can use the access path (ACCPTH) parameter on the Save Changed Objects (SAVCHGOBJ), Save Library (SAVLIB), and Save Object (SAVOBJ) commands.

Normally, the system saves only descriptions of logical files; however, the system saves access paths under the following conditions:

- ACCPTH(*YES) is specified.
- All physical files under the logical file are being saved and are in the same library.
- The logical file is MAINT(*IMMED) or MAINT(*DLY).

Note:

The logical file itself is not saved when you have specified the ACCPTH(*YES) parameter. You must save the logical file explicitly.

> **Related concepts**
>
> Backup and recovery

Restoring access paths:

The system has the ability to restore access paths. It usually restores an access path faster than it rebuilds an access path.

The system can restore access paths if:
- They were previously saved.
- All the physical files on which they depend are restored at the same time.

For example, assume that a logical file is built over a physical file that contains 500 000 records. You have determined through the Display Object Description (DSPOBJD) command that the size of the logical file is about 15 megabytes.

In this example, it takes about 50 minutes to rebuild the access path for the logical file. It takes about 1 minute to restore the same access path from a tape. (This assumes that the system builds approximately 10 000 index entries per minute.)

After restoring the access path, you might need to update the file by applying the latest journal changes. For example, the system applies approximately 80 000 to 100 000 journal entries to the file per hour. This assumes that each of the physical files to which entries are being applied has only one access path built over it. This rate drops proportionally for each access path of *IMMED maintenance that is present over the physical file. Even with this additional recovery time, you usually find that it is faster to restore access paths than to rebuild them.

> **Related concepts**
>
> Journal management

Journaling access paths:

Journaling access paths can reduce recovery time by reducing the number of access paths that need to be rebuilt after an abnormal system end. It is suggested that you journal access paths because larger access paths require more time to rebuild.

When you journal database files, you record images of changes to the file in the journal. The system uses these record images to recover the file after an abnormal system end.

After an abnormal end, the system might find that access paths are not synchronized with the data in the file. If an access path does not synchronize with its data, the system rebuilds the access path to ensure that the two are synchronized and usable.

When journaling access paths, 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 recovers both the data files and the access paths. The system then synchronizes the two. In such cases, you avoid the lengthy time to rebuild access paths.

In addition, other system recovery functions work with journaling access paths. For example, the system has a number of options to reduce recovery time from the failure and replacement of a disk unit. These options include user auxiliary storage pools and checksum protection. These options further reduce the chances that the entire system must reload because of the disk failure. However, you might still need to rebuild access paths when the system is started following replacement of the failed disk. By using access path journaling and some of the recovery options, you reduce your chances of having to reload the entire system and having to rebuild access paths.

Before journaling an access path, you must journal the physical files that are associated with the access path. In addition, you must use the same journal for the access path and its associated physical files. It is easy to start journaling access paths:
- You can use the system-managed access-path protection (SMAPP) facility.
- You can manage the journaling environment yourself with the Start Journal Access Path (STRJRNAP) command.
  - To start journaling the access path for the specified file, use the STRJRNAP command. You can journal access paths that have a maintenance attribute of immediate (*IMMED) or delayed (*DLY).
  - After you start journaling, the system protects the access path until the access path is deleted or until you run the End Journal Access Path (ENDJRNAP) command for that access path.

Access path journaling minimizes 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 provides the best journaling performance, while helping to protect them from a disk failure.

> **Related concepts**
> Journal management
> **Related reference**
> Start Journal Access Path (STRJRNAP) command

System-managed access-path protection:

System-managed access-path protection (SMAPP) provides automatic protection for access paths. With SMAPP support, you do not have to use the journaling commands, such as the Start Journal Access Path (STRJRNAP) command, to get the benefits of access path journaling.

SMAPP support recovers access paths after an abnormal system end rather than rebuilding them during IPL. The shipped system automatically turns on SMAPP support. The system sets SMAPP support to a value of 70 minutes.

The system determines which access paths to protect based on:
- Target access path recovery times provided by the user, or
- A system-provided default time.

You specify target access path recovery times as a system-wide value or on an auxiliary storage pool (ASP) basis. Access paths already in a user-defined journal are ineligible for SMAPP protection because they are already protected.

> **Related concepts**
> Journal management

Rebuilding access paths:

Rebuilding a database access path might take as much as one minute for every 10 000 records. But some factors might affect the time estimate for rebuilding access paths.

The following factors affect the time estimate for rebuilding access paths:

- Storage pool size. 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.
- Key length. A large key length slows rebuilding the access path because the access path constructs and stores more key information.
- Select/omit values. Select/omit processing slows the rebuilding of an access path because the system compares each record to see if it meets the select/omit values.
- Record length. A large record length slows the rebuilding of an access path because the system looks at more data.
- Storage device that contains the data. The relative speed of the storage device that contains the actual data and the device that stores the access path affects 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 affect on how fast the system builds the access path while trying to maintain an efficient access path.

Controlling when access paths are rebuilt:

If the system ends abnormally, it automatically lists the files that require access path recovery during the next initial program load (IPL). You can control when access paths are rebuilt.

You can decide whether to rebuild the access path:
- During the IPL
- After the IPL
- When you first use the file.

You can also:
- Change the scheduling order in which the access paths are rebuilt
- Hold rebuilding of an access path indefinitely
- Continue the IPL process while access paths with a sequence value that is less than or equal to the *IPL threshold* value are rebuilding.
- Control the rebuilding of access paths after the system has completed the IPL process by using the Edit Rebuild of Access Paths (EDTRBDAP) command.

The IPL threshold value determines which access paths to rebuild during the IPL. All access paths with a sequence value that is less than or equal to the IPL threshold value rebuild during the IPL. Changing the IPL threshold value to 99 means that all access paths with a sequence value of 1 through 99 rebuild during the IPL. Changing the IPL threshold value to 0 means that no access paths rebuild until after the system completes its IPL, except those access paths that were being journaled and those access paths for system files.

The access path recovery value for a file is determined by the value you specified for the RECOVER parameter on the create and change file commands. The default recovery value for *IPL (rebuild during IPL) is 25 and the default value for AFTIPL (rebuild after IPL) is 75; therefore, RECOVER(*IPL) will show as 25. The initial IPL threshold value is 50; this allows the parameters to affect when the access path is rebuilt. You can override this value on the Edit Rebuild of Access Paths display.

If a file is not needed immediately after IPL time, specify that the file can be rebuilt at a later time. This should reduce the number of access paths 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 paths rebuilt should rebuild the access paths 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 to

rebuild first. This method shortens the IPL time and can make the first of several applications available faster. However, the overall time to rebuild access paths probably is longer. (Other work might be running when the access paths are being rebuilt, and there might be less main storage available to rebuild the access paths).

Related reference

Edit Rebuild of Access Paths (EDTRBDAP) command

Designing files to reduce access path rebuilding time:

File design can help reduce access path recovery time.

For example, you might divide a large master file into a history file and a transaction file. The system uses the transaction file for adding new data. The system uses the history file 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, you shorten the time to rebuild access paths.

However, if the system abnormally ended during the day, the access path to the smaller transaction file might need to be rebuilt. Still, the access path to the large history file, being read-only for most of the day, would rarely be unsynchronized with its data. Therefore, you reduce the chance of rebuilding this access path.

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 above file design might require a more complex application design. After evaluating your situation, you might decide to use system-supplied functions like journaling your access paths rather than design applications that are more complex.

Other methods to avoid rebuilding access paths:

If you do not journal access paths or use system-managed access-path protection (SMAPP), consider other system functions that reduce the chances of rebuilding access paths.

The system uses a file synchronization indicator to determine if an access path needs to be rebuilt. Normally, the synchronization indicator is on, indicating the synchronization of the access path and its associated data. 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.

You need to periodically synchronize the data with its access path to reduce the number of access paths you must rebuild. 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. Specify the force-access-path (FRCACCPTH) parameter on the create, change, or override database file commands.
* Force write ratio of 2 or greater. Specify the force-write-ratio (FRCRATIO) parameter on the create, change, or override database file commands.
* Force end of data. Running the force-end-of-data operation in your program can synchronize the file's data and its access path. (Some high-level languages do not have a force-end-of-data operation. See your high-level language topic collection for further details.)

Performing one of the methods mentioned previously synchronizes the access path and the data. However, the next change to the data in the file can turn the synchronization indicator off again.

Note that each of the methods can be costly in terms of performance; therefore, use them with caution. Consider journaling access paths along with saving access paths or using SMAPP as the primary means of protecting access paths.

Database recovery process after an abnormal system end

After an abnormal system end, the system proceeds through several automatic recovery steps. The system rebuilds the directory and synchronizes the journal to the files being journaled. The system performs recovery operations during IPL and after IPL.

Database file recovery during the IPL:

During the initial program load (IPL), nothing but the recovery function is active on the system.

Database file recovery during the IPL consists of the following actions:
* The listed 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 member
  – Grant authority
  – Revoke authority
  – Start journal physical file
  – Start journal access path
  – End journal physical file
  – End journal access path
  – Change journal
  – Delete journal
  – Recover SQL views
  – Remove physical file constraint
* The listed functions that were in progress when the system ended are backed out (and you must run them again):
  – Create file
  – Add member
  – Change file
  – Create journal
  – Restore journal
  – Add physical file constraint
* If the operator is doing the IPL (attended IPL), the Edit Rebuild of Access paths display appears on the operator's display. The display allows the operator to edit the RECOVER option for the files that were in use for which immediate or delayed maintenance was specified. If all access paths are valid, or the IPL is unattended, no displays appear.
* Access paths that:
  – have immediate or delayed maintenance
  – are specified for recovery during IPL (from the RECOVER option or changed by the Edit Rebuild of Access Paths display)
  – are rebuilt and a message is sent when you start journaling your access paths. Placing journal receivers in their own user auxiliary storage pool provides the best journaling performance, while helping to protect them from a disk failure.
  Related concepts

Journal management

Database file recovery after the IPL:

This recovery of database files runs after the initial program load (IPL) is completed. Interactive and batch jobs can continue with the recovery.

The database file recovery after the IPL consists of the following actions:
- The access paths for immediate or delayed maintenance files that specify recovery after IPL are rebuilt.
- The system history log receives messages that indicate the success or failure of the rebuild operations.
- After IPL completion, use the Edit Rebuilt of Access Paths (EDTRBDAP) command to order the rebuilding of access paths.
- After IPL completion, the Edit Check Pending Constraints (EDTCPCST) command displays a list of the physical file constraints in check pending. This command specifies the verification sequence of the check pending constraints.

Note: If you are not using journaling for a file, records might or might not exist after IPL recovery:
  - For added records, if after the IPL recovery the Nth record added exists, then all records added preceding N also exist.
  - For updated and deleted records, if the update or delete operation on the Nth record is present after the IPL recovery, there is no guarantee that the records updated or deleted prior to the Nth record are also present in the database.
  - For REUSEFLT(*YES), records added are treated as updates, and these records might not exist after IPL recovery.

  Related reference

  Edit Rebuild of Access Paths (EDTRBDAP) command

  Edit Check Pending Constraints (EDTCPCST) command

Effects of the storage pool paging option on database recovery:

The shared pool paging option controls whether the system dynamically adjusts the paging characteristics of the storage pool for optimum performance.
- The system does not dynamically adjust paging characteristics for a paging option of *FIXED.
- The system dynamically adjusts paging characteristics for a paging option of *CALC.
- You can also control the paging characteristics through an application programming interface. For more information, see Change Pool Tuning Information API (QWCCHGTN) in the Application programming interfaces (APIs) topic.

A shared pool paging option other than *FIXED can have an impact on data loss for nonjournaled physical files in a system failure. When you do not journal physical files, data loss from a system failure, where memory is not saved, can increase for *CALC or USRDFN paging options. You can write file changes to auxiliary storage less frequently for these options. There is a risk of data loss for nonjournaled files with the *FIXED option, but the risk can be higher for *CALC or user-defined (USRDFN) paging options. For more information about the paging option, see Automatically tune performance of the Performance topic.

  Related concepts

  Adjusting performance automatically

  Application programming interfaces

  Performance

  Related reference

  Change Pool Tuning Information (QWCCHGTN) API

Database file recovery options table:

The table outlines the recovery options for database files.

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	• 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

Database save and restore

You can save and restore database files and related objects with any supported device and media or a save file.

A save file (or the media) receives a copy, written in special format, of the saved information. You can remove and store the media for future use on the System i platform. Restored information is read from the media or from a save file into storage, where system users access the information.

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. An operator does not need to load tapes or diskettes when saving to a save file. However, periodically use the Save Save File Data (SAVSAVFDTA) command to save the save file data on tape or diskette. Periodically remove and store the tapes or diskettes away from the site. These media are then available to help you recover in case of a site disaster.

> **Related concepts**
>
> Backup and recovery
>
> **Related reference**
>
> Save Save File Data (SAVSAVFDTA) command

Database considerations for save and restore

Consider these tips when using the i5/OS database save and restore functions.

• 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. Display the object description and other save/restore related information by using the Display Object Description (DSPOBJD) command with DETAIL(*FULL).

• To display the objects in a save file, use the Display Save File (DSPSAVF) command.

• To display the objects on the media, specify DATA(SAVRST) on the Display Diskette (DSPDKT) or Display Tape (DSPTAP) command.

- To display the last save/restore date for a database file, type: DSPFD FILE(file-name) TYPE(*MBR).

Also consider automatically writing records to auxiliary storage.

Force-writing data to auxiliary storage

The force-write ratio (FRCRATIO) parameter on the create file and override database file commands indicates how often the records are to be written to auxiliary storage.

A force-write ratio of one immediately writes every add, update, and delete request to auxiliary storage for the file in question. However, choosing this option can reduce system performance. Therefore, consider saving your files and journaling your files as the primary methods for protecting database files.

Related concepts

Backup and recovery

Related reference

Display Object Description (DSPOBJD) command

Display Save File (DSPSAVF) command

Display Tape (DSPTAP) command

Using source files

DB2 for i5/OS provides a range of support for source files.

Related concepts

"Setting up source files" on page 14
You can either create a source file or use an IBM-supplied source file.

Working with source files

You can use various methods to enter and maintain data in a source file.

Related tasks

"Creating a logical file" on page 39
You can create a logical file using data description specifications (DDS).

Using the source entry utility:

You can use the source entry utility (SEU) to enter and change source in a source file. SEU is part of the IBM WebSphere Development Studio for System i licensed program.

If you use SEU to enter source in a database file, SEU adds the sequence number and date fields to each source record.

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 can 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. For more information about how to update source files, see the *ADTS for AS/400®:*

Source Entry Utility manual, SC09-2605. This manual is available from the IBM Publications Center as a printed hardcopy that you can order or in an online format that you can download at no charge.

Using device source files:

Tape and diskette 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:

You can use the Copy Source File (CPYSRCF) and Copy File (CPYF) commands to write data to and from source file members.

When you are copying from a database source file to another database source file that has an insert trigger associated with it, the trigger program is called for each record copied.

Copying to and from source files using the Copy Source File (CPYSRCF) command:

The Copy Source File (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 for 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 are always replaced. The CPYSRCF command also supports a unique printing format when TOFILE(*PRINT) is specified. Therefore, when you are copying database source files, you might want to use the CPYSRCF command.

The CPYSRCF command automatically converts the data from the from-file CCSID to the to-file CCSID.

> **Related reference**
>
> Copy Source File (CPYSRCF) command

Copying to and from source files using the Copy File (CPYF) command:

The Copy File (CPYF) command provides additional functions over the Copy Source File (CPYSRCF) command for copying to and from source files.

The CPYF command allows:
- Copying from database source files to device files
- Copying from device files to database source files
- Copying between database files that are not source files and source database files
- Printing a source member in hexadecimal format
- Copying source with selection values

> **Related reference**
>
> Copy File (CPYF) command

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 example is about 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 example is about 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 example is about copying the member PAY1 to the printer file QSYSPRT (the default for *PRINT). A format similar to the one used by service entry utility (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.

Loading and unloading data from systems other than System i:

You can use the Copy From Import File (CPYFRMIMPF) and Copy To Import File (CPYTOIMPF) commands to import (load) or export (unload) data from and to systems other than System i.

To import database data from a system other than System i into an externally described DB2 for i5/OS database file with the CPYFRMIMPF and CPYTOIMPF commands, follow these steps:

1. Create an import file for the data that you want to copy. The import file can be a database source file or an externally-described database file that has 1 field. The field must have a data type of CHARACTER, IGC OPEN, IGC EITHER, IGC ONLY, or UCS-2.
2. Send the data to the import file (or, the from file). The system performs any required ASCII-to-EBCDIC conversion during this process. You can send the data in several ways:
   - TCP/IP file transfer (file transfer)
   - System i Access support (file transfer and Open Database Connectivity (ODBC))
   - Copy from Tape File (CPYFRMTAP) command
3. Create an externally described DB2 for i5/OS database file or a distributed data management (DDM) file into which you want to copy the data.
4. Use the CPYFRMIMPF command to copy the data from the import file to your database file. If you have the DB2 Symmetric Multiprocessing option installed on the i5/OS operating system, the operating system will copy the file in parallel.

To export database data to a system other than System i, use the CPYTOIMPF command to copy the data from your database file to the import file. Then send the data to the system to which you are exporting the data.

Related reference

Copy From Import File (CPYFRMIMPF) command

Copy To Import File (CPYTOIMPF) command

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 perform any input/output operations on the file just as you do on 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).

The program processing the FILEA member of the QDDSSRC file can:
- 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).

Related concepts

"Displaying the description of the fields in a file" on page 217
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).

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 control language 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 follow this procedure:
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.
2. Give the source member the same name as the object to be created.

For example, to create the control language (CL) program PGMA using the command defaults, you can 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)
```

> **Related concepts**
>
> "IBM-supplied source files" on page 16
> For your convenience, the i5/OS licensed program and other licensed programs provide a database
> source file for each type of source.

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 Database file management topic.

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 control language (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.

Related concepts

Database file management

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 and time when 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 and time 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 and 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.

Related reference

Display Object Description (DSPOBJD) command

Display File Description (DSPFD) command

Managing a source file

You can change source file attributes and the manner in which the data in a source file is stored. You can determine when a source statement was last changed. You can also use an IBM-supplied source file for online documentation.

Changing source file attributes:

There are various methods for changing source file attributes.

If you are using the source entry utility (SEU) to maintain source files, see the *ADTS for AS/400: Source Entry Utility* manual, SC09-2605, for information about how to change source files. This manual is available from the IBM Publications Center as a printed hardcopy that you can order or in an online format that you can download at no charge. If you are not using SEU to maintain source files, you must totally replace the existing source 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 must 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 then created again using the changed source file. For example, if you change the source file FRSOURCE created in the "Creating an object using a source file" on page 246

246, you must 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 it uses an arrival sequence access path. You must specify several parameters to assign unique sequence numbers.

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.

> **Related reference**
> Reorganize Physical File Member (RGZPFM) command

Determining when a source statement was changed:

Each source record contains a date field that is automatically updated by the source entry utility (SEU) if a change is made to the statement. You can use this date field 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 to the member any time it is closed after being updated.

The second date/time field reflects any changes to the member. This includes all changes caused by SEU, commands (such as CRYF and CPYSRCF), 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 field is used by the Save Changed Objects (SAVCHGOBJ) command to determine if the member should be saved. Both date/time fields can be displayed with the Display File Description (DSPFD) command specifying TYPE(*MBR). There are two changed 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 might be a 1- or 2-second difference in that date/time value.
- `Last change date/time`. This value reflects any changes to the member. This includes all changes caused by SEU, commands (such as CPYF and 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.

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 the source entry utility (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.

Controlling the integrity of your database with constraints

A *constraint* is a restriction or limitation placed on a database file to ensure that the data in your database remains consistent when you add, change, and remove records.

- Unique constraints and primary key constraints let you create enforced unique keys for a physical file beyond the file access path.
- Check constraints provide another check for the validity of your data by testing the data in an expression.

Primary key and unique constraints can be used as the parent key when adding a referential constraint.

Setting up constraints for your database

You can use physical file constraints to control data integrity in your database. You can add constraints using the Add Physical File Constraint (ADDPFCST) command or System i Navigator.

To add a physical file constraint, use the ADDPFCST command.

- To add a unique constraint, specify a value of *UNQCST on the Type parameter. You must also specify one or more field names for the Key parameter.
- To add a primary key constraint, specify a value of *PRIKEY on the Type parameter. The key that you specify on the command becomes the primary access path of the file. If the file does not have a keyed access path that can be shared, the system creates one. You must also specify one or more field names for the Key parameter.
- To add a check constraint, specify a value of *CHKCST on the Type parameter. You must also specify a check constraint expression on the CHKCST parameter. The check constraint expression has the same syntax as the expression used for check conditions that are defined by using SQL.

You can also add constraints by using System i Navigator or when you are using the SQL CREATE TABLE and ALTER TABLE statements.

Rules for setting up constraints

Here is a list of rules that apply to all physical file constraints.

- The file must be a physical file.
- A file can have a maximum of one member, MAXMBR(1).
- A constraint can be defined when the file has zero members. A constraint cannot be established, however, until the file has one, and only one, member.
- A file can have a maximum of one primary key constraint, but might have many unique constraints.
- There is a maximum of 300 constraint relations per file. This maximum value is the sum of the following constraints:
  - The unique constraints
  - The primary key constraint
  - The check constraints
  - The referential constraints, whether they are participating as a parent or a dependent, and whether the constraints are defined or established.
- Constraint names must be unique in a library.
- Constraints cannot be added to files in the QTEMP library.

- Referential constraints must have the parent and dependent file in the same auxiliary storage pool (ASP).

   Related concepts

   DB2 for i5/OS SQL reference

   SQL programming

   Getting started with System i Navigator

   Related reference

   Add Physical File Constraint (ADDPFCST) command

   CREATE TABLE

   ALTER TABLE

Removing unique, primary key, or check constraints

You can use the Remove Physical File Constraint (RMVPFCST) command to remove a physical file constraint. The full effects of the command depend on the type of constraint that you remove and how it is used.

To remove a physical file constraint, use the RMVPFCST command.

- To remove a unique constraint, specify a value of *UNQCST on the Type parameter.
- To remove a primary key constraint, specify a value of *PRIKEY on the Type parameter.
- To remove a check constraint, specify a value of *CHKCST on the Type parameter.

You can specify any of the following values on the Constraint (CST) parameter for each of the constraint types:

- CST(*ALL) to remove all of the constraints you specify on the Type parameter.
- CST(constraint-name) to remove a specific constraint.
- CST(*CHKPND) to remove only those constraints that are in check pending status.
- Use CST(*ALL) with TYPE(*ALL) to remove all constraints from the file.

You can also remove a constraint using SQL or System i Navigator.

Considerations for removing constraints

If you remove a primary key or a unique constraint, and the associated access path is shared by a logical file, the ownership of the shared path transfers to the logical file. If the access path is not shared, it is removed.

When you remove a primary key constraint with the RMVPFCST command, the system sends an inquiry message to determine if the key specifications should be removed from the file. A reply of 'K' maintains the key specifications in the file. The file remains keyed. A reply of 'G' indicates that the file will have an arrival sequence access path when the command completes.

Note: When you remove a primary key constraint with the SQL ALTER TABLE statement, the inquiry message is not sent. The key specifications are always removed and the file has an arrival sequence access path when the ALTER TABLE completes.

   Related concepts

   DB2 for i5/OS SQL reference

   Getting started with System i Navigator

   SQL programming

   Related reference

   Remove Physical File Constraint (RMVPFCST) command

Working with a group of constraints

You can use the Work with Physical File Constraints (WRKPFCST) command to display and manage a list of constraints.

To display a list of the constraints that exist for a particular file, use the WRKPFCST command. From this display, you can change or remove a constraint and display a list of the records that placed a file constraint into check pending status.

> **Related reference**
>
> Work with Physical File Constraints (WRKPFCST) command

Details: Working with a group of constraints:

The Work with Physical File Constraints display shows all the constraints defined for a physical file.

```
                   Work with Physical File Constraints

 Type options, press Enter.
   2=Change   4=Remove   6=Display records in check pending


                                                    Check
 Opt  Constraint   File        Library   Type      State   Pending
   _   DEPTCST      EMPDIV7     EPPROD    *REFCST   EST/ENB  No
   _   ACCTCST      EMPDIV7     EPPROD    *REFCST   EST/ENB  Yes
   _   STAT84       EMPDIV7     EPPROD    *REFCST   DEF/ENB  No
   _   FENSTER      REVSCHED    EPPROD    *REFCST   EST/DSB  Yes
   _   IRSSTAT3     REVSCHED    EPPROD    *UNQCST
   _   IFRNUMBERO > REVSCHED    EPPROD    *UNQCST
   _   EVALDATE     QUOTOSCHEM  EPPROD    *REFCST   EST/ENB  No
   _   STKOPT       CANSCRONN9  EPPROD    *PRIKEY
   _   CHKDEPT      EMPDIV2     EPPROD    *CHKCST   EST/ENB  No


 Parameters for options 2, 4, 6 or command
 ===>_____
 F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F15=Sort by
 F16=Repeat position to  F17=Position to  F22=Display constraint name
```

The display lists the constraint names, the file name, and the library name. In addition, the following information is displayed:

- The Type column identifies the constraint as referential, check, unique, or primary key.
- The State column indicates whether the constraint is defined or established and whether it is enabled or disabled. The State column only applies to referential and check constraints.
- The Check Pending column contains the check pending status of the constraint. Unique and primary key constraints do not have a state because they are always established and enabled.

For each of the listed constraints, you can perform the following actions:

- To change a referential or check constraint to any of its permissible states, select Change (option 2). For example, you can enable a constraint that is currently disabled. This option performs the same functions as the Change Physical File Constraint (CHGPFCST) command.
- To remove a constraint, select Remove (option 4). This option performs the same functions as the Remove Physical File Constraint (RMVPFCST) command.
- To display the records that are in check pending state, select Display (option 6). This option performs the same functions as the Display Check Pending Constraint (DSPCPCST) command. The DSPCPCST command applies only to referential and check constraints.

Working with constraints that are in check pending status:

When you add a referential or check constraint to a physical file, the system automatically checks all the records in the file to ensure that they meet the constraint definition. If the constraint is not valid or if it cannot be verified, the system places it in check pending status.

This check is also performed when the system is being restored.

To work with the constraints that are in check pending status, follow these steps:

1. Make the constraint inactive by running the Change Physical File Constraint (CHGPFCST) command and specifying *DISABLED on the Constraint state parameter.
2. Display the list of records that caused the constraint to be marked as check pending by running the Display Check Pending Constraints (DSPCPCST) command.

   Note: The length of time that this command runs depends on the number of records the file contains.
3. Schedule the verification of the constraints that are in check pending status by running the Edit Check Pending Constraints (EDTCPCST) command.
4. Make the constraint active by running the CHGPFCST command again and specify *ENABLED on the Constraint state parameter.

   Related reference

   Change Physical File Constraint (CHGPFCST) command

   Display Check Pending Constraint (DSPCPCST) command

   Edit Check Pending Constraints (EDTCPCST) command

Displaying records that put a constraint in check pending status:

You can use the Display Check Pending Constraint (DSPCPCST) command to display the records that put a constraint in check pending status.

It is often useful to examine the records that do not conform to the rules of your constraint. You can then change either the record or the constraint as necessary.

Note: Before you perform the following step, you should run the Change Physical File Constraint (CHGPFCST) command to disable the constraint.

To display or print the list of records that have caused a constraint to be placed in check pending status, run the DSPCPCST command.

   Related reference

   Display Check Pending Constraint (DSPCPCST) command

   Change Physical File Constraint (CHGPFCST) command

Processing constraints that are in check pending status:

It might take a long time for the system to validate constraints that are created for large files. You can list the constraints that are in check pending status and schedule them for verification as required.

To display and edit the list of constraints that are in check pending status, follow these steps:

1. Run the Edit Check Pending Constraints (EDTCPCST) command.
2. Check the status of the constraint you want to process.
3. If the constraint is in a status other than RUN or READY, change the *HLD value in the Seq field to a value between 1 and 99.
4. Press Enter.

```
               Edit Check Pending Constraints

 Type sequence, press Enter.
 Sequence:   1-99, *HLD

               ------------Constraints----------  Verify    Elapsed
  Seq  Status  Cst        File        Library     Time      Time
  1    RUN     EMP1       DEP         EPPROD      00:01:00  00:00:50
  1    READY   CONST  >   DEP         EPPROD      00:02:00  00:00:00
 *HLD CHKPND   FORTH  >   STYBAK      EPPROD      00:03:00  00:00:00
 *HLD CHKPND   CST88      STYBAK      EPPROD      00:10:00  00:00:00
 *HLD CHKPND   CS317      STYBAK      EPPROD      00:20:00  00:00:00
 *HLD CHKPND   KSTAN      STYBAK      EPPROD      02:30:00  00:00:00




                                                   Bottom
  F3=Exit  F5=Refresh  F12=Cancel  F13=Repeat all  F15=Sort by
  F16=Repeat position to  F17=Position to  F22=Display constraint name
```

Details: Processing constraints that are in check pending status

Additional information about processing constraints that are in check pending status includes the Status field of the Edit Check Pending Constraints display, the Constraint column, and the Verify time column.

The Status field of the Edit Check Pending Constraints display has one of the following values:
- **RUN** indicates that the constraint is being verified.
- **READY** indicates that the constraint is ready to be verified.
- **NOTVLD** indicates that the access path that is associated with the constraint is not valid. After the access path has been rebuilt, the system automatically verifies the constraint. This value applies only to a referential constraint.
- **HELD** indicates that the constraint is not being verified. You must change the sequence to a value from 1 to 99 to change this state.
- **CHKPND** indicates that the system attempted to verify the constraint, but the constraint is still in check pending. You must change the sequence to a value from 1 to 99 to change this state.

The Constraint column contains the first five characters of the constraint name. A > symbol follows the name if it exceeds five characters. You can display the whole long name, put the cursor on that line and press the F22 key.

The Verify time column shows the time it would take to verify the constraint if there were no other jobs on the system. The elapsed time column indicates the time already spent on verifying the constraint.

> **Related reference**
>
> Edit Check Pending Constraints (EDTCPCST) command

Unique constraints

Unique constraints act as controls in a database to ensure that rows are unique.

For example, you can specify a customer identification number as a unique constraint in your database. If anyone attempts to create a new customer with the same customer number, an error message is sent to the database administrator.

Unique constraints identify a field or set of fields in a database file whose values must be unique across records in the file. The field must be in ascending order, and can be null-capable.

A file can have multiple unique constraints, but you cannot duplicate unique constraints. The same key fields, regardless of order, constitute a duplicate constraint.

A unique constraint can be used as the parent key when adding a referential constraint.

Primary key constraints

A primary key constraint is a unique key with special attributes that make the key the primary access path for the file.

Primary key constraints identify a field or set of fields in a database file whose values must be unique across records in the file. The field must be in ascending order, and can be null-capable. If it is null-capable, a check constraint is implicitly added so that null values cannot be entered in the field. You can define only one primary key constraint for a file.

A primary key constraint can be used as the parent key when adding a referential constraint.

Check constraints

You use check constraints to maintain limits on field values so that they conform to your database requirements.

Check constraints ensure data validity during insert or update operations by checking the data against a check constraint expression that you define.

For example, you can create a check constraint on a field and define that the values that are inserted into the field must be between 1 and 100. If a value does not fall within the range, the insert or update operation against your database is not processed.

Check constraints are much like referential constraints in terms of their states:
- **Defined and enabled**. The constraint definition has been added to the file, and the constraint will be enforced after the constraint is established.
- **Defined and disabled** . The constraint definition has been added to the file, but the constraint will not be enforced.
- **Established and enabled**. The constraint has been added to the file and all of the pieces of the file are there for enforcement.
- **Established and disabled**. The constraint has been added to the file and all of the pieces of the file are there for enforcement, but the constraint will not be enforced.

A check constraint, like a referential constraint, can have a check pending status. If the data in any field violates the check constraint expression, then the constraint is in check pending status. For the insertion or update of a record, if the data violates the check constraint expression, then the insert or update operation is not allowed.

A check constraint that contains one or more Large Object (LOB) fields is restricted to a narrower range of operations than a check constraint without LOB fields. When the check constraint includes one or more LOB fields, the LOB fields can only be involved in direct comparisons to:
- Other LOB fields of the same type and same maximum length.
- Literal values.
- The null value.

Operations known as derived operations, such as the Substring or Concat operations, are not allowed against LOB fields in a check constraint. The diagnostic message CPD32E6 will be sent when you try to add a check constraint that attempts a derived operation against a LOB field.

Ensuring data integrity with referential constraints

You use referential constraints to enforce the referential integrity of your database. Referential integrity encompasses all of the mechanisms and techniques that you can use to ensure that your database contains only valid data.

Adding referential constraints

You can add referential constraints on physical files with no more than one member. A referential constraint is a file-level attribute; therefore, you can create the constraint before the member exists.

Before you add referential constraints:

Before you add referential constraints, make sure that you meet these prerequisites.

- There must be a parent file with a key capable of being a parent key. If the parent file has no primary key or unique constraint, the system tries to add a primary key constraint to the parent file if the field attributes of the potential parent key match those of the foreign key field attributes of the dependent file.
- There must be a dependent file with certain attributes that match the attributes of the parent file:
  - Sort sequence (SRTSEQ) must match for data types CHAR, OPEN, EITHER, and HEX.
  - The coded character set identifier (CCSID) must match for each SRTSEQ table unless either (or both) of the CCSIDs is 65535.
  - Each sort sequence table must match exactly.
- The dependent file must contain a foreign key that matches the following attributes of the parent key:
  - Data type
  - Length
  - Precision (packed, zoned, or binary)
  - CCSID (unless either has a CCSID of 65535)
  - REFSHIFT (if data type is OPEN, EITHER, or ONLY)

Defining the parent file in a referential constraint:

A parent file must be a physical file with a maximum of one member. You can either create a parent file or use an existing file as the parent file.

The concept of a parent key applies only in terms of a referential constraint. When a referential constraint is added to the dependent file, a parent key is required for the parent file. To prepare for this, you must first add either a primary key constraint or a unique constraint to the parent file with the appropriate set of fields for the key. When the referential constraint is added, a search is conducted of unique constraints (and primary key) for a match. If a match is found, then the access path of the constraint is used as the parent key in the referential constraint relationship.

To create a new physical file as a parent file, follow these steps:
1. Use the Create Physical File (CRTPF) command to create the file.
2. Use the Add Physical File Constraint (ADDPFCST) command to either add a primary key constraint or a unique constraint. The primary key can be null-capable, but the system creates an implicit check constraint to prevent the insertion of null values in the field.

Note: You can use the SQL CREATE TABLE statement to perform the preceding steps with one step.

To use an existing file as a parent file, choose from the following options:
- You can add a primary key constraint to a file with the ADDPFCST command. Specify *PRIKEY for the TYPE parameter. You must also specify the key field or fields with the KEY parameter.

  If a primary key constraint already exists for the file, the ADDPFCST command with TYPE(*PRIKEY) will fail because a file can have only one primary key. If you want a different primary key constraint, you must first remove the existing primary key constraint with the Remove Physical File Constraint (RMVPFCST) command. Then you can add a new primary key constraint.

- You can add a unique constraint to a file with the ADDPFCST command. Specify *UNQCST for the TYPE parameter. You must also specify the key field or fields with the KEY parameter. You can also add a unique constraint with the Structured Query Language (SQL) ALTER TABLE statement.

  If the parent file does not have a primary key or unique constraint that can be used as the parent key, the system will attempt to automatically add a primary key constraint when adding a referential constraint.

  If the parent file has a uniquely keyed access path, where the access path fields match the foreign key's fields (both for the number of fields and matching attributes), then a primary key constraint will be implicitly added to the parent file. This will become the parent key for the referential constraint.

  If the parent file is arrival sequence access path, then if the fields specified for the parent key match the foreign key's fields (matching attributes), then a primary key constraint will be implicitly added to the parent file. This will become the parent key for the referential constraint.

What to do when you cannot define a parent key

For an existing file with a primary key or unique constraint, if neither constraint will suffice as the parent key, there are options you can turn to.

You can take either of the following actions when you cannot define a parent key.
- Delete the file and create it again with the appropriate keys.
- Add a unique or primary key constraint to the created file.

  Related reference

  Create Physical File (CRTPF) command

  Add Physical File Constraint (ADDPFCST) command

  Remove Physical File Constraint (RMVPFCST) command

Defining the dependent file in a referential constraint:

A dependent file must be a physical file with a maximum of one member. You can either create a dependent file as you do any physical file or use an existing file as the dependent file.

The dependent file does not require a keyed access path when you create the actual constraint. If no existing access paths meet the foreign key criteria, the system adds an access path to the file.

Specifying referential constraint rules:

Referential constraints allow you to specify the rules that you want the system to enforce when you delete or update records.

To specify the rules that you want to enforce with referential constraints, follow these steps:
1. Run the Add Physical File Constraint (ADDPFCST) command.
2. Specify the rule that you want to enforce when you delete records (the *delete rule*) by choosing a value for the DLTRULE parameter.
3. Specify the rule that you want to enforce when you update records (the *update rule*) by choosing a value for the UPDRULE parameter.

You can also add referential constraints using System i Navigator.

  Related concepts

  Getting started with System i Navigator

  SQL programming

  Related reference

  Add Physical File Constraint (ADDPFCST) command

Details: Specifying referential constraint delete rules:

You can specify one of these values for the DLTRULE parameter. The delete rule specifies the action that the system takes when you delete a parent key value. The delete rule does not affect null parent key values.

- *NOACTION (the default value)
  - Record deletion in a parent file will not occur if the parent key value has a matching foreign key value.
- *CASCADE
  - Record deletion in a parent file causes records in the dependent file to be deleted when the parent key value matches the foreign key value.
- *SETNULL
  - Record deletion in a parent file updates those records in the dependent file where the value of the parent non-null key matches the foreign key value. For those dependent records that meet the preceding criteria, all null-capable fields in the foreign key are set to null. Foreign key fields with the non-null attribute are not updated.
- *SETDFT
  - Record deletion in a parent file updates those records in the dependent file where the value of the parent non-null key matches the foreign key value. For those dependent records that meet the preceding criteria, the foreign key field or fields are set to their corresponding default values.
- *RESTRICT
  - Record deletion in a parent file will not occur if the parent key value has a matching foreign key value.

  Note: The system enforces a delete *RESTRICT rule immediately when the deletion is attempted. The system enforces other constraints at the logical end of the operation. The operation, in the case of other constraints, includes any trigger programs that are run before or after the delete operation. It is possible for a trigger program to correct a potential referential integrity violation. For example, a trigger program can add a parent record if one does not exist. The *RESTRICT rule does not prevent the violation.

Details: Specifying referential constraint update rules:

You can specify one of these values for the UPDRULE parameter. The UPDRULE parameter identifies the update rule for the constraint relationship between parent and dependent files. The update rule specifies the action that the system takes when you update the parent file.

- *NOACTION (the default value)
  - Record update in a parent file does not occur if there is a matching foreign key value in the dependent file.
- *RESTRICT
  - Record update in a parent file does not occur if a value of the non-null parent key matches a foreign key value.

  Note: The system enforces an update *RESTRICT rule immediately when you attempt the update operation. The system enforces other constraints at the logical end of the operation. For example, a trigger program can add a parent record if one does not exist. The *RESTRICT rule does not prevent the violation.

Details: Specifying referential constraint rules—journaling requirements:

If you perform insert, update, or delete operations on a file that is associated with a referential constraint, and the delete rule, update rule, or both are other than *RESTRICT, you must use journaling.

You must journal both the parent and the dependent files to the same journal. In addition, you are responsible for starting the journaling for the parent and dependent files.

You can start the journaling for the parent and dependent files with the Start Journal Physical File (STRJRNPF) command.

If you are inserting, updating, or deleting records to a file associated with a referential constraint that has a delete rule, update rule, or both rules, other than *RESTRICT, commitment control is required. If you have not started commitment control, the system starts and ends the commit cycle automatically for you.

> **Related reference**
> Start Journal Physical File (STRJRNPF) command

Details: Adding referential constraints:

Consider these limitations when adding referential constraints.
- A parent file must be a physical file.
- A parent file can have a maximum of one member, MAXMBR(1).
- A dependent file must be a physical file.
- A dependent file can have a maximum of one member, MAXMBR(1).
- You can define a constraint when both or either of the dependent and the parent files have zero members. A constraint cannot be established unless both files have a member.
- A file can have a maximum of one primary key, but might have many unique constraints.
- There is a maximum of 300 constraint relations per file. This maximum value is the sum of:
  - The referential constraints whether participating as a parent or a dependent, and whether the constraints are defined or established.
  - The unique constraints, which includes the primary key constraint.
  - The check constraints.
- Only externally described files are allowed in referential constraints. Source files are not allowed. Program described files are not allowed.
- Files with insert, update, or delete capabilities are not allowed in *RESTRICT relationships.
- Constraint names must be unique in a library.
- You cannot add constraints to files in the QTEMP library.
- You cannot add a referential constraint where the parent file is in one ASP and the dependent file is in a different ASP.

Details: Avoiding constraint cycles:

A *constraint cycle* is a sequence of constraint relationships in which a descendent of a parent file becomes a parent to the original parent file. You can use constraint cycles in a DB2 for i5/OS database; however, you should avoid using them.

Verifying referential constraints

The system automatically verifies the validity of a referential constraint when you add the constraint with the Add Physical File Constraint (ADDPFCST) command. The system verifies that every non-null value in the foreign key matches a corresponding value in the parent key.

If the verification is successful, the constraint rules are enforced on subsequent accesses by a user or an application program. An unsuccessful verification causes the constraint to be marked as check pending. If the constraint is added with the ADDPFCST command, it will be in check pending but disabled state.

Note: It is not uncommon to add a referential constraint to existing files that contain large amounts of data. The ADDPFCST command can take several hours to complete when a very large number of

records are involved. The add process places an exclusive lock on the files. You should take this time factor and file availability into consideration before you add a referential constraint.

Enabling or disabling referential constraints

You can enable or disable a referential constraint using the Change Physical File Constraint (CHGPFCST) command or System i Navigator.

To enable or disable a referential constraint relationship, use the CHGPFCST command. You must specify the dependent file when changing a referential constraint; you cannot disable or enable a constraint by specifying the parent file.

You can also enable or disable a referential constraint using System i Navigator.

You must have a minimum of object management authority (or ALTER privilege) to the dependent file in order to enable or disable a constraint.

Details: Enabling or disabling referential constraints

When the system enables or disables a constraint, it locks the parent and the dependent files, both members, and both access paths. It removes the locks when the enable or disable operation is complete.

Attempting to enable an enabled constraint or disable a disabled constraint does nothing but cause the issuance of an informational message.

An established/disabled or check pending constraint relationship can be enabled. The enabling causes the system to verify the constraint again. If verification finds mismatches between the parent and the foreign keys, the constraint is marked as check pending.

Disabling a constraint relationship allows all file input/output (I/O) operations for both the parent and the dependent files, if the user has the correct authority. The entire infrastructure of the constraint remains. The parent key and the foreign key access paths are still maintained. However, there is no referential enforcement that is performed for the two files in the disabled relationship. All remaining enabled constraints are still enforced.

Disabling a constraint can allow file I/O operations in performance-critical situations to run faster. Always consider the trade-off in this kind of a situation. The file data can become referentially not valid. When the constraint is enabled, depending on the file size, the system will take time to re-verify the referential constraint relationship.

Note: Users and applications must be cautious when modifying files with a constraint relationship in the established and disabled state. Relationships can be violated and not detected until the constraint is enabled again.

The Allocate Object (ALCOBJ) command can allocate (lock) files while a constraint relationship is disabled. This allocation prevents others from changing the files while this referential constraint relationship is disabled. Specify a lock state of Exclusive read (*EXCLRD) on the command so other users can read the files. After the constraint is enabled again, the Deallocate Object (DLCOBJ) command unlocks the files.

When you enable or disable multiple constraints, they are processed sequentially. If a constraint cannot be modified, you receive a diagnostic message, and the function proceeds to the next constraint in the list. When all constraints have been processed, you receive a completion message listing the number of constraints modified.

> **Related concepts**
>
> Getting started with System i Navigator

SQL programming

Related reference

Change Physical File Constraint (CHGPFCST) command

Allocate Object (ALCOBJ) command

Deallocate Object (DLCOBJ) command

Removing referential constraints

You can remove referential constraints in various ways. The full impact of the removal depends on the constraint you are removing and the certain conditions that surround the constraint.

To remove a referential constraint, follow these steps:

1. Run the Remove Physical File Constraint (RMVPFCST) command.
2. Specify the constraint or constraints you want to remove using the CST and TYPE parameters.
   - Use the CST parameter to specify all constraints or a specific constraint name.
   - Use the TYPE parameter to specify a particular type of constraint.

You can also remove a referential constraint using System i Navigator.

When you remove a referential constraint, the system removes the associated foreign keys and access paths from the file. The system does not remove the foreign key access path if any logical file or other constraint on the system uses it.

If you remove a referential, primary key, or unique constraint, and the associated access path is shared by a logical file, the ownership of the shared path transfers to the logical file.

Related concepts

Getting started with System i Navigator

SQL programming

Related reference

Remove Physical File Constraint (RMVPFCST) command

Details: Removing a constraint with the CST parameter:

You can specify which constraint you want to remove by using the CST parameter.

With the CST parameter, you can specify to remove:
- All constraints CST(*ALL) associated with a file where TYPE(*ALL) is specified
- A specific referential constraint CST(*constraint-name*)
- Referential or check constraints in check pending CST(*CHKPND)
- All constraints CST(*ALL) associated with a specific TYPE of constraint

Details: Removing a constraint with the TYPE parameter:

You can specify the type of constraint that you want to remove by using the TYPE parameter.

With the TYPE parameter, you can specify the type of constraint that you want to remove.
- All types: TYPE(*ALL)
  - All constraints for CST(*ALL)
  - All constraints in check pending for CST(*CHKPND)
  - The named constraint for CST(constraint-name)
- Referential constraints: TYPE(*REFCST)

- All referential constraints for CST(*ALL)
- All referential constraints in check pending for CST(*CHKPND)
- The named referential constraint for CST(constraint-name)
- Unique constraints: TYPE(*UNQCST)
  - All unique constraints except the primary key constraint for CST(*ALL)
  - Not applicable for CST(*CHKPND)—a unique constraint cannot be in check pending
  - The named unique constraint for CST(constraint-name)
- Primary key constraints: TYPE(*PRIKEY)
  - The primary constraint for CST(*ALL)
  - Not applicable for CST(*CHKPND)—the primary constraint cannot be in check pending
  - The named primary constraint for CST(constraint-name)
- Check constraints: TYPE(*CHKCST)
  - All check constraints for CST(*ALL)
  - All check constraints in check pending for CST(*CHKPND)
  - The named check constraint for CST(constraint-name)

Details: Ensuring data integrity with referential constraints

You might want to ensure data integrity with referential constraints for several reasons.

- To make sure that data values between files meet the rules of your business. For example, consider a business that maintains a list of customers in one file and a list of their accounts in another file. It does not make sense to allow the addition of an account if an associated customer does not exist. Likewise, it is not reasonable to delete a customer until you delete all of their accounts.
- To be able to define the relationships between data values.
- To have the system enforce the data relationships no matter what application makes changes.
- To improve the performance of integrity checks that are made at a high-level language (HLL) or SQL level by moving the checking into the database.

Example: Ensuring data integrity with referential constraints

Suppose that a database contains an employee file and a department file. You can define a referential constraint to ensure that every employee in the employee file belongs to a corresponding department in the department file.

Both the employee and department files have a department number field named DEPTNO. The related records of these database files are those for which employee.DEPTNO equals department.DEPTNO.

1. Use the Add Physical File Constraint (ADDPFCST) command, add a primary key constraint or a unique constraint to the department file for the DEPTNO field. This primary key or unique constraint will later become a parent key. It is not yet a parent key because a referential constraint has not yet been added.
2. Add a referential constraint to the employee file using the ADDPFCST command. The employee file will be the dependent file. The foreign key will be employee.DEPTNO. The department file will be the parent file with parent key department.DEPTNO. Because there is either a primary key constraint or a unique constraint with the DEPTNO field as the key, the constraint will serve as the parent key associated with the referential constraint.

The referential constraint has update and delete rules that must be followed for record insert, update, and delete operations on the parent or the dependent file.

Referential integrity terms

A discussion of referential integrity requires an understanding of several terms. These terms are arranged in an order that might help you understand their relationship to each other.

Primary key constraint

    A field or set of fields in a database file that must be unique, ascending, and cannot contain null values. The primary key is the primary file access path. The primary key constraint can be used as the parent key when adding a referential constraint. A primary key constraint is really a unique constraint with some special attributes.

Unique constraint

    A field or set of fields in a database file that must be unique, ascending, and can contain null values.

Parent key

    A field or set of fields in a database file that must be unique, ascending, and might or might not contain null values. The parent key of the parent file is used to add a referential constraint to the dependent file. The parent key must be either a primary key or a unique constraint.

Foreign key

    A field or set of fields in which each non-null value must match a value in the parent key of the related parent file.

    The attributes (data type, length, and so forth) must be the same as the parent key of the parent file.

Parent file

    The file in a referential constraint relationship that contains the parent key.

Dependent file

    The file in a referential constraint relationship that contains the foreign key. The dependent file is dependent on the parent file. That is, for every non-null value in the foreign key of the dependent file, there must be a corresponding non-null value in the parent key of the parent file.

Check pending

    The state that occurs when the database does not know with certainty whether for a referential constraint this is true: for every non-null value in the foreign key of the dependent file, there must be a corresponding non-null value in the parent key of the parent file.

Delete rule

    A definition of what action the database should take when there is an attempt to delete a parent record.

Update rule

    A definition of what action the database should take when there is an attempt to update a parent record.

Referential integrity enforcement

The input/output (I/O) access for a database file that is associated with an established and enabled constraint varies, depending on whether the file contains the parent key or foreign key in the constraint relationship. The system enforces referential integrity on all parent and dependent file I/O requests.

The database enforces constraint rules for all I/O requests whether from application programs or system commands (such as the Initialize Physical File Member (INZPFM) command) or SQL statements or file I/O utilities (such as STRSEU).

Foreign key enforcement:

The rules that you specify for a referential constraint apply to parent key changes. The database enforces a no-action rule on the insert and update operations on the foreign key to ensure that the value of every non-null foreign key matches the value of the parent key.

The system returns a referential constraint violation if a matching parent key does not exist for the new foreign key value, and does not insert or update the dependent record.

Parent key enforcement:

The rules that you specify for a referential constraint determine how the database processes the delete and update operations on the parent key. The system enforces the unique attribute of a parent key on all parent file I/O.

Enforcement of delete rules:

When you delete a record from a parent file, the system checks the dependent file for any dependent records (matching non-null foreign key values). If the system finds any dependent records, the delete rule determines the action to be taken.
- **No Action**. If the system finds any dependent records, it returns a constraint violation and does not delete records.
- **Cascade**. The system deletes dependent records that its finds in the dependent file.
- **Set Null**. The system sets null-capable fields in the foreign key to null in every dependent record that it finds.
- **Set Default**. The system sets all fields of the foreign key to their default value when it deletes the matching parent key.
- **Restrict**. Same as no action except that enforcement is immediate.

If part of the delete rule enforcement fails, the entire delete operation fails and all associated changes are rolled back. For example, a delete cascade rule causes the database to delete ten dependent records, but a system failure occurs while deleting the last record. The database will not allow deletion of the parent key record, and the deleted dependent records are re-inserted.

If a referential constraint enforcement causes a change to a record, the associated journal entry will have an indicator value noting that a referential constraint caused the record change. For example, a dependent record that is deleted by a delete cascade rule will have a journal entry indicator which indicates that the record change was generated during referential constraint enforcement.

Enforcement of update rules:

When the system updates a parent key in a parent file, it checks for any dependent records (matching non-null foreign values) in the dependent file. If the system finds any dependent records, the update rule for the constraint relationship determines the action to be taken.
- **No Action**. If the system finds any dependent records, it returns a constraint violation, does not update any records.
- **Restrict**. The system performs the same as above, but enforcement is immediate.

Constraint states

A file can be in several constraint states.
- **Non-constraint relationship state**. No referential constraint exists for a file in this state. If a constraint relationship once existed for the file, all information about it has been removed.
- **Defined state**. A constraint relationship is defined between a dependent and a parent file. It is not necessary to create the member in either file to define a constraint relationship. In the defined state, the constraint can be:
  - Defined and enabled. A defined and enabled constraint relationship is for definition purposes only. The rules for the constraint are not enforced. A constraint in this state remains enabled when it goes to the established state.
  - Defined and disabled. A defined constraint relationship that is disabled is for definition purposes only. The rules for the constraint are not enforced. A constraint in this state remains disabled when it goes to the established state.

- **Established state**. The dependent file has a constraint relationship with the parent file. A constraint will be established only if the attributes match between the foreign and parent key. Members must exist for both files. In the established state, the constraint can be:
  - Established and enabled. An established constraint relationship that is enabled causes the database to enforce referential integrity.
  - Established and disabled. An established constraint relationship that is disabled directs the database to not enforce referential integrity.

Check pending status in referential constraints

Check pending is the condition of a constraint relationship when potential mismatches exist between parent and foreign keys. When the system determines that referential integrity might have been violated, the constraint relationship is marked as check pending.

For example:
- A restore operation where only data in the dependent file is restored and this data is no longer synchronized (a foreign key does not have a parent) with the parent file on the system.
- A system failure allowed a parent key value to be deleted when a matching foreign key exists. This can only occur when the dependent and parent files are not journaled.
- A foreign key value does not have a corresponding parent key value. This can happen when you add a referential constraint to existing files that have never before been part of a constraint relationship.

Check pending status is either *NO or *YES.

Check pending applies only to the constraints in the established state. A referential constraint that is established and enabled can have a check pending status of *YES or *NO.

To get a constraint relationship out of check pending, you must disable the relationship, correct the key (foreign, parent, or both) data, and enable the constraint again. The database then verifies the constraint relationship again.

When a relationship is in check pending, the parent and dependent files are in a situation that restricts their use. The parent file I/O restrictions are different from the dependent file restrictions. Check pending restrictions do not apply to the constraints that are in the established and disabled state (which are always in check pending status).

Dependent file restrictions in check pending:

These restrictions apply to a dependent file that has an established and enabled referential constraint in check pending.

A dependent file in a constraint relationship that is marked as check pending cannot have any file input/output (I/O) operations performed on it. You must correct the file mismatches between the dependent and parent files. Also, you must take the relationship out of check pending before the system allows any I/O operations. The system does not allow records to be read from such a file because the user or application might not be aware of the check pending status and the constraint violation.

To perform I/O operations on a dependent file with an enabled referential constraint in check pending, you can first disable the constraint and then perform the I/O operations that you want.

Parent file restrictions in check pending:

These restrictions apply to a parent file that has an established and enabled referential constraint in check pending.

You can open the parent file of a constraint relationship that the system marks as check pending, but you are limited in the types of input/output (I/O) operations that you can do. You can read and insert records, but you cannot delete or update records.

To perform update and delete operations on a parent file with an enabled referential constraint in check pending, you can first disable the constraint and then perform the I/O operations that you want.

Referential integrity and CL commands

Referential integrity affects the characteristics of some CL commands.

- **Add Physical File Member (ADDPFM)**

  In the case where a constraint relationship is defined between a dependent file and a parent file each having zero members:

  - If a member is first added to the parent file, the constraint relationship remains in the defined state.
  - If a member is then added to the dependent file, the foreign key access path is built, and a constraint relationship is established with the parent.

- **Change Physical File (CHGPF)**

  When a constraint relationship exists for a file, you cannot change certain parameters available in the CHGPF command. The following parameters are restricted:

  MAXMBRS
  The maximum number of members for a file that has a constraint relationship is one: MAXMBRS(1).

  CCSID
  The CCSID of a file that is not associated with a constraint, can be changed. If the file is associated with a constraint, the CCSID can only be changed to 65535.

- **Clear Physical File Member (CLRPFM)**

  The CLRPFM command fails when issued for a parent file that contains records and is associated with an enabled referential constraint.

- **FORTRAN Force-End-Of-Data (FEOD)**

  The FEOD operation fails when issued for a parent file that is associated with an enabled referential constraint relationship.

- **Create Duplicate Object (CRTDUPOBJ)**

  If CST(*NO) is specified, constraints will not be duplicated in the new file. If CST(*YES) is specified, constraints will be duplicated. The following rules describe how constraints are duplicated:

  - If the parent file is duplicated either to the same library or to a different library, the system cross reference file is used to locate the dependent file of a defined referential constraint. Also, the system attempts to establish the constraint relationship.
  - If the dependent file is duplicated, then the TOLIB is used to determine constraint relationships:
    - If both the parent and dependent files are in the same library, the referential constraint relationship will be established with the parent file in the TOLIB.
    - If the parent and dependent files are in different libraries, then the referential constraint relationship of the duplicated dependent file will be established with the original parent file

- **Copy File (CPYF)**

  When the CPYF command creates a new file and the original file has constraints, the constraints are not copied to the new file.

- **Move Object (MOVOBJ)**

  The MOVOBJ command moves a file from one library to another. The system attempts to establish any defined referential constraints that can exist for the file in the new library.

- **Rename Object (RNMOBJ)**

  The RNMOBJ command renames a file within the same library or renames a library.

An attempt is made to establish any defined referential constraints that can exist for the renamed file or library.

- **Delete File (DLTF)**

  The DLTF command has an optional keyword that specifies how referential constraint relationships are handled. The RMVCST keyword applies to the dependent file in a constraint relationship. The keyword specifies how much of the constraint relationship of the dependent file is removed when the parent file is deleted:

  *RESTRICT

  If a constraint relationship is defined or established between a parent file and dependent file, the parent file is not deleted and the constraint relationship is not removed. This is the default value.

  *REMOVE

  The parent file is deleted, and the constraint relationship and definition are removed. The constraint relationship between the parent file and the dependent file is removed. The dependent file's corresponding foreign key access path or paths, as well as the constraint definition, are removed.

  *KEEP

  The parent file is deleted, and the referential constraint relationship definition is left in the defined state. The dependent file's corresponding foreign key access path and constraint definition are not removed.

- **Remove Physical File Member (RMVM)**

  When the member of a parent file in a constraint relationship is removed, the constraint relationship is put in the defined state. The foreign key access path and referential constraint definition are not removed. The parent key access path is removed because the parent member was removed; the parent constraint definition remains at the file level.

  When the member of a dependent file in a constraint relationship is removed, the constraint relationship is put in the defined state. The parent key access path and constraint definition are not removed. The foreign key access path is removed because the dependent member was removed; the referential constraint definition is not removed.

- **Save and restore commands**

  If the parent file is restored to a library, the system uses the system cross reference files to locate the dependent file of a defined referential constraint. An attempt is made to establish the constraint relationship.

  If the dependent file is restored, the TOLIB is used to determine constraint relationships:

  - If both the parent and dependent files are in the same library, the referential constraint relationship is established with the parent file in the TOLIB.
  - If the parent and dependent files are in different libraries, the referential constraint relationship of the duplicated dependent file is established with the original parent file.

  The order of the restore of dependent and parent files within a constraint relationship does not matter (parent restored before dependent or dependent restored before parent). The constraint relationship will eventually be established.

Triggering automatic events in your database

A *trigger* is a set of actions that run automatically when a specified change or read operation is performed on a specified database file. You can define a set of trigger actions in any high-level language that is supported on the i5/OS operating system.

The change operation can be an insert, update, or delete high-level language statement in an application program. The read operation can be a fetch, get, or read high-level language statement in an application program.

You can also use SQL triggers.

"Database data protection and monitoring" on page 7

To ensure data integrity and consistency, you can enforce either business rules or data type rules.

SQL triggers

Related information

→ Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries

Uses for triggers

Triggers in the database have several uses.

Triggers allow you to perform the following tasks:
* Enforce business rules
* Validate input data
* Generate a unique value for a newly inserted row on a different file (surrogate function)
* Write to other files for audit trail purposes
* Query from other files for cross-referencing purposes
* Access system functions (for example, print an exception message when a rule is violated)
* Replicate data to different files to achieve data consistency

Benefits of using triggers in your business

Triggers offer benefits to your business.

These benefits are:
* Faster application development. Because the database stores triggers, you do not have to code the trigger actions into each database application.
* Global enforcement of business rules. Define a trigger once and then reuse it for any application that uses the database.
* Easier maintenance. If a business policy changes, you need to change only the corresponding trigger program instead of each application program.
* Improve performance in client/server environment. All rules run in the server before the result returns.

Creating trigger programs

You can create and work with a program that contains a set of trigger actions.

To add a trigger to a database file, follow these steps:
1. Supply a trigger program. You can write a trigger program in a high-level language (HLL), Structured Query Language (SQL), or control language (CL).
2. Use one of the following methods to add the trigger:
   * The Add Physical File Trigger (ADDPFTRG) command. You must specify your trigger program in the trigger program (PGM) parameter on the command.
   * System i Navigator.
   * The CREATE TRIGGER SQL statement.

   Related reference

   Add Physical File Trigger (ADDPFTRG) command

   CREATE TRIGGER

Adding triggers using System i Navigator:

Using System i Navigator, you can define system triggers and SQL triggers. Additionally, you can enable or disable a trigger on a physical database file.

A trigger is a set of actions that run automatically when a specified change operation is performed on a specified database file. In this discussion, a table is a physical file. The change or read operation can be an insert, update, or delete high-level language statement in an application program, or an SQL INSERT, UPDATE, or DELETE statement.

To add a trigger, follow these steps:

1. From System i Navigator, expand *your system* → **Databases**.
2. Expand the database that you want to work with.
3. Expand **Schemas** and the schema that contains the table to which you want to add the trigger.
4. Click **Tables**.
5. Right-click the table to which you want to add the trigger, and select **New** → **Trigger**. Click **External** to add a system trigger or **SQL** to add an SQL trigger.

   Related concepts

   "Triggering automatic events in your database" on page 267
   A *trigger* is a set of actions that run automatically when a specified change or read operation is performed on a specified database file. You can define a set of trigger actions in any high-level language that is supported on the i5/OS operating system.
   SQL triggers

How trigger programs work:

When a user or an application issues a change or read operation on a database file that has an associated trigger, the operation calls the appropriate trigger program or programs.

The change or read operation passes two parameters to the trigger program, as described in the following table.

Parameter	Description	Input or output	Type
1	Trigger buffer, which contains the information about the current change operation that is calling this trigger program.	Input	CHAR(*)
2	Trigger buffer length.	Input	BINARY(4)

From these inputs, the trigger program can refer to a copy of the original or the new records. You must code the trigger program so that it accepts these parameters.

   Related concepts

   "Trigger buffer sections" on page 277
   The trigger buffer has two logical sections: a static section and a variable section.

Other important information about working with trigger programs:

Here are the recommendations, precautions, and error messages for trigger programs. Information about monitoring and commitment control is also included.

Recommendations for trigger programs:

Consider these recommendations when you create a trigger program.
- Create the trigger program so that it runs under the user profile of the user who created it. In this way, users who do not have the same level of authority to the program will not encounter errors.
- Create the program with USRPRF(*OWNER) and *EXCLUDE public authority, and do not grant authorities to the trigger program to USER(*PUBLIC). Avoid having the trigger program altered or

replaced by other users. The database calls the trigger program even if the user causing the trigger program to run does not have authority to the trigger program.

- Create the program as ACTGRP(*CALLER) if the program is running in an Integrated Language Environment (ILE). This allows the trigger program to run under the same commitment definition as the application.
- Open the file with a commit lock level the same as the application's commit lock level. This allows the trigger program to run under the same commit lock level as the application.
- Create the program in the physical file's library.
- Use commit or rollback in the trigger program if the trigger program runs under a different activation group than the application.
- Signal an exception if an error occurs or is detected in the trigger program. If an error message is not signalled from the trigger program, the database assumes that the trigger ran successfully. This might cause the user data to end up in an inconsistent state.

Precautions to take when coding trigger programs:

Trigger programs can be very powerful. But you must take caution when coding trigger programs.

Be careful when designing trigger programs that access a system resource, such as a tape drive. For instance, a trigger program that copies record changes to tape media can be useful, but the program itself cannot detect if the tape drive is ready or if it contains the correct tape. You must take these kinds of resource issues into considerations when designing trigger programs.

In addition, use read triggers with extreme caution. Using a read trigger causes a trigger to be called for every record that is read. During a query, this means that triggers can be called many times as records are processed multiple times by the query. This can impact system performance.

Functions to use with care in trigger programs:

Some control language (CL) commands and functions are not recommended in a trigger program and need to be carefully considered if they are to be used.

These CL commands and functions are:
- STRCMTCTL (Start Commitment Control)
- RCLSPLSTG (Reclaim Spool Storage)
- RCLRSC (Reclaim Resources)
- CHGSYSLIBL (Change System Library List)
- DLTLICPGM, RSTLICPGM, and SAVLICPGM (Delete, Restore, and Save Licensed Program)
- SAVLIB (Save Library) with SAVACT other than (*NO)
- Any commands with DKT or TAP
- Any migration commands
- The debug program (a security exposure)
- Any commands related to remote job entry (RJE)
- Invoking another CL or interactive entry—might reach lock resource limit.

Commands, statements, and operations that you cannot use in trigger programs:

A trigger program cannot include some commands, statements, and operations.

The system returns an exception if you use these:

- The commitment definition associated with the insert, update, delete, or read operation that called the trigger does not allow the COMMIT operation. The COMMIT operation *is* allowed for any other commitment definition in the job.
- The commitment definition associated with the insert, update, delete, or read operation that called the trigger does not allow the ROLLBACK operation. The ROLLBACK operation *is* allowed for any other commitment definition in the job.
- The SQL CONNECT, DISCONNECT, SET CONNECTION, and RELEASE statements *are not* allowed.
- The commitment definition associated with the insert, update, delete, or read operation that called the trigger does not allow the ENDCMTCTL CL command. The ENDCMTCTL CL command *is* allowed for any other commitment definition in the job.
- An attempt to add a local API commitment resource (QTNADDCR) to the same commitment definition associated with the insert, update, delete, or read operation that called the trigger.
- An attempt to do any input/output operation to a file that a trigger program has opened with *SHARE and is the file that caused the trigger program to be called.
- The called trigger program that uses the same commitment definition as the insert, update, delete, or read operation that called the trigger and that already has an existing remote resource. However, the system puts the entire transaction into a rollback-required state:
  - If the trigger program fails and signals an escape message AND
  - Any remote resource was updated during the nonprimary commit cycle for either a system other than System i or for the system that is at a pre-Version 3 Release 2 level.
- The trigger program can add a remote resource to the commitment definition associated with the insert, update, delete, or read operation that called the trigger. This allows for LU62 remote resources (protected conversation) and DFM remote resources (DDM file open), but not DRDA® remote resources.
- If a failure occurs when changing a remote resource from a trigger program, the trigger program must end by signalling an escape message. This allows the system to ensure that the entire transaction, for all remote locations, properly rolls back. If the trigger program does not end with an escape message, the databases on the various remote locations might become inconsistent.
- A commit lock level of the application program is passed to the trigger program. Run the trigger program under the same lock level as the application program.
- The trigger program and application program can run in the same or different activation groups. Compile the trigger program with ACTGRP(*CALLER) to achieve consistency between the trigger program and the application program.
- A trigger program calls other programs or it can be nested (that is, a statement in a trigger program causes the calling of another trigger program). In addition, a trigger program itself can call a trigger program. The maximum trigger nested level for insert, update, delete, or read is 200. When the trigger program runs under commitment control, the following situations will result in an error:
  - Any update of the same record that has already been changed by the change operation or by an operation in the trigger program.
  - Conflicting operations on the same record within one change operation. For example, the change operation inserts a record, then the record is deleted by the trigger program.

  Notes:
    1. If the change operation is not running under commitment control, the system always protects the change operation. However, the system does not monitor updating the same record within the trigger program.
    2. The ALWREPCHG(*NO|YES) parameter of the Add Physical File Trigger (ADDPFTRG) command controls repeated changes under commitment control. Changing from the default value to ALWREPCHG(*YES) allows the same record or updated record associated with the trigger program to repeatedly change.

- The Allow Repeated Change ALWREPCHG(*YES) parameter on the Add Physical File Trigger (ADDPFTRG) command also affects trigger programs defined to be called before insert and update database operations. If the trigger program updates the new record in the trigger buffer and ALWREPCHG(*YES) is specified, the actual insert or update operation on the associated physical file uses the modified new record image. This option can be helpful in trigger programs that are designed for data validation and data correction. Because the trigger program receives physical file record images (even for logical files), it can change any field of that record image.
- The trigger program is called for each row that is changed in or read from the physical file.
- If the physical file or the dependent logical file is opened for insert SEQONLY(*YES) processing, and the physical file has an insert trigger program associated with it, the system changes the open to SEQONLY(*NO) so it can call the trigger program for each row that is inserted.

Monitoring the use of trigger programs:

DB2 for i5/OS provides the capability to associate trigger programs with database files. Trigger-program capability is common across the industry for high-function database managers.

When you associate a trigger program with a database file, you specify when the trigger program runs. For example, you can set up the customer order file to run a trigger program whenever a new record is added to the file. When the customer's outstanding balance exceeds the credit limit, the trigger program can print a warning letter to the customer and send a message to the credit manager.

Trigger programs are a productive way both to provide application functions and to manage information. Trigger programs also provide the ability for someone with devious intentions to create a "Trojan horse" on your system. A destructive program can be sitting and waiting to run when a certain event occurs in a database file on your system.

Note: In history, the Trojan horse was a large hollow wooden horse that was filled with Greek soldiers. After the horse was introduced within the walls of Troy, the soldiers climbed out of the horse and fought the Trojans. In the computer world, a program that hides destructive functions is often called a Trojan horse.

When your system ships, the ability to add a trigger program to a database file is restricted. If you are managing object authority carefully, the typical user will not have sufficient authority to add a trigger program to a database file. (Appendix D in the *Security Reference* book tells the authority that is required or all commands, including the Add Physical File Trigger (ADDPFTRG) command.

You can use the Print Trigger Programs (PRTTRGPGM) command to print a list of all the trigger programs in a specific library or in all libraries. The following example shows the report:

```
                            Trigger Programs (Full Report)

Specified library . . . . . . :   CUSTLIB
                            Trigger      Trigger      Trigger      Trigger      Trigger
Library      File         Library      Program      Time         Event        Condition
CUSTLIB      MB106        ARPGMLIB     INITADDR     Before       Update       Always
CUSTLIB      MB107        ARPGMLIB     INITNAME     Before       Update       Always
```

You can use the initial report as a base to evaluate any trigger programs that already exist on your system. Then, you can print the changed report regularly to see whether new trigger programs have been added to your system.

When you evaluate trigger programs, consider the following questions:
- Who created the trigger program? You can use the Display Object Description (DSPOBJD) command to determine this.

- What does the program do? You will have to look at the source program or talk to the program creator to determine this. For example, does the trigger program check to see who the user is? Perhaps the trigger program is waiting for a particular user (QSECOFR) in order to gain access to system resources.

After you have established a base of information, you can print the changed report regularly to monitor new trigger programs that have been added to your system. The following example shows the changed report:

```
                              Trigger Programs (Changed Report)
Specified library  . . . . . . :   LIBX
Last changed report  . . . . . :   07/01/21  14:33:37
                          Trigger       Trigger       Trigger       Trigger       Trigger
Library      File        Library       Program       Time          Event         Condition
INVLIB       MB108       INVPGM        NEWPRICE      After         Delete        Always
INVLIB       MB110       INVPGM        NEWDSCNT      After         Delete        Always
```

Trigger and application programs that are under commitment control:

Here are the considerations for trigger and application programs that are running under commitment control.

When the trigger program and the application program run under the same commitment definition, a failure of the trigger program causes the rollback of all statements that are associated with the trigger program. This includes any statement in a nested trigger program. The originating change operation also rolls back. This requires the trigger program to signal an exception when it encounters an error.

When the trigger program and the application program run under different commitment definitions, the COMMIT statements in the application program only affect its own commitment definition. The programmer must commit the changes in the trigger program by issuing the COMMIT statement.

When insert or update record operations are performed under commitment control, the detection of any specific duplicate key errors is deferred until the logical end of the operation, to allow for the possibility that such errors have been resolved by that time. In the case of a trigger program running in the same commitment definition as its calling program, the logical end of the operation occurs after the single or blocked insert, update, or delete record operation is performed by the calling program, and control returns from any called before or after trigger programs. As a result, duplicate key errors are not detectable in trigger programs that use the same commitment definition as the insert, update, or delete record operation that called the trigger programs.

Trigger and application programs that are not under commitment control:

Here are the considerations for trigger and application programs that are not running under commitment control.

If both programs do not run under commitment control, any error in a trigger program leaves files in the state that exists when the error occurs. No rollback occurs.

If the trigger program does not run under commitment control and the application program does run under commitment control, all changes from the trigger program are committed when either:
- A commit operation is performed in the trigger program.
- The activation group ends. In the normal case, an implicit commit is performed when the activation group ends. However, if an abnormal system failure occurs, a rollback is performed.

Trigger program error messages:

If a failure occurs when the trigger program is running, the program must signal an appropriate escape message before exiting. Otherwise, the application assumes that the trigger program ran successfully.

The message can be the original message that is signalled from the system or a message that is created by the trigger program creator.

Example: Trigger program:

This example shows an external trigger program that is written in ILE C with embedded SQL.

For more trigger program examples, see the IBM Redbooks™ publication Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database™ for iSeries™ .

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 290.

```c
#include "string.h"
#include "stdlib.h"
#include "stdio.h"
#include <recio.h>
#include <xxcvt.h>
#include "qsysinc/h/trgbuf"        /* Trigger input parameter       */
#include "lib1/csrc/msghand1"      /* User defined message handler  */
/*******************************************************************/
/* This is a trigger program which is called whenever there is an   */
/* update to the EMPLOYEE table. If the employee's commission is    */
/* greater than the maximum commission, this trigger program will   */
/* increase the employee's salary by 1.04 percent and insert into   */
/* the RAISE table.                                                 */
/*                                                                  */
/* The EMPLOYEE record information is passed from the input parameter*/
/* to this trigger program.                                         */
/*******************************************************************/

Qdb_Trigger_Buffer_t *hstruct;
char *datapt;

    /****************************************************/
    /* Structure of the EMPLOYEE record which is used to */
    /* store the old or the new record that is passed to */
    /* this trigger program.                            */
    /*                                                  */
    /* Note : You must ensure that all the numeric fields */
    /*        are aligned at 4 byte boundary in C.      */
    /*        Used either Packed struct or filler to reach */
    /*        the byte boundary alignment.              */
    /****************************************************/

_Packed struct rec{
        char  empn[6];
  _Packed struct { short fstlen ;
                char  fstnam[12];
             } fstname;
        char  minit[1];
_Packed struct { short lstlen;
                char lstnam[15];
             } lstname;
      char  dept[3];
      char  phone[4];
      char  hdate[10];
      char  jobn[8];
      short edclvl;
      char  sex1[1];
      char  bdate[10];
      decimal(9,2) salary1;
```

```
            decimal(9,2) bonus1;
            decimal(9,2) comm1;
            } oldbuf, newbuf;
EXEC SQL INCLUDE SQLCA;

main(int argc, char **argv)
{
int i;
int obufoff;                        /* old buffer offset         */
int nuloff;                         /* old null byte map offset  */
int nbufoff;                        /* new buffer offset         */
int nul2off;                        /* new null byte map offset  */
short work_days  = 253;             /* work days during in one year */
decimal(9,2) commission = 2000.00;  /* cutoff to qualify for     */
decimal(9,2) percentage = 1.04;     /* raised salary as percentage */
char raise_date[12] = "1982-06-01"; /* effective raise date      */


struct {
        char   empno[6];
        char   name[30];
        decimal(9,2) salary;
        decimal(9,2) new_salary;
        } rpt1;

  /*****************************************************/
  /* Start to monitor any exception.                 */
  /*****************************************************/

  _FEEDBACK fc;
  _HDLR_ENTRY hdlr = main_handler;
                    /**************************************/
                    /* Make the exception handler active.  */
                    /**************************************/
  CEEHDLR(&hdlr, NULL, &fc);
                    /**************************************/
                    /* Ensure exception handler OK        */
                    /**************************************/
  if (fc.MsgNo != CEE0000)
    {
      printf("Failed to register exception handler.\n");
      exit(99);
    };

  /*****************************************************/
  /* Move the data from the trigger buffer to the local */
  /* structure for reference.                          */
  /*****************************************************/

hstruct = (Qdb_Trigger_Buffer_t *)argv[1];
datapt  = (char *) hstruct;

obufoff = hstruct ->Old_Record_Offset;      /* old buffer   */
memcpy(&oldbuf,datapt+obufoff,; hstruct->Old_Record_Len);

nbufoff = hstruct ->New_Record_Offset;      /* new buffer   */
memcpy(&newbuf,datapt+nbufoff,; hstruct->New_Record_Len);

EXEC SQL WHENEVER SQLERROR  GO TO ERR_EXIT;

  /*****************************************************/
  /* Set the transaction isolation level to the same as */
  /* the application based on the input parameter in the */
  /* trigger buffer.                                   */
  /*****************************************************/

if(strcmp(hstruct->Commit_Lock_Level,"0") == 0)
     EXEC SQL SET TRANSACTION ISOLATION LEVEL NONE;
```

```
else{
  if(strcmp(hstruct->Commit_Lock_Level,"1") == 0)
      EXEC SQL SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED, READ
              WRITE;
  else {
     if(strcmp(hstruct->Commit_Lock_Level,"2") == 0)
       EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  else
     if(strcmp(hstruct->Commit_Lock_Level,"3") == 0)
       EXEC SQL SET TRANSACTION ISOLATION LEVEL ALL;
  }
}

 /*******************************************************/
 /* If the employee's commission is greater than maximum */
 /* commission, then increase the employee's salary      */
 /* by 1.04 percent and insert into the RAISE table.     */
 /*******************************************************/

if (newbuf.comm1 >= commission)
{
  EXEC SQL  SELECT EMPNO, EMPNAME, SALARY
        INTO :rpt1.empno, :rpt1.name, :rpt1.salary
        FROM TRGPERF/EMP_ACT
        WHERE EMP_ACT.EMPNO=:newbuf.empn ;

  if (sqlca.sqlcode == 0) then
  {
     rpt1.new_salary = salary * percentage;
     EXEC SQL INSERT INTO TRGPERF/RAISE VALUES(:rpt1);
  }
  goto finished;
}
err_exit:
  exit(1);

  /* All done  */
finished:
  return;
}  /* end of main line  */
/*******************************************************************/
/*    INCLUDE NAME : MSGHAND1                                      */
/*                                                                 */
/*    DESCRIPTION  : Message handler to signal an exception to     */
/*                   the application to inform that an             */
/*                   error occured in the trigger program.        */
/*                                                                 */
/*    NOTE : This message handler is a user defined routine.      */
/*                                                                 */
/*******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#include <leawi.h>

#pragma linkage (QMHSNDPM, OS)
void QMHSNDPM(char *,          /* Message identifier            */
             void *,          /* Qualified message file name   */
             void *,          /* Message data or text          */
             int,             /* Length of message data or text */
             char *,          /* Message type                  */
             char *,          /* Call message queue            */
             int,             /* Call stack counter            */
             void *,          /* Message key                   */
             void *,          /* Error code                    */
             ...);            /* Optionals:
                                   length of call message queue
```

```
                                    name
                                    Call stack entry qualification
                                    display external messages
                                    screen wait time               */
/*****************************************************************/
/********  This is the start of the exception handler function.   */
/*****************************************************************/
  void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc,
                    _FEEDBACK *new)
   {
                            /***************************************/
                            /* Initialize variables for call to    */
                            /* QMHSNDPM.                            */
                            /* User must create a message file and  */
                            /* define a message ID to match the     */
                            /* following data.                      */
                            /***************************************/
  char      message_id[7] = "TRG9999";
  char      message_file[20] = "MSGF      LIB1        ";
  char      message_data[50] = "Trigger error                " ;
  int       message_len = 30;
  char      message_type[10] = "*ESCAPE   ";
  char      message_q[10] = "_C_pep     ";
  int       pgm_stack_cnt = 1;
  char      message_key[4];

                            /***************************************/
                            /* Declare error code structure for    */
                            /* QMHSNDPM.                            */
                            /***************************************/
struct error_code {
  int bytes_provided;
  int bytes_available;
  char message_id[7];
} error_code;

error_code.bytes_provided = 15;
                            /***************************************/
                            /* Set the error handler to resume and */
                            /* mark the last escape message as     */
                            /* handled.                            */
                            /***************************************/
*rc = CEE_HDLR_RESUME;
                            /***************************************/
                            /* Send my own *ESCAPE message.        */
                            /***************************************/
QMHSNDPM(message_id,
         &message_file,
         &message_data,
          message_len,
          message_type,
          message_q,
          pgm_stack_cnt,
         &message_key,
         &error_code );
                            /***************************************/
                            /* Check that the call to QMHSNDPM     */
                            /* finished correctly.                 */
                            /***************************************/
if (error_code.bytes_available != 0)
   {
     printf("Error in QMHOVPM : %s\n", error_code.message_id);
   }
}
```

Trigger buffer sections:

The trigger buffer has two logical sections: a static section and a variable section.

The static section occupies (in decimal) offset 0 through 95. It contains the following items:
- A trigger template that contains the physical file name, member name, trigger event, trigger time, commit lock level, and CCSID of the current record and relative record number.
- Offsets and lengths of the record areas and null byte maps.

The variable section contains areas for old record, old null byte map, new record, and new null byte map.

The following table provides a summary of the fields in the trigger buffer.

Offset			
Dec	Hex	Type	Field
0	0	CHAR(10)	Physical file name
10	A	CHAR(10)	Physical file library name
20	14	CHAR(10)	Physical file member name
30	1E	CHAR(1)	Trigger event
31	1F	CHAR(1)	Trigger time
32	20	CHAR(1)	Commit lock level
33	21	CHAR(3)	Reserved
36	24	BINARY(4)	CCSID of data
40	28	BIN(4)	Relative Record Number
44	2C	CHAR(4)	Reserved
48	30	BINARY(4)	Original record offset
52	34	BINARY(4)	Original record length
56	38	BINARY(4)	Original record null byte map offset
60	3C	BINARY(4)	Original record null byte map length
64	40	BINARY(4)	New record offset
68	44	BINARY(4)	New record length
72	48	BINARY(4)	New record null byte map offset
76	4C	BINARY(4)	New record null byte map length
80	50	CHAR(*)	Reserved
*	*	CHAR(*)	Original record
*	*	CHAR(*)	Original record null byte map
*	*	CHAR(*)	New record
*	*	CHAR(*)	New record null byte map

Trigger buffer field descriptions:

Fields in the buffer field are described in alphabetic order.

CCSID of data
> The CCSID of the data in the new or the original records. The data is converted to the job CCSID by the database. SBCS data is converted to the single byte associated CCSID. DBCS data is converted to the double byte associated CCSID.

Commit lock level
> The commit lock level of the current application program. The possible values are:

'0'   *NONE

'1'   *CHG

'2'   *CS

'3'   *ALL

File library name

The name of the library in which the database file resides.

File member name

The name of the database file member.

File name

The name of the physical file being changed or read.

New record

A copy of the record that is being inserted or updated in a physical file as a result of the change operation. The new record only applies to the insert or update operations.

New record length

The maximum length is 32 766 bytes.

New record null byte map

This structure contains the NULL value information for each field of the new record. Each byte represents one field. The possible values for each byte are:

'0'   Not NULL

'1'   NULL

New record null byte map length

The length is equal to the number of fields in the physical file.

New record null byte map offset

The location of the null byte map of the new record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the new value of the record does not apply to the change operation, for example, a delete operation.

New record offset

The location of the new record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the new value of the record does not apply to the change operation, for example, a delete operation.

Original record

A copy of the original physical record before being updated, deleted, or read. The original record applies only to update, delete, and read operations.

Original record length

The maximum length is 32 766 bytes.

Original record null byte map

This structure contains the NULL value information for each field of the original record. Each byte represents one field. The possible values for each byte are:

'0'   Not NULL

'1'   NULL

Original record null byte map length

The length is equal to the number of fields in the physical file.

Original record null byte map offset

The location of the null byte map of the original record. The offset value is from the beginning of

the trigger buffer. This field is not applicable if the original value of the record does not apply to the change operation, for example, an insert operation.

Original record offset

The location of the original record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the original value of the record does not apply to the operation; for example, an insert operation.

Relative Record Number

The relative record number of the record to be updated or deleted (*BEFORE triggers) or the relative record number of the record which was inserted, updated, deleted, or read(*AFTER triggers).

Trigger event

The event that caused the trigger program to be called. The possible values are:

'1'   Insert operation

'2'   Delete operation

'3'   Update operation

'4'   Read operation

Trigger time

The time, relative to the operation on the database file, when the trigger program is called. The possible values are:

'1'   After the change or read operation

'2'   Before the change operation

'3'   Instead of the change operation

Adding triggers

You can add a trigger to a specific database file.

To add a trigger, follow these steps:
1. Ensure that you have the proper authority and the file has the proper data capabilities.
2. Use one of the following methods to associate the trigger program with a specific database file:
   - Use System i Navigator to create a new file or to edit the properties of an existing file.
   - Use the Add Physical File Trigger (ADDPFTRG) command.
   - Use the CREATE TRIGGER SQL statement.

Note: If the trigger program resides in QTEMP library, the trigger program cannot be associated with a physical file.

After you have created the association between the trigger program and the file, the system calls the trigger program when a change or read operation is initiated against the database file, a member of the file, and any logical file created over the physical file.

You can associate a maximum of 300 triggers with one database file. Each insert, delete, or update operation can call multiple triggers before and after the operation occurs. Each read operation can call multiple triggers after the operation occurs. This topic shows how to add a trigger to a file.

You can associate a maximum of three triggers with one logical file. That is, you can create one INSTEAD OF trigger per insert, update, or delete operation. The trigger is called instead of performing the operation.

The number of triggers called after a read operation that is issued by a query might not be equal to the number of records that are actually returned. This is because the query might have read a different number of records, causing a trigger to be called for each read operation, before returning the correct number of records.

An SQL update operation involves a simultaneous read operation followed by a write operation. Read triggers are not run for SQL update operations. An update trigger should be specified to cover this read operation followed by a write operation.

Required authorities and data capabilities for triggers

To add triggers, you must have the following required authorities:
- Object management or alter authority to the file
- Object operational authority to the file
- Read authority to the file
- Update authority and object operational authority to the file if ADDPFTRG ALWREPCHG(*YES) is specified
- Execute authority to the file's library
- Execute authority to the trigger program
- Execute authority to the trigger program's library (This authority is only required for adding triggers.)

The file must have appropriate data capabilities before you add a trigger:
- CRTPF ALWUPD(*NO) conflicts with *UPDATE Trigger
- CRTPF ALWDLT(*NO) conflicts with *DELETE Trigger
  Related reference
  Add Physical File Trigger (ADDPFTRG) command

Displaying triggers
You can use the Display File Description (DSPFD) command to display a list of triggers that are associated with a file. Specify TYPE(*TRG) or TYPE(*ALL) to get the list.

The DSPFD command provides the following information:
- The number of trigger programs
- The trigger name and library
- The trigger status
- The trigger program names and libraries
- The trigger events
- The trigger times
- The trigger update conditions
- The trigger type
- The trigger mode
- The trigger orientation
- The trigger creation date/time
- The number of trigger update columns
- List of trigger update columns
  Related reference
  Display File Description (DSPFD) command

Removing triggers

You can remove triggers using the Remove Physical File Trigger (RMVPFTRG) command, the SQL DROP TRIGGER statement, or System i Navigator.

Use the RMVPFTRG command to remove the association of a file and the trigger program. After you remove the association, the system takes no action when a change or read operation occurs to the physical file. The trigger program, however, remains on the system.

> **Related concepts**
>
> Getting started with System i Navigator
>
> SQL programming
>
> **Related reference**
>
> Remove Physical File Trigger (RMVPFTRG) command

Enabling or disabling physical file triggers

You can enable or disable triggers using the Change Physical File Trigger (CHGPFTRG) command or System i Navigator.

Use the CHGPFTRG command to enable or disable named triggers or to enable or disable all triggers for a file. Disabling triggers causes the trigger program not to be called when a change operation occurs to the physical file. Enabling triggers causes the trigger program again to be called when a change operation occurs to the physical file. You can also enable or disable triggers using System i Navigator.

Triggers on logical files cannot be enabled or disabled.

> **Related concepts**
>
> Getting started with System i Navigator
>
> SQL programming
>
> **Related reference**
>
> Change Physical File Trigger (CHGPFTRG) command

Triggers and their relationship to CL commands

Triggers interact with CL commands in several ways.

Save/Restore Base File (SAVOBJ/RSTOBJ)

- The Save/Restore function will not search for the trigger program during save/restore time. It is your responsibility to manage the program. During run time, if the system has not restored the trigger program, the system returns a hard error with the trigger program name, physical file name, and trigger event.
- If the entire library (*ALL) is saved and the file and all trigger programs are in the same library and they are restored to a different library, then all the trigger program names are changed in the file to reflect the new library.

Save/Restore Trigger Program (SAVOBJ/RSTOBJ)

- If you restore the trigger program in a different library, the change operation fails because the trigger program is not in the original library. The error returns the trigger program name, physical file name, and trigger event information.

  There are two ways to recover in this situation:
  - Restore the trigger program to the same library
  - Create a new trigger program with the same name in the new library

Delete File (DLTF)

- If a file is deleted, the association between this file and its trigger programs are removed. For system triggers, the trigger programs remain on the system.

Copy File (CPYF)

- If a to-file associates with an insert trigger, each inserted record calls the trigger program.
- If a to-file associates with a delete trigger program and the CPYF command specifies MBROPT(*REPLACE), the copy operation fails.
- Copy with CREATE(*YES) does not propagate the trigger information

Create Duplicate Object (CRTDUPOBJ)

If TRG(*NO) is specified, triggers will not be duplicated in the new file. If TRG(*YES) is specified, triggers will be duplicated. The following rules describe how triggers are duplicated:

- When a physical file and its trigger program are originally in the same library, the trigger program library will always be changed to the new library, even if the trigger program does not exist in the new library. In addition, the following rules apply:
  - If the CRTDUPOBJ command is duplicating both the physical file and its trigger program to a new library, then the new trigger program will be associated with the new physical file.
  - If the CRTDUPOBJ command is duplicating only the physical file, then the trigger program with the same program name in the TO library will be associated with the new physical file. This is true even if there is no trigger program by that name in the TO library. The library of the trigger program will be changed.
  - If the CRTDUPOBJ command is duplicating only the trigger program, then the new trigger program will not be associated with any physical files.
- When a physical file and its trigger program are originally in different libraries:
  - The old trigger program will be associated with the new physical file. Even if the new physical file is duplicated to the same library as the trigger program, the old trigger program will still be associated with the new physical file.
- A trigger program cannot be added if the program is in the QTEMP library. For database files, the CRTDUPOBJ command attempts to locate the trigger program in the TO library. If the CRTDUPOBJ command is used with QTEMP specified as the new library, CRTDUPOBJ attempts to create as much of the object as possible. The file is created, but the trigger cannot be added, so the file remains in QTEMP without a member.

Clear Physical File Member (CLRPFM)

- If the physical file is associated with a delete trigger, the CLRPFM operation fails.

Initialize Physical File Member (INZPFM)

- If the physical file is associated with an insert trigger, the INZPFM operation fails.

FORTRAN Force-End-Of-Data (FEOD)

- If the physical file is associated with a delete trigger, the FEOD operation fails.

Apply Journaled Changes or Remove Journaled Changes (APYJRNCHG/RMVJRNCHG)

- If the physical file is associated with any type of trigger, the APYJRNCHG and RMVJRNCHG operations do not call the trigger program. Therefore, you must make sure to have all the files within the trigger program journaled. Then, when using the APYJRNCHG or RMVJRNCHG command, make sure to specify all of these files. This ensures that all the physical file changes for the application program and the trigger programs are consistent.

  Note: If any trigger program functions do not relate to database files and cannot be explicitly journaled, send journal entries to record relevant information. Use the Send Journal Entry (SNDJRNE) command or the Send Journal Entry (QJOSJRNE) API. Use this information during database file recovery to ensure consistency.

Triggers and their relationship to referential integrity

A physical file can have both triggers and referential constraints associated with it. The running order among trigger actions and referential constraints depends on the constraints and triggers that are associated with the file.

In some cases, the system evaluates referential constraints before the system calls an after trigger program. This is the case with constraints that specify the RESTRICT rule.

In some cases, all statements in the trigger program, including nested trigger programs, run before the constraint is applied. This is true for NO ACTION, CASCADE, SET NULL, and SET DEFAULT referential constraint rules. When you specify these rules, the system evaluates the file's constraints based on the nested results of trigger programs. For example, an application inserts employee records into an EMP file that has a constraint and trigger:

- The referential constraint specifies that the department number for an inserted employee record to the EMP file must exist in the DEPT file.
- Whenever an insert to the EMP file occurs, the trigger program checks if the department number exists in the DEPT file. The trigger program then adds the number if it does not exist.

When the insertion to the EMP file occurs, the system calls the trigger program first. If the department number does not exist in the DEPT file, the trigger program inserts the new department number into the DEPT file. Then the system evaluates the referential constraint. In this case, the insertion is successful because the department number exists in the DEPT file.

There are some restrictions when both a trigger and referential constraint are defined for the same physical file:

- If a delete trigger associates with a physical file, that file must not be a dependent file in a referential constraint with a delete rule of CASCADE.
- If an update trigger associates with a physical file, no field in this physical file can be a foreign key in a referential constraint with a delete rule of SET NULL or SET DEFAULT.

If failure occurs during either a trigger program or referential constraint validation, all trigger programs associated with the change operation roll back if all the files run under the same commitment definition. The referential constraints are guaranteed when all files in the trigger program and the referential integrity network run under the same commitment definition. If you open the files without commitment control or in a mixed scenario, undesired results might occur.

You can use triggers to enforce referential constraints and business rules. For example, you can use triggers to simulate the update cascade constraints on a physical file. However, you would not have the same functional capabilities as provided by the constraints that the system referential integrity functions define. You might lose the following referential integrity advantages if you define them with triggers:

- Dependent files might contain rows that violate one or more referential constraints that put the constraint into check pending but still allow file operations.
- The ability to inform users when the system places a constraint in check pending.
- When an application runs under COMMIT(*NONE) and an error occurs during a cascaded delete, the database rolls back all changes.
- While saving a file that is associated with a constraint, the database network saves all dependent files in the same library.

Database distribution

DB2 Multisystem (i5/OS option 27) provides a simple and direct method of distributing a database file over multiple systems in a loosely coupled environment.

DB2 Multisystem allows users on distributed systems to have real-time query and update access to a distributed database as if it existed totally on their particular systems. DB2 Multisystem places new records on the appropriate system based on a user-defined key field or fields. DB2 Multisystem chooses a system on the basis of either a system-supplied or user-defined hashing algorithm.

Query performance is improved by a factor approaching the number of nodes in the environment. For example, a query against a database distributed over four systems runs in approximately one quarter of the time. However, performance can vary greatly when queries involve joins and grouping. Performance is also influenced by the balance of the data across the multiple nodes. Multisystem runs the query on each system concurrently. DB2 Multisystem can significantly reduce query time on very large databases.

Related concepts

DB2 Multisystem

Double-byte character set considerations

A double-byte character set (DBCS) is a character set that represents each character with 2 bytes. The database on the i5/OS operating system supports DBCS.

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.

DBCS field data types

There are two general kinds of double-byte character set (DBCS) data: bracketed-DBCS data and graphic (nonbracketed) DBCS data.

Bracketed-DBCS data is preceded by a DBCS shift-out character and followed by a DBCS shift-in character. Graphic-DBCS data is not surrounded by shift-out and shift-in characters. The application program might require special processing to handle bracketed-DBCS data that would not be required for graphic-DBCS data.

The specific DBCS data types (specified in position 35 on the DDS coding form.) are:

Entry	Meaning
O	DBCS-open: A character string that contains both single-byte and bracketed double-byte data.
E	DBCS-either: A character string that contains either all single-byte data or all bracketed double-byte data.
J	DBCS-only: A character string that contains only bracketed double-byte data.
G	DBCS-graphic: A character string that contains only nonbracketed double-byte data.

Note: Files containing DBCS data types can be created on a single-byte character set (SBCS) system. Files containing DBCS data types can be opened and used on a SBCS system, however, coded character set identifier (CCSID) conversion errors can occur when the system tries to convert from a DBCS or mixed CCSID to a SBCS CCSID. These errors do not occur if the job CCSID is 65535.

DBCS constants

A constant identifies the actual character string to be used. The character string is enclosed in single quotation marks and a string of DBCS characters is surrounded by the DBCS shift-out and shift-in characters (represented by the characters < and > in the following examples). A DBCS-graphic constant is preceded by the character *G*.

The types of DBCS constants are:

Type   Example

DBCS-Only

'<A1A2A3>'

DBCS-Open

'<A1A2A3>BCD'

DBCS-Graphic

G'<A1A2A3>'

DBCS field mapping considerations

The table shows what types of data mapping are valid between physical and logical files for double-byte character set (DBCS) fields.

Physical file data type	Logical file data type								
	Character	Hexa-decimal	DBCS- open	DBCS-either	DBCS- only	DBCS-graphic	UCS2-graphic	UTF-8	UTF-16
Character	Valid	Valid	Valid	Not valid	Not valid	Not valid	Valid	Valid	Valid
Hexadecimal	Valid	Valid	Valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid
DBCS-open	Not valid	Valid	Valid	Not valid	Not valid	Not valid	Valid	Valid	Valid
DBCS-either	Not valid	Valid	Valid	Valid	Not valid	Not valid	Not valid[1]	Not valid[1]	Not valid[1]
DBCS-only	Valid	Valid	Valid	Valid	Valid	Valid	Not valid[1]	Not valid[1]	Not valid[1]
DBCS-graphic	Not valid	Not valid	Valid	Valid	Valid	Valid	Valid	Valid	Valid
UCS2-graphic	Valid	Not valid	Valid	Not valid[1]	Not valid[1]	Valid	Valid	Valid	Valid
UTF-8	Valid	Not valid	Valid	Not valid[1]	Not valid[1]	Valid	Valid	Valid	Valid
UTF-16	Valid	Not valid	Valid	Not valid[1]	Not valid[1]	Valid	Valid	Valid	Valid

Note: In the table, [1] indicates that these mappings are not supported because of the possibility of substitution characters appearing after conversion.

DBCS field concatenation

When fields are concatenated, the data types can change (the resulting data type is automatically determined by the system). Double-byte character set (DBCS) field concatenation follows these rules.

* The operating system 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 (H) fields, the resulting data type is DBCS open (O).
  - If the concatenation contains two or more DBCS open (O) fields, the resulting data type is a variable-length DBCS open (O) field.
  - If the concatenation contains one or more variable-length fields of any data type, the resulting data type is variable length.
  - A DBCS-graphic (G) field can be concatenated only to another DBCS-graphic field. The resulting data type is DBCS-graphic (G).
  - A UCS2-graphic (G) field can be concatenated to another UCS2-graphic field, a UTF-8 character field, or a UTF-16 graphic field. The resulting data type is UTF-16 if one of the operands is UTF-16, UTF-8 if one of the operands is UTF-8 and no operands are UTF-16, and otherwise UCS-2.
  - A UTF-8 character (A) field can be concatenated with another UTF-8 field, a UTF-16 field, or a UCS-2 field. The resulting data type is UTF-16 if one of the operands is UTF-16, UTF-8 if one of the operands is UTF-8 and no operands are UTF-16, and otherwise UCS-2.

– A UTF-16 graphic (G) field can be concatenated with another UTF-16 field, a UTF-8 field, or a UCS-2 field. The resulting data type is UTF-16 if one of the operands is UTF-16, UTF-8 if one of the operands is UTF-8 and no operands are UTF-16, and otherwise UCS-2.

- 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 the concatenated field is character (A), DBCS-open (O), or DBCS-only (J), its total length cannot exceed 32,766 bytes (32,740 bytes if the field is variable length).

  The length of DBCS-graphic (G) fields is expressed as the number of double-byte *characters* (the actual length is twice the number of characters); therefore, the total length of the concatenated field cannot exceed 16,383 characters (16,370 characters if the field is variable length).

- 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 bracketed-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 concatenation contains one or more hexadecimal fields, 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.

3. Resulting data types for concatenated DBCS fields might differ when using The Open Query File (OPNQRYF) command.

Related concepts

"Using concatenation with DBCS fields" on page 288
When double-byte character set (DBCS) fields are included in a concatenation through the Open Query File (OPNQRYF) command, the resulting data type is the same as the data type of the concatenated field in a logical file, with some slight variations.

DBCS field substring operations

A substring operation allows you to use part of a field or constant in a logical file.

For bracketed-DBCS data types, the starting position and the length of the substring refer to the number of *bytes*; therefore, each double-byte character counts as two positions. For the DBCS-graphic (G) data type, the starting position and the length of the substring refer to the number of *characters*; therefore, each double-byte character counts as one position.

Comparing DBCS fields in a logical file

When you compare two fields or compare a field and constants, fixed-length fields can be compared to variable-length fields if the types are compatible. The table shows the valid comparisons for double-byte character set (DBCS) fields in a logical file.

Table 50. Valid comparisons for DBCS fields in a logical file

	Any numeric	Character	Hexadecimal	DBCS-open	DBCS-either	DBCS-only	DBCS-graphic	UCS2-graphic	UTF-8	UTF-16	Date	Time	Timestamp
Any numeric	Valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid
Character	Not valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid	Valid	Valid	Not valid	Not valid	Not valid
Hexadecimal	Not valid	Valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid
DBCS-open	Not valid	Valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Valid	Valid	Not valid	Not valid	Not valid
DBCS-either	Not valid	Valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid
DBCS-only	Not valid	Not valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid

Table 50. Valid comparisons for DBCS fields in a logical file (continued)

	Any numeric	Character	Hexadecimal	DBCS-open	DBCS-either	DBCS-only	DBCS-graphic	UCS2-graphic	UTF-8	UTF-16	Date	Time	Timestamp
DBCS-graphic	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Valid	Not valid	Valid	Valid	Not valid	Not valid	Not valid
UCS2-graphic	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Valid	Valid	Valid	Not valid	Not valid	Not valid
UTF-8	Not valid	Valid	Not valid	Valid	Not valid	Not valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid
UTF-16	Not valid	Valid	Not valid	Valid	Not valid	Not valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid
Date	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Valid	Not valid	Not valid
Time	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Valid	Not valid
Timestamp	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Valid

Using DBCS fields in the Open Query File (OPNQRYF) command

Double-byte character set (DBCS) fields can be used for the wildcard, concatenation, and sort sequence functions of the Open Query File (OPNQRYF) command.

Using the wildcard function with DBCS fields

The wildcard (%WLDCRD) function with a double-byte character set (DBCS) field differs depending on whether the function is used with a bracketed-DBCS field or a DBCS-graphic field.

When using the wildcard function with a bracketed-DBCS field, both single-byte and double-byte wildcard 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.

When using the wildcard function with a DBCS-graphic field, only double-byte wildcard values (asterisk and underline) are allowed. The following special rules apply:

- A double-byte underline refers to one double-byte character.
- A double-byte asterisk refers to any number of double-byte characters.

Comparing DBCS fields through the Open Query File (OPNQRYF) command

When you compare two fields or constants, fixed-length fields can be compared to variable-length fields if the types are compatible. The table shows the valid comparisons for double-byte character set (DBCS) fields through the Open Query File (OPNQRYF) command.

Table 51. Valid comparisons for DBCS fields through the OPNQRYF command

	Any numeric	Character	Hexa-decimal	DBCS- open	DBCS- either	DBCS- only	DBCS-graphic	UCS2- graphic	Date	Time	Time- stamp
Any numeric	Valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid
Character	Not valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Valid	Valid	Valid	Valid
Hexa- decimal	Not valid	Valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Valid	Valid	Valid
DBCS- open	Not valid	Valid	Valid	Valid	Valid	Valid	Not valid	Valid	Valid	Valid	Valid
DBCS- either	Not valid	Valid	Valid	Valid	Valid	Valid	Not valid	Valid	Valid	Valid	Valid
DBCS- only	Not valid	Not valid	Valid	Valid	Valid	Valid	Not valid	Valid	Not valid	Not valid	Not valid
DBCS- graphic	Not valid	Not valid	Not valid	Not valid	Not valid	Not valid	Valid	Valid	Not valid	Not valid	Not valid
UCS2- graphic	Not valid	Valid	Not valid	Valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid
Date	Not valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid	Valid	Not valid	Not valid
Time	Not valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid	Not valid	Valid	Not valid
Time- stamp	Not valid	Valid	Valid	Valid	Valid	Not valid	Not valid	Not valid	Not valid	Not valid	Valid

Using concatenation with DBCS fields

When double-byte character set (DBCS) fields are included in a concatenation through the Open Query File (OPNQRYF) command, the resulting data type is the same as the data type of the concatenated field in a logical file, with some slight variations.

The following rules apply when using concatenation with DBCS fields through the OPNQRYF command:

- If the concatenation contains one or more hexadecimal (H) fields, the resulting data type is hexadecimal (H).
- If the concatenation contains one or more UCS2-graphic fields, the resulting data type is UCS2-graphic.
- If all fields in the concatenation are DBCS-only (J), the resulting data type is variable length DBCS-only (J).
- If the concatenation contains one or more DBCS (O, E, J) fields, but no hexadecimal (H) or UCS2-graphic fields, the resulting data type is variable length DBCS open (O).
- If the concatenation contains one or more variable length fields of any data type, the resulting data type is variable length.
- If a DBCS-graphic (G) field is concatenated to another DBCS-graphic (G) field, the resulting data type is DBCS-graphic (G).

   Related concepts

   "DBCS field concatenation" on page 286
   When fields are concatenated, the data types can change (the resulting data type is automatically determined by the system). Double-byte character set (DBCS) field concatenation follows these rules.

Using sort sequence with DBCS fields

When a sort sequence is specified on the Open Query File (OPNQRYF) command, the double-byte character set (DBCS) data is not translated.

Only SBCS data in DBCS-either or DBCS-open fields is translated. UCS2 data is translated.

Related information for Database programming

Product manuals and other information center topic collections contain information that relates to the Database programming topic collection. You can view or print any of the PDF files.

Manuals

- IDDU Use, SC41-5704   (562 KB)

   This manual provides the administrative secretary, business professional, or programmer with information about using the interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system.

- Query for i5/OS, SC41-5210   (1351 KB)

   This manual provides the administrative secretary, business professional, or programmer with information about using IBM Query for i5/OS to get data from any database file. It describes how to sign on to Query and how to define and run queries to create reports that contain the selected data.

Other information

- Application programming interfaces

   This topic collection provides the application programmer with the information needed to develop system-level applications and other i5/OS applications using the application programming interfaces (APIs).

- Backup and recovery

   These topic collections contain information about how to plan a backup and recovery strategy, how to set up disk protection for your data, how to back up your system, and how to control your system shutdown in the event of a failure.

- Commitment control

   This topic collection contains information about using commitment control to ensure that database changes are synchronized.

- Control language

This topic collection provides the application programmer and system programmer with detailed information about control language (CL) and commands.

- Database file management

  This topic collection provides the application programmer with information about using files in application programs. Included are topics on the Copy File (CPYF) command and the override commands.

- DB2 for i5/OS SQL reference

  This topic collection provides the application programmer, programmer, or database administrator with detailed information about Structured Query Language (SQL) statements and their parameters.

- DDS concepts

  This topic collection provides the application programmer with descriptions of the entries and keywords needed to describe database files (both logical and physical) and certain device files (for displays, printers, and the intersystem communications function (ICF) that are external to the user's programs).

- Disk management

  This topic collection helps you manage and protect disk units and disk pools for continuously available information.

- i5/OS globalization

  This topic collection provides the data processing manager, system operator, system manager, application programmer, user, and system engineer with information about the i5/OS national language support function. It prepares the user for planning, installing, configuring, and using the i5/OS globalization and multilingual system. It also explains database management of multilingual data and application considerations for a multilingual system.

- Journal management

  This topic collection provides information about how to set up, manage, and troubleshoot system-managed access-path protection (SMAPP), local journals, and remote journals.

- Performance

  This topic collection provides a description of tuning the system, collecting performance data, working with system values to control or change the overall operation of the system, and gathering data to determine who is using the system and what resources are being used.

- Security reference

  This topic collection provides information about planning, setting up, managing, and auditing security on your system.

- SQL programming

  This topic collection provides the application programmer, programmer, or database administrator with an overview of how to design, write, run, and test SQL statements. It also describes interactive SQL.

- Work management

  This topic collection provides the programmer with information about how to create and change a work management environment.

  Related reference

  "PDF file for Database programming" on page 2
  You can view and print a PDF file of this information.

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER

EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Appendix. Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

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

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

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

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

COPYRIGHT LICENSE:

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

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This Database programming publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

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

AS/400
COBOL/400
DB2
DB2 Universal Database
DRDA
i5/OS
IBM
IBM (logo)
Integrated Language Environment
iSeries
Redbooks
REXX
RPG/400
System i
System/36
System/38
WebSphere

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

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

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

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM ®

Printed in USA