



@server

iSeries

DB2 Multisystem

Version 5





@server

iSeries

DB2 Multisystem

Version 5

Contents

About DB2 Multisystem v

Who should read DB2 Multisystem. v

Code Disclaimer Information v

Chapter 1. Introducing DB2 Multisystem 1

Benefits of using DB2 Multisystem 2

DB2 Multisystem: Basic terms and concepts 2

Chapter 2. Introducing node groups with DB2 Multisystem 5

How node groups work with DB2 Multisystem. 5

Tasks to complete before using the node group commands with DB2 Multisystem 6

Creating node groups using the CRTNODGRP command with DB2 Multisystem 6

Displaying node groups using the DSPNODGRP command with DB2 Multisystem 9

Changing node groups using the CHGNODGRPA command with DB2 Multisystem 10

Deleting node groups using the DLTNODGRP command with DB2 Multisystem 11

Chapter 3. Creating distributed files with DB2 Multisystem 13

Creating a distributed physical file with DB2 Multisystem 13

Restrictions when creating or working with distributed files with DB2 Multisystem 15

Using distributed files with DB2 Multisystem 16

Issuing CL commands against distributed files with DB2 Multisystem. 17

CL Commands: Allowable to run against a distributed file with DB2 Multisystem 17

CL Commands: Affecting only local pieces of a distributed file with DB2 Multisystem 18

CL commands: Affecting all the pieces of a distributed file with DB2 Multisystem 19

Journaling considerations with DB2 Multisystem 20

Using the copy file (CPYF) command with distributed files with DB2 Multisystem 20

Partitioning with DB2 Multisystem 22

Planning for partitioning with DB2 Multisystem 22

Choosing a partitioning key with DB2 Multisystem 23

Customizing the distribution of data with DB2 Multisystem 24

Chapter 4. Scalar functions available with DB2 Multisystem 25

PARTITION with DB2 Multisystem 25

Examples of PARTITION with DB2 Multisystem 25

HASH with DB2 Multisystem 26

Example of HASH with DB2 Multisystem 26

NODENAME with DB2 Multisystem 27

Examples of NODENAME with DB2 Multisystem 27

NODENUMBER with DB2 Multisystem 28

Example of NODENUMBER with DB2

Multisystem 28

Special registers with DB2 Multisystem 28

Relative record numbering (RRN) function with

DB2 Multisystem 29

Chapter 5. Performance and scalability with DB2 Multisystem 31

Why should you use DB2 Multisystem? 31

Performance enhancement tip with DB2

Multisystem 33

How can DB2 Multisystem help you expand your database system? 33

Redistribution issues for adding systems to a network 33

Chapter 6. Query design for performance with DB2® Multisystem . . . 35

Optimization overview with DB2 Multisystem. 36

Implementation and optimization of a single file query with DB2 Multisystem 36

Implementation and optimization of record ordering with DB2 Multisystem. 38

Implementation and optimization of the UNION and DISTINCT clauses with DB2 Multisystem. 39

Processing of the DSTDTA and ALWCOPYDTA parameters with DB2 Multisystem. 39

Implementation and optimization of joins with DB2 Multisystem 39

Co-located join with DB2 Multisystem 40

Directed join with DB2 Multisystem 41

Repartitioned join with DB2 Multisystem 42

Broadcast join with DB2 Multisystem. 43

Join optimization with DB2 Multisystem. 44

Partitioning keys over join fields with DB2

Multisystem 44

Implementation and optimization of grouping with DB2 Multisystem 45

One-step grouping with DB2 Multisystem 45

Two-step grouping with DB2 Multisystem 45

Grouping and joins with DB2 Multisystem 46

Subquery support with DB2 Multisystem 47

Access plans with DB2 Multisystem 47

Reusable open data paths (ODPs) with DB2 Multisystem 47

Temporary result writer with DB2 Multisystem 48

Temp writer job: Advantages with DB2

Multisystem 49

Temp Writer Job: Disadvantages with DB2

Multisystem 50

Control of the temp writer with DB2 Multisystem 50

Optimizer messages with DB2 Multisystem. 50

Changes to the change query attributes (CHGQRYA)
 command with DB2 Multisystem 52
 Asynchronous job usage (ASYNCJ) parameter
 with DB2 Multisystem. 52
 Apply remote (APYRMT) parameter 53
 Summary of performance considerations 54

Bibliography 55
Index 57

About DB2 Multisystem

This book describes the fundamental concepts of DB2 Multisystem, such as distributed relational database files, node groups, and partitioning. This book provides the information necessary to create and to use database files that are partitioned across multiple iSeries servers. Information is provided on how to configure the systems, how to create the files, and how the files can be used in applications.

To learn if you should read DB2 Multisystem, see "Who should read DB2 Multisystem"

Who should read DB2 Multisystem

This book is intended for system administrators or database managers who manage databases containing large amounts of data. Users of this book should have a good understanding of how to create and to use databases and should be familiar with database management and system administration.

To get started with DB2 Multisystem, proceed to Chapter 1, "Introducing DB2 Multisystem" on page 1

Code Disclaimer Information

This document contains programming examples.

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.

All sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All programs contained herein are provided to you "AS IS" without any warranties of any kind. The implied warranties of non-infringement, merchantability and fitness for a particular purpose are expressly disclaimed.

Chapter 1. Introducing DB2 Multisystem

DB2 Multisystem is a parallel processing technique that provides an almost unlimited scalability option for databases. Using DB2 Multisystem, you have the capability to attach multiple iSeries servers (up to 32 servers) together in a "shared nothing" cluster. (Shared nothing means that each system in the coupled network owns and manages its own main memory and disk storage.) Once the systems are connected, database files can be spread across the storage units on each connected system. The database files can have data partitioned (distributed) across a set of systems, and each system has access to all of the data in the file. Yet to users, the file behaves like a local file on their system. From the user's perspective, the database appears as a single database—the user can run queries in parallel across all the systems in the network and have real-time access to the data in the files.

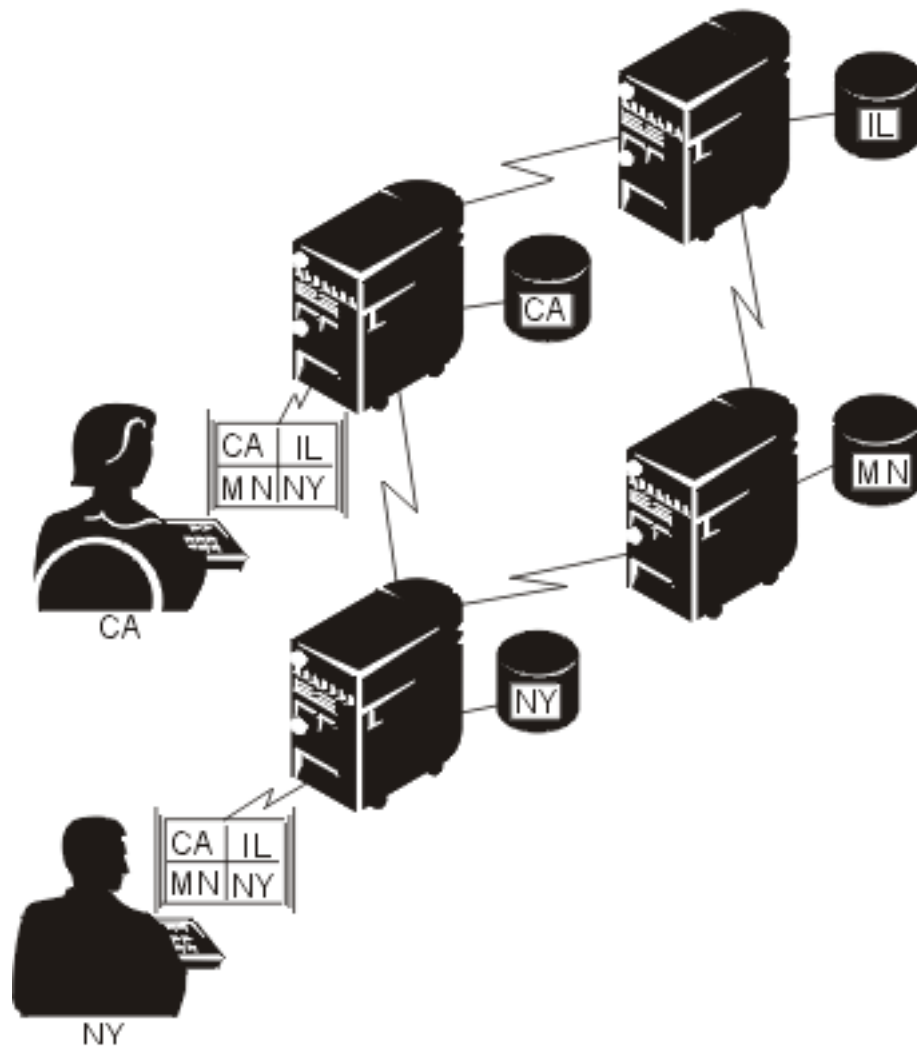


Figure 1. Distribution of Database Files across Systems

This parallel processing technique means that heavy usage on one server does not degrade the performance on the other connected systems in the network. If you

have large volumes of data and the need to run queries, DB2 Multisystem provides you with a method of running those queries in one of the most efficient methods available. In most cases, query performance improves because the queries no longer run against local files, but run in parallel across several servers.

If you have not yet installed DB2 Multisystem, the *Software Installation* book, contains the information that you need in the procedure for installing additional licensed programs. To install DB2 Multisystem, use option 27 in the list of installable options for the OS/400 operating system.

For more introductory information, see the following:

- “Benefits of using DB2 Multisystem”
- “DB2 Multisystem: Basic terms and concepts”

Benefits of using DB2 Multisystem

You can realize the benefits of using DB2 Multisystem in several ways:

- Query performance may improve by running in parallel (pieces of the query are run simultaneously on different servers)
- Need for data replication decreases because all of the servers can access all of the data
- Much larger database files can be accommodated
- Applications are no longer concerned with the location of remote data
- When growth is needed, you can redistribute the file across more systems, and applications can run unchanged on the new systems

With DB2 Multisystem, you can use the same input/output (I/O) methods (GETs, PUTs, and UPDATES) or file access methods that you have used in the past. No additional or different I/O methods or file access methods are required.

Your applications do not have to change-whatever connectivity methods you currently use, unless you are using OptiConnect, will also work for any distributed files you create. With OptiConnect, you must use the OptiConnect controller descriptions. For more information on OptiConnect, see the *OptiConnect for OS/400* book.

DB2 Multisystem: Basic terms and concepts

A **distributed file** is a database file that is spread across multiple iSeries servers. This section describes some of the main concepts that are used in discussing the creation and use of distributed files by DB2 Multisystem. Each of the terms and concepts is discussed in more detail in the following chapters.

Each server that will have a piece of a distributed file is called a **node**. Each server is identified by the name that is defined for it in the relational database directory.

A group of systems that will contain one or more distributed files is called a **node group**. A node group is a system object that contains the list of nodes across which the data will be distributed. A system can be a node in more than one node group.

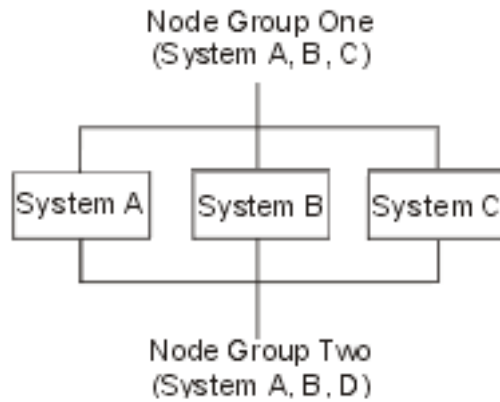


Figure 2. Node Groups

A file is distributed across all the systems in a node group through **partitioning**.

A **partition number** is a number from 0 to 1023. Each partition number is assigned to a node in the node group. Each node can be assigned many partition numbers. The correlation between nodes and partition numbers is stored in a **partition map**. The partition map is also stored as part of the node group object. You can provide the partition map when you create a node group; otherwise, the system will generate a default map.

You define a partition map by using a **partitioning file**. A partitioning file is a physical file that defines a node number for each partition number.

A **partitioning key** consists of one or more fields in the file that is being distributed. The partitioning key is used to determine which node in the node group is to physically contain rows with certain values. This is done by using **hashing**, an operating system function that takes the value of the partitioning key for a record and maps it to a partition number. The node corresponding to that partition number will be used to store the record.

The following example shows what partition number and nodes might look like for a distributed table for two systems. The table has a partitioning key of LASTNAME.

Partition Number	Node
0	SYSA
1	SYSB
2	SYSA
3	SYSB
...	

Figure 3. Partition Map. Partition number 0 contains SYSA, partition number 1 contains node SYSB, partition number 2 contains SYSA, partition number 3 contains SYSB. This pattern repeats.

The hashing of the partitioning key will determine a number that corresponds to a partition number. For example, a record that has a value of 'Andrews' may hash to partition number 1. A record that has a value of 'Anderson' may hash to partition number 2. If you refer to the partition map shown in Figure 3, records for partition number 1 are stored at SYSB, while records for partition number 2 are stored at SYSA.

Chapter 2. Introducing node groups with DB2 Multisystem

To enable database files to be visible across a set of iSeries servers, you must first define the group of systems (node group) that you want the files on.

A node group can have 2 to 32 nodes defined for it. The number of nodes that is defined for the node group determines the number of systems across which the database file is created. The local system must be one of the systems that is specified in the node group. When the system creates the node group, the system assigns a number, starting with number 1, to each node.

To work with node groups, see the following topics:

- “How node groups work with DB2 Multisystem”
- “Tasks to complete before using the node group commands with DB2 Multisystem” on page 6
- “Creating node groups using the CRTNODGRP command with DB2 Multisystem” on page 6
- “Displaying node groups using the DSPNODGRP command with DB2 Multisystem” on page 9
- “Changing node groups using the CHGNODGRPA command with DB2 Multisystem” on page 10
- “Deleting node groups using the DLTNODGRP command with DB2 Multisystem” on page 11

How node groups work with DB2 Multisystem

A node group is a system object (*NODGRP), which is stored on the system on which it was created. It is not a distributed object. The *NODGRP system object contains all the information about the systems in the group as well as information about how the data in the data files should be partitioned (distributed). The default partitioning is for each system (node) to receive an equal share of the data.

The partitioning is handled using a hash algorithm. When a node group is created, partition numbers ranging from 0 to 1023 are associated with the node group. With the default partitioning, an equal number of partitions are assigned to each of the nodes in the node group. When data is added to the file, the data in the partitioning key is hashed, which results in a partition number. The entire record of data is not hashed—only the data in the partitioning key is hashed through the hash algorithm. The node that is associated with the resulting partition number is where the record of data will physically reside. Therefore, with the default partitioning, each node stores an equal share of the data, provided that there are enough records of data and a wide range of values.


If you do not want each node to receive an equal share of the data or if you want to control which systems have specific pieces of data, you can change how the data is partitioned, either by specifying a custom partitioning scheme on the Create Node Group (CRTNODGRP) command using the partition file (PTNFILE) parameter, or by changing the partitioning scheme later using the Change Node Group Attributes (CHGNODGRPA) command. Using the PTNFILE parameter, you can set the node number for each of the partitions within the node group; in other words, the PTNFILE parameter allows you to tailor how you want data to be

partitioned on the systems in the node group. (The PTNFILE parameter is used in an example in “Creating node groups using the CRTNODGRP command with DB2 Multisystem”.) For more information on partitioning, see “Partitioning with DB2 Multisystem” on page 22.

Because a node group is a system object, it can be saved and restored using the Save Object (SAVOBJ) command and the Restore Object (RSTOBJ) command. You can restore a node group object either to the system on which it was created or to any of the systems in the node group. If the node group object is restored to a system that is not in the node group, the object will be unusable.

Tasks to complete before using the node group commands with DB2 Multisystem

Before using the Create Node Group (CRTNODGRP) command or any of the node group commands, you must ensure that the distributed relational database network you are using has been properly set up. If this is a new distributed

relational database network, see Distributed Database Programming  for information on establishing the network.

You need to ensure that one system in the network is defined as the local (*LOCAL) system. Use the Work with RDB (Relational Database) Directory Entries (WRKRDBDIRE) command to display the details about the entries. If a local system is not defined, you can do so by specifying *LOCAL for the remote location name (RMTLOCNAME) parameter of the Add RDB Directory Entries (ADDRDBDIRE) command, for example:

```
ADDRDBDIRE RDB(MP000) RMTLOCNAME(*LOCAL) TEXT ('New York')
```

The iSeries server in New York, named MP000, is defined as the local system in the relational database directory. You can have only one local relational database defined on an iSeries server as the system name or local location name specified for this system in your network configuration. This can help you identify a database name and correlate it to a particular system in your distributed relational database network, especially if your network is complex.

For DB2 Multisystem to properly distribute files to the iSeries servers within the node groups that you define, you must have the remote database (RDB) names consistent across all the nodes (systems) in the node group.

For example, if you plan to have three systems in your node group, each system must have at least three entries in the RDB directory. On each system, the three names must all be the same. On each of the three systems, an entry exists for the local system, which is identified as *LOCAL. The other two entries contain the appropriate remote location information.

Creating node groups using the CRTNODGRP command with DB2 Multisystem

This section uses two CL command examples to show how to create a node group by using the Create Node Group (CRTNODGRP) command.

In the following example, a node group with default partitioning (equal partitioning across the systems) is created:

```
CRTNODGRP NODGRP(LIB1/GROUP1) RDB(SYSTEMA SYSTEMB SYSTEMC SYSTEMD)
          TEXT('Node group for test files')
```

In this example, the command creates a node group that contains four nodes. Note that each of the nodes must be defined RDB entries (previously added to the relational database directory using the ADDRDBDIRE command) and that one node must be defined as local (*LOCAL).

The partitioning attributes default to assigning one-fourth of the partitions to each node number. This node group can be used on the NODGRP parameter of the Create Physical File (CRTPF) command to create a distributed file. For more information on distributed files, see Chapter 3, “Creating distributed files with DB2 Multisystem”.

Note: One-fourth of the data is not necessarily partitioned on each system because of the variation within the data. For example, you decide to partition your data based on telephone area code by selecting the area code field as the partitioning key. As it happens, 60% of your business comes from the 507 area code. The system that has the partition for the 507 area code will receive 60%, not 25%, of the data.

In the following example, a node group with specified partitioning is created by using the partitioning file (PTNFILE) parameter:

```
CRTNODGRP NODGRP(LIB1/GROUP2) RDB(SYSTEMA SYSTEMB SYSTEMC)
          PTNFILE(LIB1/PTN1)
          TEXT('Partition most of the data to SYSTEMA')
```

In this example, the command creates a node group that contains three nodes (SYSTEMA, SYSTEMB, and SYSTEMC). The partitioning attributes are taken from the file called PTN1. This file can be set up to force a higher percentage of the records to be located on a particular system.

The file PTN1 in this example is a partitioning file. This file is not a distributed file, but a regular local physical file that can be used to set up a custom partitioning scheme. The partitioning file must have one 2-byte binary field. The partitioning file must contain 1024 records in which each record contains a valid node number.

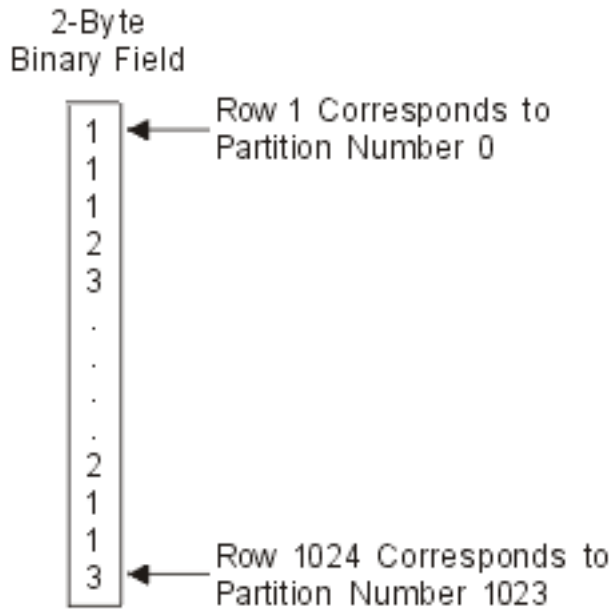


Figure 4. Example of the Contents of Partitioning File PTNFILE

If the node group contains three nodes, all of the records in the partitioning file must have numbers 1, 2, or 3. The node numbers are assigned in the order that the RDB names were specified on the Create Node Group (CRTNODGRP) command. A higher percentage of data can be forced to a particular node by having more records containing that node number in the partitioning file. This is a method for customizing the partitioning with respect to the amount of data that will physically reside on each system. To customize the partitioning with respect to specific values residing on specific nodes, use the Change Node Group Attributes (CHGNODGRPA) command. See “Changing node groups using the CHGNODGRPA command with DB2 Multisystem” on page 10 for more information.

You should note that, because the node group information is stored in the distributed file, the file is not immediately sensitive to changes in the node group or to changes in the RDB directory entries that are included in the node group. You can make modifications to node groups and RDB directory entries, but until you use the CHGPF command and specify the changed node group, your files will not change their behavior.

Another concept is that of a **visibility node**. A visibility node within a node group contains the file object (part of the mechanism that allows the file to be distributed across several nodes), but no data. A visibility node retains a current level of the file object at all times; the visibility node simply has no data stored on it. In contrast, a node (sometimes called a **data node**) contains data. As an example of how you could use a visibility node in your node group, let’s have the iSeries server that your sales executives use be part of your node group. These executives probably do not want to run queries on a regular basis, but on occasion they may want to run a specific query. From their server, they can run their queries, access real-time data, and receive the results of their query. So even though none of the data is stored on their server, because their system is a visibility node, the executives can run the query whenever necessary.

To specify a node as being a visibility node, you must use the PTNFILE parameter on the Create Node Group (CRTNODGRP) command. If the partitioning file contains no records for a particular node number, that node is a visibility node.

Displaying node groups using the DSPNODGRP command with DB2 Multisystem

The Display Node Group (DSPNODGRP) command displays the nodes (systems) in a node group. It also displays the partitioning scheme for the node group (partitioning is discussed later in “Partitioning with DB2 Multisystem” on page 22).

The following example shows how to display a node group named GROUP1 as well as the partitioning scheme that is associated with the node group. This information is displayed to you at your workstation. For complete details on the

DSPNODGRP command, see the Control Language  topic in the Information Center.

```
DSPNODGRP NODGRP(LIB1/GROUP1)
```

When you issue the DSPNODGRP command with a node group name specified, the Display Node Group display is shown. This display shows you the names of systems (listed in the relational database column) and the node number that is assigned to the system. This is a direct method for determining what system has what node number.

Display Node Group

Node Group: GROUP1 Library: LIB1

Relational Database	Node Number
SYSTEMA	1
SYSTEMB	2
SYSTEMC	3

BottomF3=Exit F11=Partitioning Data F12=Cancel F17=Top F18=Bottom

Figure 5. Display Node Group: Database to Node Number Correlation in the Node Group display

To see the node number that is assigned to each partition number, use F11 (Partitioning Data) on the Display Node Group display. The next display shows you the node number that is assigned to each partition number. Mapping between the system and the node number (or node number to system) can be easily done by using the DSPNODGRP command.

Display Node Group	
Node Group:	GROUP1 Library: LIB1
Partition Number	Node Number
0	1
1	2
2	3
3	1
4	2
5	3
6	1
7	2
8	3
9	1
10	2
11	3
12	1
13	2
14	3
More...	
F3=Exit	F11=Node Data F12=Cancel F17=Top F18=Bottom

Figure 6. Display Node Group: Partition Number to Node Number Correlation

The following example prints a list of the systems in the node group named GROUP2 as well as the associated partitioning scheme:

```
DSPNODGRP NODGRP(LIB1/GROUP2) OUTPUT(*PRINT)
```

Changing node groups using the CHGNODGRPA command with DB2 Multisystem

The Change Node Group Attributes (CHGNODGRPA) command changes the data partitioning attributes for a node group. The node group contains a table with 1024 partitions; each partition contains a node number. Node numbers were assigned when the node group was created and correspond to the relational databases specified on the RDB parameter of the Create Node Group (CRTNODGRP) command. Use the Display Node Group (DSPNODGRP) command to see the valid node number values and the correlation between node numbers and relational database names.

The CHGNODGRPA command does not affect any existing distributed files that were created using the specified node group. For the changed node group to be used, the changed node group must be specified either when creating a new file or on the Change Physical File (CHGPF) command. For complete details on the

CHGNODGRPA command, see the Control Language  topic in the Information Center.

This first example shows how to change the partitioning attributes of the node group named GROUP1 in library LIB1:

```
CHGNODGRPA NODGRP(LIB1/GROUP1) PTNNBR(1019)
           NODNBR(2)
```

In this example, the partition number 1019 is specified, and any records that hash to 1019 are written to node number 2. This provides a method for directing specific partition numbers to specific nodes within a node group.

The second example changes the partitioning attributes of the node group named GROUP2. (GROUP2 is found by using the library search list, *LIBL.) The value

specified on the comparison data value (CMPDTA) parameter is hashed, and the resulting partition number is changed from its existing node number to node number 3. (Hashing and partitioning are discussed in “Partitioning with DB2 Multisystem” on page 22.)

```
CHGNODGRPA NODGRP(GROUP2) CMPDTA('CHICAGO')  
NODNBR(3)
```

Any files that are created using this node group and that have a partitioning key consisting of a character field will store records that contain 'CHICAGO' in the partitioning key on node number 3. To allow for files with multiple fields in the partitioning key, you can specify up to 300 values on the compare data (CMPDTA) parameter.


When you enter values on the CMPDTA parameter, you should be aware that the character data is case-sensitive. This means that 'Chicago' and 'CHICAGO' do not result in the same partition number. Numeric data should be entered simply as numeric digits; do not use a decimal point, leading zeros, or following zeros.

All values are hashed to obtain a partition number, which is then associated with the node number that is specified on the node number (NODNBR) parameter. The text of the completion message, CPC3207, shows the partition number that was changed. Be aware that by issuing the CHGNODGRPA command many times and for many different values that you increase the chance of changing the same partition number twice. If this occurs, the node number that is specified on the most recent change is in effect for the node group.

For additional information on customizing the distribution of your data, see “Customizing the distribution of data with DB2 Multisystem” on page 24.

Deleting node groups using the DLTNODGRP command with DB2 Multisystem

The Delete Node Group (DLTNODGRP) command deletes a previously created node group. This command does not affect any files that were created using the node group. For complete details on the DLTNODGRP command, see the Control

Language  topic in the Information Center.

This following example shows how to delete the node group named GROUP1. Any files that are created with this node group are not affected:

```
DLTNODGRP NODGRP(LIB1/GROUP1)
```

Even though the deletion of the node group does not affect files that were created using that node group, it is not recommended that you delete node groups after using them. Once the node group is deleted, you can no longer use the DSPNODGRP command to display the nodes and the partitioning scheme. However, you can use the Display File Description (DSPFD) command with TYPE(*NODGRP) specified to see the node group associated with a particular file.

Chapter 3. Creating distributed files with DB2 Multisystem

A distributed file is a database file that is spread across multiple iSeries servers. Distributed files can be updated, and they can be accessed through such methods as SQL, query tools, and the Display Physical File Member (DSPPFM) command. For database operations, distributed files are treated just like local files, for the most part. For information on CL command changes for distributed files, see “Issuing CL commands against distributed files with DB2 Multisystem” on page 17.

To create a database file as a distributed file, you can use either the Create Physical File (CRTPF) command or the SQL CREATE TABLE statement. Both methods are discussed in more detail in this chapter. The CRTPF command has two parameters, node group (NODGRP) and partitioning key (PTNKEY), that create the file as a distributed file. You will see the distributed database file as an object (*FILE with the PF attribute) that has the name and library that you specified when you ran the CRTPF command.

If you want to change an existing nondistributed database physical file into a distributed file, you can do this by using the Change Physical File (CHGPF) command. On the CHGPF command, the NODGRP and PTNKEY parameters allow you to make the change to a distributed file. Through the CHGPF command, you can also change the data partitioning attributes for an existing distributed database file by specifying values for the NODGRP and PTNKEY parameters. Specifying values for these parameters causes the data to be redistributed according to the partitioning table in the node group.

Note: All logical files that are based over the physical file that is being changed to a distributed file also become distributed files. For large files or large networks, redistributing data can take some time and should not be done frequently.

To work with distributed files, see the following:

- “Creating a distributed physical file with DB2 Multisystem”
- “Using distributed files with DB2 Multisystem” on page 16
- “Partitioning with DB2 Multisystem” on page 22
- “Customizing the distribution of data with DB2 Multisystem” on page 24

Creating a distributed physical file with DB2 Multisystem

This section discusses how to create a distributed physical file using the CRTPF command or the SQL CREATE TABLE statement.

When you want to create a partitioned file, you need to specify the following parameters on the Create Physical File (CRTPF) command:

- A node group name for the NODGRP parameter
- The field or fields that are to be used as the partitioning key (use the PTNKEY parameter)


The partitioning key determines where (on what node) each record of data will physically reside. You specify the partitioning key when the Create Physical File (CRTPF) command is run or when the SQL CREATE TABLE statement is run. The

values of the fields that make up the partitioning key for each record are processed by the HASH algorithm to determine where the record will be located.

If you have a single-field partitioning key, all records with the same value in that field will reside on the same system.

If you want to create a distributed physical file, your user profile must exist on every node within the node group, and your user profile must have the authority needed to create a distributed file on every node. If you need to create a distributed file in a specific library, that library must exist on every node in the node group, and your user profile must have the necessary authority to create files in those libraries. If any of these factors are not true, the file will not be created.

The way that the systems are configured can influence the user profile that is used on the remote system. To ensure that your user profile is used on the remote system, that system should be configured as a secure location. To determine if a system is configured as a secure location, use the Work with Configuration Lists (WRKCFGL) command. For more information, see Distributed Database

Programming  topic in the Information Center.

The following example shows how to create a physical file named PAYROLL that is partitioned (specified by using the NODGRP parameter) and has a single partitioning key on the employee number (EMPNUM) field:

```
CRTPF FILE(PRODLIB/PAYROLL) SCRFILE(PRODLIB/DDS) SRCMBR(PAYROLL)
      NODGRP(PRODLIB/PRODGROUP) PTNKEY(EMPNUM)
```

When the CRTPF command is run, the system creates a distributed physical file to hold the local data associated with the distributed file. The CRTPF command also creates physical files on all of the remote systems specified in the node group.

The ownership of the physical file and the public authority on all the systems is consistent. This consistency also includes any authority specified on the AUT parameter of the CRTPF command.

The SQL CREATE TABLE statement also can be used to specify the node group and the partitioning key. In the following example, an SQL table called PAYROLL is created. The example uses the IN *nodgroup-name* clause and the PARTITIONING KEY clause.

```
CREATE TABLE PRODLIB/PAYROLL
  (EMPNUM INT, EMPLNAME CHAR(12), EMPFNAME CHAR (12))
  IN PRODLIB/PRODGROUP
  PARTITIONING KEY (EMPNUM)
```

When the **PARTITIONING KEY** clause is not specified, the first column of the primary key, if one is defined, is used as the first partitioning key. If no primary key is defined, the first column defined for the table that does not have a data type of date, time, timestamp, or floating-point numeric is used as the partitioning key.

To see if a file is partitioned, use the Display File Description (DSPFD) command. If the file is partitioned, the DSPFD command shows the name of the node group, shows the details of the node group stored in the file object (including the entire partition map), and lists the fields of the partitioning key.

For a list of restrictions you need to know when using distributed files with DB2 Multisystem, see “Restrictions when creating or working with distributed files with DB2 Multisystem”.

Restrictions when creating or working with distributed files with DB2 Multisystem

You need to be aware of the following restrictions when creating or working with distributed files:

- First-change first-out (FCFO) access paths cannot be used because the access paths are partitioned across multiple nodes.
- A distributed file can have a maximum of one member.
- A distributed file is not allowed in a temporary library (QTEMP).
- Data in the partitioning key has a limited update capability. Generally, when choosing a partitioning key, you should choose fields whose values do not get updated. Updates to the partitioning key are allowed as long as the update does not cause the record to be partitioned to a different node.
- Date, time, timestamp, or floating-point numeric fields cannot be used in the partitioning key.
- Source physical files are not supported.
- Externally described files are supported for distributed files; program-described files are not supported.
- If the access path is unique, the partitioning key must be a subset of the unique key access path.
- Constraints are supported, and referential constraints are supported only if the node group of both the parent and foreign key files are identical and **all** of the fields of the partitioning key are included in the constraint. The partitioning key must be a subset of the constraint fields. Also, for unique and primary constraints, if the access path is unique, the partitioning key must be a subset of the unique key access path.
- On the CRTPF command, the system parameter must have the value *LCL specified (CRTPF SYSTEM(*LCL)). SYSTEM(*RMT) is not allowed.
- Any time a logical file is created over a distributed file, the logical file also becomes distributed, which means that you cannot build a local logical file over just one piece of the physical file on a specific node. SQL views are the exception to this, if the view is a join and if all of the underlying physical files do not have the same node group. In this case, the view is only created on the local system. Even though this view is not distributed, if you query the view, data is retrieved from all of the nodes, not just from the node where the view was created.

Join files can only be created using SQL.

For DDS-created logical files, only one based-on file is allowed.

- Coded character set identifiers (CCSIDs) and sort sequence (SRTSEQ) tables are resolved from the originating system.
- Variable-length record (VLR) processing is not supported. This does not mean that variable-length fields are not supported for distributed files. This restriction only refers to languages and applications that request VLR processing when a file is opened.
- End-of-file delay (EOFDLY) processing is not supported.
- Data File Utility (DFU) does not work against distributed files, because DFU uses relative record number processing to access records.
- A distributed file cannot be created into a library located on an independent auxiliary storage pool (IASP).

Using distributed files with DB2 Multisystem

Once the file is created, the system ensures that the data is partitioned and that the files remain at concurrent levels. Activity that occurs automatically once the file is created includes:

- All indexes created for the file are created on all the nodes.
- Authority changes are sent to all nodes.
- The system prevents the file from being moved and prevents its library from being renamed.
- If the file itself is renamed, its new name is reflected on all nodes.
- Several commands, such as Allocate Object (ALCOBJ), Reorganize Physical File Member (RGZPFM), and Start Journal Physical File (STRJRNPf), now affect all of the pieces of the file. This allows you to maintain the concept of a local file when working with partitioned files. See “CL commands: Affecting all the pieces of a distributed file with DB2 Multisystem” on page 19 for a complete list of these CL commands.

You can issue the Allocate Object (ALCOBJ) command from any of the nodes in the node group. This locks all the pieces and ensures the same integrity that would be granted when allocating a local file. All of these actions are handled by the system, which keeps you from having to enter the commands on each node.

In the case of the Start Journal Physical File (STRJRNPf) command, journaling is started on each system. Therefore, each system must have its own journal and journal receiver. Each system has its own journal entries; recovery using the journal entries must be done on each system individually. The commands to start and end journaling affect all of the systems in the node group simultaneously. See “Journaling considerations with DB2 Multisystem” on page 20 for additional information.

- Several commands, such as Dump Object (DMPOBJ), Save Object (SAVOBJ), and Work with Object Locks (WRKOBJLCK), only affect the piece of the file on the system where the command was issued. See “CL Commands: Affecting only local pieces of a distributed file with DB2 Multisystem” on page 18 for a complete list of these CL commands.

Once a file has been created as a distributed file, the opening of the file actually results in an open of the local piece of the file as well as connections being made to all of the remote systems. Once the file is created, it can be accessed from any of the systems in the node group. The system also determines which nodes and records it needs to use to complete the file I/O task (GETS, PUTs, and UPDATES, for example). You do not have to physically influence or specify any of this activity.

Note that Distributed Relational Database Architecture (DRDA[®]) and distributed data management (DDM) requests can target distributed files. Previously distributed applications that use DRDA or DDM to access a database file on a remote system can continue to work even if that database file was changed to be a distributed file.

You should be aware that the arrival sequence of the records is different for distributed database files than that of a local database file.

Because distributed files are physically distributed across systems, you cannot rely on the arrival sequence or relative record number of the records. With a local database file, the records are dealt with in order. If, for example, you are inserting

data into a local database file, the data is inserted starting with the first record and continuing through the last record. All records are inserted in sequential order. Records on an individual node are inserted the same way that records are inserted for a local file.

When data is read from a local database file, the data is read from the first record on through the last record. This is not true for a distributed database file. With a distributed database file, the data is read from the records (first to last record) in the first node, then the second node, and so on. For example, reading to record 27 no longer means that you read to a single record. With a distributed file, each node in the node group may contain its own record 27—none of which would be the same.

For more information on issuing CL commands, see “Issuing CL commands against distributed files with DB2 Multisystem”.

Issuing CL commands against distributed files with DB2 Multisystem

Because a distributed file has a system object type of *FILE, many of the CL commands that access physical files can be run against distributed files. However, the behavior of some CL commands changes when issued against a distributed file versus a nondistributed file. The following sections address how CL commands work with distributed files. For complete information on CL commands, see the

Control Language  topic in the Information Center.

To learn about specific types of CL commands you can use with DB2 Multisystem, see the following:

- “CL Commands: Allowable to run against a distributed file with DB2 Multisystem”
- “CL Commands: Affecting only local pieces of a distributed file with DB2 Multisystem” on page 18
- “CL commands: Affecting all the pieces of a distributed file with DB2 Multisystem” on page 19

CL Commands: Allowable to run against a distributed file with DB2 Multisystem

The following CL commands or specific parameters cannot be run against distributed files:

- SHARE parameter of the Change Logical File Member (CHGLFM)
- SHARE parameter of the Change Physical File Member (CHGPFM)
- Create Duplicate Object (CRTDUPOBJ)
- Initialize Physical File Member (INZPFM)
- Move Object (MOVOBJ)
- Position Database File (POSDBF)
- Remove Member (RMVM)
- Rename Library (RNMLIB) for libraries that contain distributed files
- Rename Member (RNMM)
- Integrated File System (IFS) command, COPY

CL Commands: Affecting only local pieces of a distributed file with DB2 Multisystem

The following CL commands, when run, affect only the piece of the distributed file that is located on the local system (the system from which the command was run):

- Apply Journalized Changes (APYJRNCHG). See “Journaling considerations with DB2 Multisystem” on page 20 for additional information on this command.
- Display Object Description (DSPOBJD)
- Dump Object (DMPOBJ)
- End Journal Access Path (ENDJRNAP)
- Remove Journalized Changes (RMVJRNCHG). See “Journaling considerations with DB2 Multisystem” on page 20 for additional information on this command.
- Restore Object (RSTOBJ)
- Save Object (SAVOBJ)
- Start Journal Access Path (STRJRNAP)

You can use the Submit Remote Command (SBMRMTCMD) command to issue any CL command to all of the remote systems associated with a distributed file. By issuing a CL command on the local system and then issuing the same command through the SBMRMTCMD command for a distributed file, you can run a CL command on all the systems of a distributed file. You do not need to do this for CL commands that automatically run on all of the pieces of a distributed file. See “CL commands: Affecting all the pieces of a distributed file with DB2 Multisystem” on page 19 for more information.

The Display File Description (DSPFD) command can be used to display node group information for distributed files. The DSPFD command shows you the name of the node group, the fields in the partitioning key, and a full description of the node group. To display this information, you must specify *ALL or *NODGRP for the TYPE parameter on the DSPFD command.

The Display Physical File Member (DSPPFM) command can be used to display the local records of a distributed file; however, if you want to display remote data as well as local data, you should specify *ALLDATA on the from record (FROMRCD) parameter on the command.

When using the Save Object (SAVOBJ) command or the Restore Object (RSTOBJ) command for distributed files, each piece of the distributed file must be saved and restored individually. A piece of the file can only be restored back to the system from which it was saved if it is to be maintained as part of the distributed file. If necessary, the Allocate Object (ALLOBJ) command can be used to obtain a lock on all of the pieces of the file to prevent any updates from being made to the file during the save process.

The system automatically distributes any logical file when the file is restored if the following are true:

- The logical file was saved as a nondistributed file.
- The logical file is restored to the system when its based-on files are distributed.

The saved pieces of the file also can be used to create a local file. To do this, you must restore the piece of the file either to a different library or to a system that was not in the node group used when the distributed file was created. To get all the records in the distributed file into a local file, you must restore each piece of

the file to the same system and then copy the records into one aggregate file. Use the Copy File (CPYF) command to copy the records to the aggregate file.

CL commands: Affecting all the pieces of a distributed file with DB2 Multisystem

The following CL commands, when run, affect all the pieces of the distributed file. When you run these commands on your system, the command is automatically run on all the nodes within the node group.

This convention allows you to maintain consistency across the node group without having to enter the same command on each system. With authority changes, some inconsistency across the node group may occur. For example, if a user ID is deleted from one system in the node group, the ability to maintain consistency across the node group is lost.

Authority errors are handled individually.

The following commands affect all pieces of the distributed file:

- Add Logical File Member (ADDLFM)
- Add Physical File Constraint (ADDPFCST)
- Add Physical File Member (ADDPFM)
- Add Physical File Trigger (ADDPFTRG)
- Allocate Object (ALCOBJ)
- Change Logical File (CHGLF)
- Change Object Owner (CHGOBJOWN)
- Change Physical File (CHGPF)
- Change Physical File Constraint (CHGPFCST)
- Clear Physical File Member (CLRPFM)
- Copy File (CPYF). See “Using the copy file (CPYF) command with distributed files with DB2 Multisystem” on page 20 for additional information on this command.
- Create Logical File (CRTLF)
- Deallocate Object (DLCOBJ)
- Delete File (DLTF)
- End Journal Physical File (ENDJRNPf). See “Journaling considerations with DB2 Multisystem” on page 20 for additional information on this command.
- Grant Object Authority (GRTOBJAUT)
- Remove Physical File Constraint (RMVPFCST)
- Remove Physical File Trigger (RMVPFTRG)
- Rename Object (RNMOBJ)
- Reorganize Physical File Member (RGZPFM)
- Revoke Object Authority (RVKOBJAUT)
- Start Journal Physical File (STRJRNPf). See “Journaling considerations with DB2 Multisystem” on page 20 for additional information on this command.

For these commands, if any objects other than the distributed file are referred to, it is your responsibility to create those objects on each system. For example, when using the Add Physical File Trigger (ADDPFTRG) command, you must ensure that the trigger program exists on all of the systems. Otherwise, an error occurs. This same concept applies to the Start Journal Physical File (STRJRNPf) command, where the journal must exist on all of the systems.

If the user profile does not exist on the remote node and you issue the GRTOBJAUT command or the RVKOBJAUT command, the authority is granted or revoked on all the nodes where the profile exists and is ignored on any nodes where the profile does not exist.

Journaling considerations with DB2 Multisystem: Although the Start Journal Physical File (STRJRNPf) and End Journal Physical File (ENDJRNPf) commands are distributed to other systems, the actual journaling takes place on each system independently and to each system's own journal receiver.

As an example, you have two systems (A and B) on which you have a distributed file. You need to create a journal and a receiver on both system A and system B, and the journal name and library must be the same on both systems. When you issue the STRJRNPf command, the command is distributed to both systems and journaling starts on both systems. However, the journal receiver on system A will contain only the data for changes that occur to the piece of the file that resides on system A. The journal receiver on system B will contain only the data for changes that occur to the piece of the file that resides on system B.

This affects your save and restore strategy as well as your backup strategy; for example:

- After you issue the STRJRNPf command, you should save the database file from each of the systems in the file's node group.
- You need to practice standard journal management on each of the systems. You need to change and to save the journal receivers appropriately, so that you can manage the disk space usage for each system. Or, you can take advantage of the system change-journal management support.

Note: Just the names of the journal must be the same on each of the systems; the attributes do not. Therefore, for example, you could specify a different journal receiver threshold value on the different systems, reflecting the available disk space on each of those systems.

- If you do need to recover a piece of a distributed database file, you only need to use the journal receiver from the system where the piece of the distributed file resided. From that journal receiver, you apply the journaled changes using the Apply Journaled Changes (APYJRNCHG) command or remove the journaled changes using the Remove Journaled Changes (RMVJRNCHG) command.
- You will **not** be able to use the journal receiver from one system to apply or remove the journaled changes to a piece of the file on another system. This is because each piece of the file on each system has its own unique journal identifier (JID).

See the *Backup and Recovery* book for more information.

Using the copy file (CPYF) command with distributed files with DB2

Multisystem: When the Copy File (CPYF) command is issued, the system tries to run the CPYF command as quickly as possible. The command parameters specified, the file attributes involved in the copy, and the size and number of records to be copied all affect how fast the command is run.

When copying data to distributed files, the performance of the copy command may be improved by using only the following parameters on the CPYF command: FROMFILE, TOFILE, FROMMBR, TOMBR, MBROPT, and FMTOPT(*NONE) or FMTOPT(*NOCHK). Also, the from-file (FROMFILE) and the to-file (TOFILE) parameters should **not** specify files that contain any null-capable fields. Generally,

the simpler the syntax of the copy command is, the greater the chance that the fastest copy operation will be obtained. When the fastest copy method is being used while copying to a distributed file, message CPC9203 is issued, stating the number of records copied to each node. Normally, if this message was not issued, the fastest copy was not performed.

When copying to a distributed file, you should consider the following differences between when the fastest copy is and is not used:

1. For the fastest copy, records are buffered for each node. As the buffers become full, they are sent to a particular node. If an error occurs after any records are placed in any of the node buffers, the system tries to send all of the records currently in the node buffers to their correct node. If an error occurs while the system is sending records to a particular node, processing continues to the next node until the system has tried to send all the node buffers.

In other words, records that follow a particular record that is in error may be written to the distributed file. This action occurs because of the simultaneous blocking done at the multiple nodes. If you do not want the records that follow a record that is in error to be written to the distributed file, you can force the fastest copy not to be used by specifying on the CPYF command either `ERRLVL(*NOMAX)` or `ERRLVL` with a value greater than or equal to 1.

When the fastest copy is not used, record blocking is attempted unless the to-file open is or is forced to become a `SEQONLY(*NO)` open.

2. When the fastest copy is used, a message is issued stating that the opening of the member was changed to `SEQONLY(*NO)`; however, the distributed to-file is opened a second time to allow for the blocking of records. You should ignore the message about the change to `SEQONLY(*NO)`.
3. When the fastest copy is used, multiple messages may be issued stating the number of records copied to each node. A message is then sent stating the total number of records copied.

When the fastest copy is not used, only the total number of records copied message is sent. No messages are sent listing the number of records copied to each node.

The following are restrictions when copying to or from distributed files:

- The `FROMRCD` parameter can be specified only with a value of `*START` or 1 when copying from a distributed file. The `TORCD` parameter cannot be specified with a value other than the default value `*END` when copying from a distributed file.
- The `MBROPT(*UPDADD)` parameter is not allowed when copying to a distributed file.
- The `COMPRESS(*NO)` parameter is not allowed when the to-file is a distributed file and the from-file is a database delete-capable file.
- For copy print listings, the `RCDNBR` value given is the position of the record in the file on a particular node when the record is a distributed file record. The same record number may appear multiple times on the listing, with each one being a record from a different node.

Partitioning with DB2 Multisystem

Partitioning is the process of distributing a file across the nodes in a node group. Partitioning is done using the hash algorithm. When a new record is added, the hash algorithm is applied to the data in the partitioning key. The result of the hash algorithm, a number between 0 and 1023, is then applied to the partitioning map to determine the node on which the record will reside.

The partition map is also used for query optimization, updates, deletes, and joins. The partition map can be customized to force certain key values to certain nodes.

For example, during I/O, the system applies the hash algorithm to the values in the partitioning key fields. The result is applied to the partition map stored in the file to determine which node stores the record.

The following example shows how these concepts relate to each other:

Employee number is the partitioning key and a record is entered into the database for an employee number of 56000. The value of 56000 is processed by the hash algorithm and the result is a partition number of 733. The partition map, which is part of the node group object and is stored in the distributed file when it is created, contains node number 1 for partition number 733. Therefore, this record will physically be stored on the system in the node group that is assigned node number 1. The partitioning key (the PTNKEY parameter) was specified by you when you created the partitioned (distributed) file.

Fields in the partitioning key can be null-capable. However, records that contain a null value within the partitioning key always hash to partition number 0. Files with a significant number of null values within the partitioning key can result in data skew on the partition number 0, because all of the records with null values hash to partition number 0.

After you have created your node group object and a partitioned distributed relational database file, you can use the DSPNODGRP command to view the relationship between partition numbers and node names. See “Displaying node groups using the DSPNODGRP command with DB2 Multisystem” on page 9 for more information on displaying partition numbers, node groups, and system names.

When creating a distributed file, the partitioning key fields are specified either on the PTNKEY parameter of the Create Physical File (CRTPF) command or in the PARTITIONING KEY clause of the SQL CREATE TABLE statement. Fields with the data types DATE, TIME, TIMESTAMP, and FLOAT are not allowed in a partitioning key.

To implement partitioning, see the following:

- “Planning for partitioning with DB2 Multisystem”
- “Choosing a partitioning key with DB2 Multisystem” on page 23

Planning for partitioning with DB2 Multisystem

In most cases, you should plan ahead about how it is that you want to use partitioning and partitioning keys. How should you systematically divide the data for placement on other systems? What data do you frequently want to join in a query? What is a meaningful choice when doing selections? What is the most efficient way to set up the partitioning key to get the data you need?

When planning the partitioning, you should set it up so that the fastest systems receive the most data. You need to consider which systems take advantage of Symmetric Multiprocessing (SMP) parallelism to improve database performance. Note that when the query optimizer builds its distributed access plan, the optimizer counts the number of records on the requesting node and multiplies that number by the total number of nodes. Although putting most of the records on the SMP systems has advantages, the optimizer can offset some of those advantages because it uses an equal number of records on each node for its calculations. For

information on SMP, see SQL Programming Concepts  and Database Programming  .

If you want to influence the partitioning, you can do so. For example, in your business, you have regional sales departments that use certain systems to complete their work. Using partitioning, you can force local data from each region to be stored on the appropriate system for that region. Therefore, the system that your employees in the Northwest United States region use will contain the data for the Northwest Region.

To set the partitioning, you can use the PTNFILE and PTNMBR parameters of the CRTPF command. Use the Change Node Group Attributes (CHGNODGRPA) command to redistribute an already partitioned file. See “Customizing the distribution of data with DB2 Multisystem” on page 24 for more information.

Performance improvements are best for queries that are made across large files. Files that are in high use for transaction processing but seldom used for queries may not be the best candidates for partitioning and should be left as local files.

For join processing, if you often join two files on a specific field, you should make that field the partitioning key for both files. You should also ensure that the fields are of the same data type.

Choosing a partitioning key with DB2 Multisystem

For the system to process the partitioned file in the most efficient manner, the following tips should be considered when setting up or using a partitioning key:

- The best partitioning key is one that has many different values and, therefore, the partitioning activity results in an even distribution of the records of data. Customer numbers, last names, claim numbers, ZIP codes (regional mailing address codes), and telephone area codes are examples of good categories for using as partitioning keys.

Gender, because only two choices exist—male or female, is an example of a poor choice for a partitioning key. Gender would cause too much data to be distributed to a single node instead of spread across the nodes. Also, when doing a query, gender as the partitioning key would cause the system to have to process through too many records of data. It would be inefficient; another field or fields of data could narrow the scope of the query and make it much more efficient. A partitioning key based on gender is a poor choice in cases where even distribution of data is desired rather than distribution based on specific values.

When preparing to change a local file into a distributed file, you can use the HASH function to get an idea of how the data may be distributed. Because the HASH function can be used against local files and with any variety of columns, you can try different partitioning keys before actually changing the file to be distributed. For example, if you were planning to use the ZIP code field of a file,

you could run the HASH function using that field to get an idea of the number of records that will HASH to each partition number. This can help you in choosing your partitioning key fields, or in creating the partition map in your node groups.

- Do not choose a field that needs to be updated often. A restriction on partitioning key fields is that they can have their values updated only if the update does not force the record to a different node.
- Do not use many fields in the partitioning key; the best choice is to use one field. Using many fields forces the system to do more work at I/O time.
- Choose a simple data type, such as fixed-length character or integer, as your partitioning key. This consideration may help performance because the hashing is done for a single field of a simple data type.
- When choosing a partitioning key, you should consider the join and grouping criteria of the queries you typically run. For example, choosing a field that is never used as a join field for a file that is involved in joins can adversely affect join performance. See Chapter 6, “Query design for performance with DB2® Multisystem” for information on running queries involving distributed files.

Customizing the distribution of data with DB2 Multisystem

Because the system is responsible for placing the data, you do not need to know where the records actually reside. However, if you want to guarantee that certain records are always stored on a particular system, you can use the Change Node Group Attributes (CHGNODGRPA) command to specify where those records reside.

As an example, suppose you want all the records for the 55902 ZIP code to reside on your system in Minneapolis, Minnesota. When you issue the CHGNODGRPA command, you would specify the 55902 ZIP code and the system node number of the local node in Minneapolis.

At this point, the 55902 ZIP has changed node groups, but the data is still distributed as it was previously. The CHGNODGRPA command does not affect the existing files. When a partitioned file is created, the partitioned file keeps a copy of the information from the node group at that time. The node group can be changed or deleted without affecting the partitioned file. For the changes to the records that are to be redistributed to take effect, either you can recreate the distributed file using the new node group, or you can use the Change Physical File (CHGPF) command and specify the new or updated node group.

Using the CHGPF command, you can do the following:

- Redistribute an already partitioned file
- Change a partitioning key (from the telephone area code to the branch ID, for example)
- Change a local file to be a distributed file
- Change a distributed file to a local file.

Note: You must also use the CHGNODGRPA command to redistribute an already partitioned file. The CHGNODGRPA command can be optionally used with the CHGPF command to do any of the other tasks.



See “Redistribution issues for adding systems to a network” on page 33 for information on changing a local file to a distributed file or a distributed file to a local file.

Chapter 4. Scalar functions available with DB2 Multisystem

For DB2 Multisystem, new scalar functions are available for you to use when working with your distributed files. These functions help you determine how to distribute the data in your files as well as determine where the data is after the file has been distributed. When working with distributed files, database administrators may find these functions to be helpful debugging tools.

These scalar functions are PARTITION, HASH, NODENAME, and NODENUMBER. You can use these functions through SQL or the Open Query File (OPNQRYF) command.

For information about the syntax of the SQL scalar functions, see the SQL

Reference . For information about syntax of the OPNQRYF scalar functions, see the Control Language  topic in the Information Center.

- “PARTITION with DB2 Multisystem”
- “HASH with DB2 Multisystem” on page 26
- “NODENAME with DB2 Multisystem” on page 27
- “NODENUMBER with DB2 Multisystem” on page 28
- “Special registers with DB2 Multisystem” on page 28

See “Code Disclaimer Information” on page v for information pertaining to code examples.

PARTITION with DB2 Multisystem

Through the PARTITION function, you can determine the partition number where a specific row of the distributed relational database is stored. Knowing the partition number allows you to determine which node in the node group contains that partition number.

For code examples of PARTITION, see “Examples of PARTITION with DB2 Multisystem”.

Examples of PARTITION with DB2 Multisystem

- Find the PARTITION number for every row of the EMPLOYEE table.

Note: This could be used to determine if there is data skew.

SQL Statement:

```
SELECT PARTITION(CORPDATA.EMPLOYEE), LASTNAME
FROM CORPDATA.EMPLOYEE
```

OPNQRYF Command:

```
OPNQRYF FILE((CORPDATA/EMPLOYEE))
FORMAT(FNAME)
MAPFLD((PART1 '%PARTITION(1)'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

- Select the employee number (EMPNO) from the EMPLOYEE table for all rows where the partition number is equal to 100.

SQL Statement:

```
SELECT EMPNO
FROM CORPDATA.EMPLOYEE
WHERE PARTITION(CORPDATA.EMPLOYEE) = 100
```

OPNQRYF Command:

```
OPNQRYF FILE((EMPLOYEE)) QRYSLT('%PARTITION(1) *EQ 100')
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

- Join the EMPLOYEE and DEPARTMENT tables, select all rows of the result where the rows of the two tables have the same partition number.

SQL Statement:

```
SELECT *
FROM CORPDATA.EMPLOYEE X, CORPDATA.DEPARTMENT Y
WHERE PARTITION(X)=PARTITION(Y)
```

OPNQRYF Command:

```
OPNQRYF FILE((CORPDATA/EMPLOYEE) (CORPDATA/DEPARTMENT))
FORMAT(FNAME)
JFLD((1/PART1 2/PART2 *EQ))
MAPFLD((PART1 '%PARTITION(1)')
(PART2 '%PARTITION(2)'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

HASH with DB2 Multisystem

The HASH function returns the partition number by applying the hash function to the specified expressions.

For a code example of HASH, see “Example of HASH with DB2 Multisystem”.

Example of HASH with DB2 Multisystem

- Use the HASH function to determine what the partitions would be if the partitioning key was composed of EMPNO and LASTNAME. This query returns the partition number for every row in EMPLOYEE.

SQL Statement:

```
SELECT HASH(EMPNO, LASTNAME)
FROM CORPDATA.EMPLOYEE
```

OPNQRYF Command:

```
OPNQRYF FILE((CORPDATA/EMPLOYEE))
FORMAT(FNAME)
MAPFLD((HASH '%HASH(1/EMPNO, 1/LASTNAME)'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

NODENAME with DB2 Multisystem

Through the NODENAME function, you can determine the name of the relational database (RDB) where a specific row of the distributed relational database is stored. Knowing the node name allows you to determine the system name that contains the row. This can be useful in determining if you want to redistribute certain rows to a specific node.

For code examples of NODENAME, see “Examples of NODENAME with DB2 Multisystem”.

Examples of NODENAME with DB2 Multisystem

- Find the node name and the partition number for every row of the EMPLOYEE table, and the corresponding value of the EMPNO columns for each row.

SQL Statement:

```
SELECT NODENAME(CORPDATA.EMPLOYEE), PARTITION(CORPDATA.EMPLOYEE), EMPNO
FROM CORPDATA.EMPLOYEE
```

- Find the node name for every record of the EMPLOYEE table.

OPNQRYF Command:

```
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 row involved in the join originated.

SQL Statement:

```
SELECT EMPNO, NODENAME(X), NODENAME(Y)
FROM CORPDATA.EMPLOYEE X, CORPDATA.DEPARTMENT Y
WHERE X.DEPTNO=Y.DEPTNO
```

OPNQRYF Command:

```
OPNQRYF FILE((CORPDATA/EMPLOYEE) (CORPDATA/DEPARTMENT))
          FORMAT(FNAME)
          JFLD((EMPLOYEE/DEPTNO DEPARTMENT/DEPTNO *EQ))
          MAPFLD((EMPNO 'EMPLOYEE/EMPNO')
                  (NODENAME1 '%NODENAME(1)')
                  (NODENAME2 '%NODENAME(2)'))
```

- Join the EMPLOYEE and DEPARTMENT tables, select all rows of the result where the rows of the two tables are on the same node.

SQL Statement:

```
SELECT *
FROM CORPDATA.EMPLOYEE X, CORPDATA.DEPARTMENT Y
WHERE NODENAME(X)=NODENAME(Y)
```

OPNQRYF Command:

```
OPNQRYF FILE((CORPDATA/EMPLOYEE) (CORPDATA/DEPARTMENT))
          FORMAT(FNAME)
          JFLD((1/NODENAME1 2/NODENAME2 *EQ))
          MAPFLD((NODENAME1 '%NODENAME(1)')
                  (NODENAME2 '%NODENAME(2)'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

NODENUMBER with DB2 Multisystem

Through the NODENUMBER function, you can determine the node number where a specific row of the distributed relational database is stored. The node number is the unique number assigned to each node within a node group when the node group was created. Knowing the node number allows you to determine the system name that contains the row. This can be useful in determining if you want to redistribute certain rows to a specific node.

For a code example of NODENUMBER, see “Example of NODENUMBER with DB2 Multisystem”.

Example of NODENUMBER with DB2 Multisystem

- If CORPDATA.EMPLOYEE was a distributed table, then the node number for each row and the employee name would be returned.

SQL Statement:

```
SELECT NODENUMBER(CORPDATA.EMPLOYEE), LASTNAME
FROM CORPDATA.EMPLOYEE
```

OPNQRYF Command:

```
OPNQRYF FILE((CORPDATA/EMPLOYEE))
        FORMAT(FNAME)
        MAPFLD((NODENAME '%NODENUMBER(1)')
              (LNAME '1/LASTNAME'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

Special registers with DB2 Multisystem

For DB2 Multisystem, all instances of special registers are resolved on the coordinator node prior to sending the query to the remote nodes. (A **coordinator** node is the system on which the query was initiated.) This way, all nodes run the query with consistent special register values.

The following are the rules regarding special registers:

- CURRENT SERVER always returns the relational database name of the coordinator node.
- The USER special register returns the user profile that is running the job on the coordinator node.
- CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP are from the time-of-day clock at the coordinator node.
- CURRENT TIMEZONE is the value of the system value QUTCOFFSET at the coordinator node.

To return a relative record number of a row stored on a node in a distributed file, see “Relative record numbering (RRN) function with DB2 Multisystem” on page 29.

Relative record numbering (RRN) function with DB2 Multisystem

The RRN function returns the relative record number of the row stored on a node in a distributed file. This means that RRN is not unique for a distributed file. A unique record in the file is specified if you combine RRN with either NODENAME or NODENUMBER.

Chapter 5. Performance and scalability with DB2 Multisystem

With DB2 Multisystem, you can increase your database capacity, realize improvements in query performance, and provide remote database access through an easier method.

Using DB2 Multisystem, your users and your applications only need to access the file from the local system. You have no need to make any code changes to be able to access data in a distributed file versus a local file. With functions like Distributed Relational Database Architecture™ (DRDA) and distributed data management (DDM), your access must be explicitly directed to a remote file or to a remote system to be able to access that remote data. DB2 Multisystem handles the remote access in such a way that it is transparent to your users.

DB2 Multisystem also provides a simple growth path for your database expansion.

See the following topics to learn about the performance and scalability of DB2 Multisystem:

- “Why should you use DB2 Multisystem?”
- “How can DB2 Multisystem help you expand your database system?” on page 33

Why should you use DB2 Multisystem?

Performance improvements can be quite significant for certain queries. Testing has shown that for queries that have a large amount of data to be processed, but with a relatively small result set, the performance gain is almost proportional to the number of systems the file is distributed across. For example, suppose you have a 5 million (5 000 000) record file that you want to query for the top 10 revenue producers. With DB2 Multisystem, the response time for the query is cut by nearly 50% by partitioning the file equally across two systems. On three systems, the response time would be nearly one-third the time of running the query on a single system. This best case scenario does not apply to complex joins where data needs to be moved between nodes.

If a file is fairly small or is primarily used for single-record read/write processing, little or no performance gain may be realized from partitioning the file. Instead, a slight degradation in performance may occur. In these cases, query performance becomes more dependent on the speed of the physical connection. However, even in these situations, the users on all the systems in the node group still have the advantage of being able to access the data, even though it is distributed, using the traditional “local file” database methods with which they are familiar. In all environments, users have the benefits of this local-system transparency and the possible elimination of data redundancy across the systems in the node group.

Another parallelism feature is DB2 UDB Symmetric Multiprocessing. With symmetric multiprocessing (SMP), when a partitioned file is processed and if any of the systems are multiprocessor systems, you will achieve a multiplier effect in terms of performance gains. If you partitioned a file across 3 systems and each system is a 4-way processor system, the functions of DB2 Multisystem and SMP will work together. Using the previous 5 million record example, the response time is approximately one-twelfth of what it would have been had the query been run

without using any of the parallelism features. The file sizes and the details of the query can affect the improvement that you actually see.

When you do queries, the bulk of the work to run the query is done in parallel, which improves the overall performance of query processing. The system divides up the query and processes the appropriate parts of the query on the appropriate system. This makes for the most efficient processing, and it is done automatically—you do not have to specify anything to make this highly efficient processing occur.

Note: The performance of some queries may not improve, especially if a large volume of data has to be moved.

Each node only has to process the records that are physically stored on that node. If the query specifies selection against the partitioning key, the query optimizer may determine that only one node needs to be queried. In the following example, the ZIP code field is the partitioning key within the SQL statement for the ORDERS file:

```
SELECT NAME, ADDRESS, BALANCE FROM PRODLIB/ORDERS WHERE ZIP='48009'
```

When the statement is run, the optimizer determines that only one node needs to be queried. Remember that all the records that contain the 48009 ZIP code are distributed to the same node.

In the next SQL statement example, the processor capability of all the iSeries servers in the node group can be used to process the statement in parallel:

```
SELECT ZIP, SUM(BALANCE) FROM PRODLIB/ORDERS GROUP BY ZIP
```

Another advantage of having the optimizer direct I/O requests only to systems that contain pertinent data is that queries can still run if one or more of the systems are not active. An example is a file that is partitioned such that each branch of a business has its data stored on a different system. If one system is unavailable, file I/O operations can still be performed for data associated with the remaining branches. The I/O request fails for the branch that is not active.

The optimizer uses two-phase commit protocols to ensure the integrity of the data. Because multiple systems are being accessed, if you request commitment control, all of the systems will use **protected conversations**. A protected conversation means that if a system failure occurs in the middle of a transaction or in the middle of a single-database operation, all of the changes made up to that point are rolled back.

When protected conversations are used, some commitment control options are changed at the remote nodes to enhance performance. The Wait for outcome option is set to Y, and the Vote read-only permitted option is set to Y. To further enhance performance, you can use the Change Commitment Options (QTNCHGCO) API to change the Wait for outcome option to N on the system where your queries are initiated. Refer to the OS/400 APIs topic in the Information Center to understand the effects of these commitment option values.

For more information on commitment control, see the *Backup and Recovery* book.

For a method to ensure best performance, see “Performance enhancement tip with DB2 Multisystem” on page 33.

Performance enhancement tip with DB2 Multisystem

One way to ensure the best performance is to specify `*BUFFERED` for the distribute data (DSTDTA) parameter on the Override with Database File (OVRDBF) command. This tells the system to retrieve data from a distributed file as quickly as possible-potentially even at the expense of immediate updates that are to be made to the file. `DSTDTA(*BUFFERED)` is the default value for the parameter when a file is opened for read-only purposes.

The other values for the DSTDTA parameter are `*CURRENT` and `*PROTECTED`. `*CURRENT` allows updates by other users to be made to the file, but at some expense to performance. When a file is opened for update, `DSTDTA(*CURRENT)` is the default value. `*PROTECTED` gives performance that is similar to that of `*CURRENT`, but `*PROTECTED` prevents updates from being made by other users while the file is open.

How can DB2 Multisystem help you expand your database system?

By using distributed relational database files, you can more easily expand the configuration of your iSeries servers. Prior to DB2 Multisystem, if you wanted to go from one system to two systems, you had several database problems to solve. If you moved one-half of your users to a new system, you might also want to move one-half of your data to that new system. This then would force you to rewrite all of your database-related applications, because the applications have to know where the data resides. After rewriting the applications, you have to use some remote access, such as Distributed Relational Database Architecture (DRDA) or distributed data management (DDM), to access the files across systems. Otherwise, some data replication function would have to be used. If you did do this, then multiple copies of the data would exist, more storage would be used, and the systems would also have to do the work of keeping the multiple copies of the files at concurrent levels.

With DB2 Multisystem, the process of adding new systems to your configuration is greatly simplified. The database files are partitioned across the systems. Then the applications are moved to the new system. The applications are unchanged; you do not have to make any programming changes to your applications. Your users can now run on the new system and immediately have access to the same data. If more growth is needed later, you can redistribute the file across the new node group that includes the additional systems.

See “Redistribution issues for adding systems to a network” for issues you need to be aware of when adding systems to a network.

Redistribution issues for adding systems to a network

The redistribution of a file across a node group is a fairly simple process. You can use the Change Physical File (CHGPF) command to specify either a new node group for a file or a new partitioning key for a file. The CHGPF command can be used to make a local file into a distributed file, to make a distributed file into a local file, or to redistribute a distributed file either across a different set of nodes or with a different partitioning key.

You should be aware that the process of redistribution may involve the movement of nearly every record in the file. For very large files, this can be a long process during which the data in the file is unavailable. You should not do file redistribution often or without proper planning.

To change a local physical file into a distributed file, you must specify the node group (NODGRP) and partitioning key (PTNKEY) parameters on the CHGPF command. Issuing this command changes the file to be distributed across the nodes in the node group, and any existing data is also distributed using the partitioning key specified on the PTNKEY parameter.

To change a distributed file into a local file, you must specify NODGRP(*NONE) on the CHGPF command. This deletes all the remote pieces of the file and forces all of the data back to the local system.

To change the partitioning key of a distributed file, specify the desired fields on the PTNKEY parameter of the CHGPF command. This does not affect which systems the file is distributed over. It does cause all of the data to be redistributed, because the hashing algorithm needs to be applied to a new partitioning key.

To specify a new set of systems over which the file should be distributed, specify a node group name on the node group (NODGRP) parameter of the CHGPF command. This results in the file being distributed over this new set of systems. You can specify a new partitioning key on the PTNKEY parameter. If the PTNKEY parameter is not specified or if *SAME is specified, the existing partitioning key is used.

The CHGPF command handles creating new pieces of the file if the node group had new systems added to it. The CHGPF command handles deleting pieces of the file if a system is not in the new node group. Note that if you want to delete and re-create a node group and then use the CRTPF command to redistribute the file, you must specify the node group name on the NODGRP parameter of the CHGPF command, even if the node group name is the same as the one that was used when the file was first created. This indicates that you do want the system to look at the node group and to redistribute the file. However, if you specify a node group on the NODGRP parameter and the system recognizes that it is identical to the node group currently stored in the file, then no redistribution occurs unless you also specified PTNKEY.

For files with referential constraints, you should do the following if you want to use the CHGPF command to make the parent file and the dependent file distributed files:




- Remove the referential constraint. If you do not remove the constraint, the file that you distribute first will encounter a constraint error, because the other file in the referential constraint relationship is not yet a distributed file.
- Use the CHGPF command to make both files distributed files.
- Add the referential constraint again.

Chapter 6. Query design for performance with DB2® Multisystem

This chapter provides you with some guidelines for designing queries so that they use query resources more efficiently when you run queries that use distributed files. This chapter also discusses how queries that use distributed files are implemented. This information can be used to tune queries so that they run more efficiently in a distributed environment.

Distributed files can be queried using SQL, the Open Query File (OPNQRYF) command, or any query interface on the system. The queries can be single file queries or join queries. You can use a combination of distributed and local files in a join.

This chapter assumes that you are familiar with running and optimizing queries in a nondistributed environment. If you want more information on these topics:

- SQL users should refer to SQL Reference  and SQL Programming Concepts
- Non-SQL users should refer to Database Programming  and the CL Reference  topic.

This chapter also shows you how to improve the performance of distributed queries by exploiting parallelism and minimizing data movement.

- “Optimization overview with DB2 Multisystem” on page 36
- “Implementation and optimization of a single file query with DB2 Multisystem” on page 36
- “Implementation and optimization of record ordering with DB2 Multisystem” on page 38
- “Implementation and optimization of the UNION and DISTINCT clauses with DB2 Multisystem” on page 39
- “Processing of the DSTDTA and ALWCPYDTA parameters with DB2 Multisystem” on page 39
- “Implementation and optimization of joins with DB2 Multisystem” on page 39
- “Implementation and optimization of grouping with DB2 Multisystem” on page 45
- “Subquery support with DB2 Multisystem” on page 47
- “Access plans with DB2 Multisystem” on page 47
- “Reusable open data paths (ODPs) with DB2 Multisystem” on page 47
- “Temporary result writer with DB2 Multisystem” on page 48
- “Optimizer messages with DB2 Multisystem” on page 50
- “Changes to the change query attributes (CHGQRYA) command with DB2 Multisystem” on page 52
- “Summary of performance considerations” on page 54

See “Code Disclaimer Information” on page v for information pertaining to code examples.

Optimization overview with DB2 Multisystem

Distributed queries are optimized at the distributed level and the local level:

- Optimization at the distributed level focuses on dividing the query into the most efficient steps and determining which nodes will process those steps.

The distributed optimizer is distinctive to distributed queries. The distributed optimizer is discussed in this chapter.

- Optimization at the local (step) level is the same optimization that occurs in a nondistributed environment; it is the optimizer with which you are probably familiar. Local-level optimization is discussed minimally in this chapter.

A basic assumption of distributed optimization is that the number of records stored at each data node is approximately equal and that all of the systems of the distributed query are of similar configurations. The decisions made by the distributed optimizer are based on the system and the data statistics of the coordinator node system.

If a distributed query requires more than one step, a temporary result file is used. A **temporary result file** is a system-created temporary file (stored in library QRECOVERY) that is used to contain the results of a particular query step. The contents of the temporary result file are used as input to the next query step.

Implementation and optimization of a single file query with DB2 Multisystem

Querying a single distributed file results in the data on all of the nodes or on a subset of the nodes of the distributed file being queried. To do this, the system where the query was specified, the coordinator node, determines the nodes of the file to which to send the query. Those nodes run the query and return the queried records to the coordinator node.

All of the examples in this chapter use the following distributed files: DEPARTMENT and EMPLOYEE. The node group for these files consists of SYSA, SYSB, and SYSC. The data is partitioned on the department number.

The following SQL statement creates the DEPARTMENT distributed file.

```
CREATE TABLE DEPARTMENT
  (DEPTNO CHAR(3) NOT NULL,
   DEPTNAME VARCHAR(20) NOT NULL,
   MGRNO CHAR(6),
   ADMRDEPT CHAR(3) NOT NULL)
  IN NODGRP1 PARTITIONING KEY(DEPTNO)
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

This is a display of the DEPARTMENT table. Its headings are node, record number, DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT.

Node	Record Number	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
SYSA	1	A00	Support Services	000010	A00
SYSB	2	A01	Planning	000010	A00

This is a display of the DEPARTMENT table. Its headings are node, record number, DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT.

Node	Record Number	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
SYSC	3	B00	Accounting	000050	B00
SYSA	4	B01	Programming	000050	B00

The following SQL statement creates the EMPLOYEE distributed file.

```
CREATE TABLE EMPLOYEE
  (EMPNO CHAR(6) NOT NULL,
  FIRSTNAME VARCHAR(12) NOT NULL,
  LASTNAME VARCHAR(15) NOT NULL,
  WORKDEPT CHAR(3) NOT NULL,
  JOB CHAR(8),
  SALARY DECIMAL(9,2))
IN NODGRP1 PARTITIONING KEY(WORKDEPT)
```

See "Code Disclaimer Information" on page v for information pertaining to code examples.

Node	Record Number	EMPNO	FIRSTNAME	LASTNAME	WORK DEPT	JOB	SALARY
SYSA	1	000010	Christine	Haas	A00	Manager	41250
SYSA	2	000020	Sally	Kwan	A00	Clerk	25000
SYSB	3	000030	John	Geyer	A01	Planner	35150
SYSB	4	000040	Irving	Stern	A01	Clerk	32320
SYSC	5	000050	Michael	Thompson	B00	Manager	38440
SYSC	6	000060	Eileen	Henderson	B00	Accountant	33790
SYSA	7	000070	Jennifer	Lutz	B01	Programmer	42325
SYSA	8	000080	David	White	B01	Programmer	36450

What follows is a query using the above defined distributed file EMPLOYEE, with index EMPIDX created over the field SALARY. The query is entered on SYSA.

```
SQL: SELECT * FROM EMPLOYEE WHERE SALARY > 40000
```

```
OPNQRYF: OPNQRYF FILE((EMPLOYEE)) QRYSLT('SALARY > 40000')
```

In this case, SYSA sends the above query to all the nodes of EMPLOYEE, including SYSA. Each node runs the query and returns the records to SYSA. Because a distributed index exists on field SALARY of file EMPLOYEE, optimization that is done on each node decides whether to use the index.

In the next example, the query is specified on SYSA, but the query is sent to a subset of the nodes where the EMPLOYEE file exists. In this case, the query is run locally on SYSA only.

```
SQL: SELECT * FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

```
OPNQRYF: OPNQRYF FILE((EMPLOYEE)) QRYSLT('WORKDEPT = 'A00')
```

See "Code Disclaimer Information" on page v for information pertaining to code examples.

The distributed query optimizer determines that there is an isolatable record selection, WORKDEPT = 'A00', involving the partitioning key, WORKDEPT, for this query. The optimizer hashes the value 'A00' and based on the hash value,

finds the node at which all of the records satisfying this condition are located. In this case, all of the records satisfying this condition are on SYSA, thus the query is sent only to that node. Because the query originated on SYSA, the query is run locally on SYSA.

The following will subset the number of nodes at which a query runs:

- All of fields of the partitioning key must be isolatable record selection
- All of the predicates must use the equal (=) operator
- All of the fields of the partitioning key must be compared to a literal

Note: For performance reasons, you should specify record selection predicates that match the partitioning key in order to direct the query to a particular node. Record selection with scalar functions of NODENAME, PARTITION, and NODENUMBER can also direct the query to specific nodes.

Implementation and optimization of record ordering with DB2 Multisystem

When ordering is specified on a query, the ordering criteria is sent along with the query, so that each node can perform the ordering in parallel. Whether a final merge or a sort is performed on the coordinator node is dependent on the type of query that you specify.

A merge of the ordered records received from each node is the most optimal. A merge occurs as the records are received by the coordinator node. The main performance advantage that a merge has over a sort is that the records can be returned without having to sort all of the records from every node.

A sort of the ordered records received from each node causes all of the records from each node to be read, sorted, and written to a temporary result file before any records are returned.

A merge can occur if ordering is specified and no UNION and no final grouping are required. Otherwise, for ordering queries, a final sort is performed on the coordinator node.

The allow copy data (ALWCPYDTA) parameter affects how each node of the distributed query processes the ordering criteria. The ALWCPYDTA parameter is specified on the Open Query File (OPNQRYF) and Start SQL (STRSQL) CL commands and also on the Create SQLxxx (CRTSQLxxx) precompiler commands:

- ALWCPYDTA(*OPTIMIZE) allows each node to choose to use either a sort or an index to implement the ordering criteria. This option is the most optimal.
- For the OPNQRYF command and the query API (QQQQRY), ALWCPYDTA(*YES) or ALWCPYDTA(*NO) enforces that each node use an index that exactly matches the specified ordering fields. This is more restrictive than how the optimizer processes ordering for local files.

Implementation and optimization of the UNION and DISTINCT clauses with DB2 Multisystem

If a unioned SELECT statement refers to a distributed file, the statement is processed as a distributed query. The processing of the statement can occur in parallel. However, the records from each unioned SELECT are brought back to the coordinator node to perform the union operation. In this regard, the union operators are processed serially.

If an ORDER BY clause is specified with a union query, all of the records from each node are received on the coordinator node and are sorted before any records are returned.

When the DISTINCT clause is specified for a distributed query, adding an ORDER BY clause returns records faster than if no ORDER BY clause was specified. DISTINCT with an ORDER BY allows each node to order the records in parallel. A final merge on the coordinator node reads the ordered records from each node, merges the records in the proper order, and eliminates duplicate records without having to do a final sort.

When the DISTINCT clause is specified without an ORDER BY clause, all of the records from each node are sent to the coordinator node where a sort is performed. Duplicate records are eliminated as the sorted records are returned.

Processing of the DSTDTA and ALWCPYDTA parameters with DB2 Multisystem

The allow copy data (ALWCPYDTA) parameter can change the value specified for the distribute data (DSTDTA) parameter of the Override Database File (OVRDBF) command.

If you specified to use live data (DSTDTA(*CURRENT)) on the override command, but either of the following are true:

- A temporary copy was required and ALWCPYDTA(*YES) was specified
- Or a temporary copy was chosen for better performance and ALWCPYDTA(*OPTIMIZE) was specified

then DSTDTA is changed to DSTDTA(*BUFFERED).

If DSTDTA(*BUFFERED) is not acceptable to you and the query does not require a temporary copy, then you need to specify ALWCPYDTA(*YES) to keep DSTDTA(*CURRENT) in effect.

Implementation and optimization of joins with DB2 Multisystem

In addition to the performance considerations for nondistributed join queries, more performance considerations exist for queries involving distributed files. Joins can be performed only when the data is partition compatible. The distributed query optimizer generates a plan that makes data partition compatible, which may involve moving data between nodes.

Data is **partition compatible** when the data in the partitioning keys of both files uses the same node group and hashes to the same node. For example, the same numeric value stored in either a large-integer field or a small-integer field hashes to the same value.

The data types that follow are partition compatible:

- Large integer (4 byte), small integer (2 byte), packed decimal, and zoned numeric.
- Fixed-length and varying-length SBCS character and DBCS-open, -either or -only.
- Fixed-length and varying-length graphic.

Date, time, timestamp, and floating-point numeric data types are not partition compatible because they cannot be partitioning keys.

Joins involving distributed files are classified into four types: co-located, directed, repartitioned, and broadcast. The following sections define the types of joins and give examples of the different join types.

- “Co-located join with DB2 Multisystem”
- “Directed join with DB2 Multisystem” on page 41
- “Repartitioned join with DB2 Multisystem” on page 42
- “Broadcast join with DB2 Multisystem” on page 43

Also see the “Join optimization with DB2 Multisystem” on page 44 for information on how distributed query optimizer behaves with joins.

Co-located join with DB2 Multisystem

In a co-located join, corresponding records of files being joined exist on the same node. The values of the partitioning key of the files being joined are partition compatible. No data needs to be moved to another node to perform the join. This method is only valid for queries where all of the fields of the partitioning keys are join fields and the join operator is the = (equals) operator. Also, the nth field (where n=1 to the number of fields in the partitioning key) of the partitioning key of the first file must be joined to the nth field of the partitioning key of the second file, and the data types of the nth fields must be partition compatible. Note that all of the fields of the partitioning key must be involved in the join. Additional join predicates that do not contain fields of the partitioning key do not affect your ability to do a co-located join.

In the following example, because the join predicate involves the partitioning key fields of both files and the fields are partition compatible, a co-located join can be performed. This implies that matching values of DEPTNO and WORKDEPT are located on the same node.

```
SQL:      SELECT DEPTNAME, FIRSTNME, LASTNAME
           FROM DEPARTMENT, EMPLOYEE
           WHERE DEPTNO=WORKDEPT
OPNQRYF:  OPNQRYF FILE((DEPARTMENT) (EMPLOYEE))
           FORMAT(JOINFMT)
           JFLD((DEPTNO WORKDEPT *EQ))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

Records returned by this query:

This is a display of the query results. The headings on this table are DEPTNAME, FIRSTNAME, and LASTNAME.

DEPTNAME	FIRSTNME	LASTNAME
Support Services	Christine	Haas

This is a display of the query results. The headings on this table are DEPTNAME, FIRSTNAME, and LASTNAME.

DEPTNAME	FIRSTNME	LASTNAME
Support Services	Sally	Kwan
Planning	John	Geyer
Planning	Irving	Stern
Accounting	Michael	Thompson
Accounting	Eileen	Henderson
Programming	Jennifer	Lutz
Programming	David	White

In the following example, the additional join predicate MGRNO=EMPNO does not affect the ability to perform a co-located join, because the partitioning keys are still involved in a join predicate.

```
SQL:      SELECT DEPTNAME, FIRSTNME, LASTNAME
          FROM DEPARTMENT, EMPLOYEE
          WHERE DEPTNO=WORKDEPT AND MGRNO=EMPNO
OPNQRYF:  OPNQRYF FILE((DEPARTMENT) (EMPLOYEE))
          FORMAT(JOINFMT)
          JFLD((DEPTNO WORKDEPT *EQ) (MGRNO EMPNO *EQ))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

Records returned by this query:

DEPTNAME	FIRSTNME	LASTNAME
Support Services	Christine	Haas
Accounting	Michael	Thompson

Directed join with DB2 Multisystem

In the directed join, the partitioning keys of at least one of the files are used as the join fields. The join fields do not match the partitioning keys of the other files. Records of one file are directed to or sent to the nodes of the second file based on the hashing of the join field values using the partition map and node group of the second file. Once the records have been moved to the nodes of the second file through a temporary distributed file, a co-located join is used to join the data. This method is valid only for equijoin queries where all fields of the partitioning key are join fields for at least one of the files.

In the following query, join field (WORKDEPT) is the partitioning key for file EMPLOYEE; however, join field (ADMRDEPT) is not the partitioning key for DEPARTMENT. If the join was attempted without moving the data, result records would be missing because record 2 of DEPARTMENT should be joined to records 1 and 2 of EMPLOYEE and these records are stored on different nodes.

```
SQL:      SELECT DEPTNAME, FIRSTNME, LASTNAME
          FROM DEPARTMENT, EMPLOYEE
          WHERE ADMRDEPT = WORKDEPT AND JOB = 'Manager'
OPNQRYF:  OPNQRYF FILE((DEPARTMENT) (EMPLOYEE))
          FORMAT(JOINFMT)
          QRYSLT('JOB *EQ 'Manager')
          JFLD((ADMRDEPT WORKDEPT *EQ))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

The records of DEPARTMENT that are needed to run the query are read, and the data in ADMRDEPT is hashed using the partitioning map and the node group of EMPLOYEE. A temporary file is created and looks like the following:

Old Node	New Node	DEPTNAME	ADMRDEPT (New Partitioning Key)
SYSA	SYSA	Support Services	A00
SYSB	SYSA	Planning	A00
SYSC	SYSC	Accounting	B00
SYSA	SYSC	Programming	B00

This temporary table is joined to EMPLOYEE to produce the following. The join works because ADMRDEPT is partition compatible with WORKDEPT.

DEPTNAME	FIRSTNME	LASTNAME
Support Services	Christine	Haas
Planning	Christine	Haas
Accounting	Michael	Thompson
Programming	Michael	Thompson

Repartitioned join with DB2 Multisystem

In a repartitioned join, the partitioning keys of the files are not used as the join fields. Records of both files must be moved by hashing the join field values of each of the files. Because neither of the files’ partitioning key fields are included in the join criteria, the files must be repartitioned by hashing on a new partitioning key that includes one or more of the join fields. This method is valid only for equijoin queries.

```
SQL:      SELECT DEPTNAME, FIRSTNME, LASTNAME
           FROM DEPARTMENT, EMPLOYEE
           WHERE MGRNO = EMPNO

OPNQRYF:  OPNQRYF FILE((DEPARTMENT) (EMPLOYEE))
           FORMAT(JOINFMT)
           JFLD((MGRNO EMPNO *EQ))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

In this example, the data must be redistributed because neither MGRNO nor EMPNO is a partitioning key.

Data from DEPARTMENT is redistributed:

Old Node	New Node	DEPTNAME	MGRNO (New Partitioning Key)
SYSA	SYSB	Support Services	000010
SYSB	SYSB	Planning	000010
SYSC	SYSC	Accounting	000050
SYSA	SYSC	Programming	000050

Data from EMPLOYEE is redistributed:

Old Node	New Node	FIRSTNME	LASTNAME	EMPNO (New Partitioning Key)
SYSA	SYSB	Christine	Haas	000010
SYSA	SYSC	Sally	Kwan	000020
SYSB	SYSA	John	Geyer	000030
SYSB	SYSB	Irving	Stern	000040
SYSC	SYSC	Michael	Thompson	000050
SYSC	SYSA	Eileen	Henderson	000060
SYSA	SYSB	Jennifer	Lutz	000070
SYSA	SYSC	David	White	000080

Records returned by this query:

DEPTNAME	FIRSTNME	LASTNAME
Support Services	Christine	Haas
Planning	Christine	Haas
Accounting	Michael	Thompson
Programming	Michael	Thompson

Broadcast join with DB2 Multisystem

In a broadcast join, all of the selected records of one file are sent or broadcast to all the nodes of the other file before the join is performed. This is the join method that is used for all nonequijoin queries. This method is also used when the join criteria uses fields that have a data type of date, time, timestamp, or floating-point numeric.

In the following example, the distributed query optimizer decides to broadcast EMPLOYEE, because applying the selection JOB = 'Manager' results in broadcasting a smaller set of records. The temporary file at each node in the node group contains all the selected records. (Records are duplicated at each node.)

```
SQL:      SELECT DEPTNAME, FIRSTNME, LASTNAME
          FROM DEPARTMENT, EMPLOYEE
          WHERE DEPTNO <> WORKDEPT AND JOB = 'Manager'

OPNQRYF:  OPNQRYF FILE((DEPARTMENT) (EMPLOYEE))
          FORMAT(JOINFMT)
          QRYSLT('JOB *EQ 'Manager')
          JFLD((DEPTNO WORKDEPT *NE))
```

See "Code Disclaimer Information" on page v for information pertaining to code examples.

The distributed query optimizer sends the following two selected records to each node:

Old Node	New Node	FIRSTNME	LASTNAME	WORKDEPT
SYSA	SYSA, SYSB, SYSC	Christine	Haas	A00
SYSC	SYSA, SYSB, SYSC	Michael	Thompson	B00

Records returned by this query:

DEPTNAME	FIRSTNME	LASTNAME
Support Services	Michael	Thompson
Planning	Christine	Haas
Planning	Michael	Thompson
Accounting	Christine	Haas
Programming	Christine	Haas
Programming	Michael	Thompson

Join optimization with DB2 Multisystem

The distributed query optimizer generates a plan to join distributed files. The distributed query optimizer looks at file sizes, expected number of records selected for each file, and the type of distributed joins that are possible; and then the optimizer breaks the query into multiple steps. Each step creates an intermediate result file that is used as input for the next step.

During optimization, a cost is calculated for each join step based on the type of distributed join. The cost reflects, in part, the amount of data movement required for that join step. The cost is used to determine the final distributed plan.

As much processing as possible is completed during each step; for example, record selection isolated to a given step is performed during that step, and as many files as possible are joined for each step. Each join step may involve more than one type of distributed join. A co-located join and a directed join may be combined into one co-located join by directing the necessary file first. A directed join and a repartitioned join may be combined by directing all the files first and then performing the join. Note that directed and repartitioned joins are really just a co-located join, with one or more files being directed before the join occurs.

When joining distributed files with local files, the distributed query optimizer calculates a cost, similar to the cost calculated when joining distributed files. Based on this cost, the distributed query optimizer may choose to do one of the following:

- Broadcast all of the local files to the data nodes of the distributed file and perform a co-located join.
- Broadcast all of the local and distributed files to the data nodes of the largest distributed file and perform a co-located join.
- Direct the distributed files back to the coordinator node and perform the join there.

To partition keys over join fields, see “Partitioning keys over join fields with DB2 Multisystem”.

Partitioning keys over join fields with DB2 Multisystem

From the preceding sections on the types of joins, you can see that data movement is required for all distributed join types except a co-located join. To eliminate the need for data movement and to maximize performance, all queries should be written so that a co-located join is possible. In other words, the partitioning keys of the distributed files should match the fields used to join the files together. For queries that are run frequently, it is more important to have the partitioning keys match the join fields than it is to match the ordering or the grouping criteria.

Implementation and optimization of grouping with DB2 Multisystem

The implementation method for grouping in queries that use distributed files is dependent on whether the partitioning key is included in the grouping criteria. Grouping is implemented using either one-step grouping or two-step grouping:

- “One-step grouping with DB2 Multisystem”
- “Two-step grouping with DB2 Multisystem”

For information on usings both grouping and joins, see “Grouping and joins with DB2 Multisystem” on page 46.

One-step grouping with DB2 Multisystem

If all the fields from the partitioning key are GROUP BY fields, then grouping can be performed using one-step grouping, because all of the data for the group is on the same node. The following is an example of one-step grouping:

```
SQL:      SELECT WORKDEPT, AVG(SALARY)
           FROM EMPLOYEE
           GROUP BY WORKDEPT

OPNQRYF:  OPNQRYF FILE((EMPLOYEE)) FORMAT(GRPFMT)
           GRPFLD(WORKDEPT)
           MAPFLD((AVGSAL '%AVG(SALARY)'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

Because WORKDEPT is both the partitioning key and the grouping field, all like values of WORKDEPT are on the same nodes; for example, all values of A00 are on SYSA, all values of A01 are on SYSB, all values of B00 are on SYSC, and all values of B01 are on SYSA. The grouping is performed in parallel on all 3 nodes.

To implement one-step grouping, all of the fields of the partitioning key must be grouping fields. Additional nonpartitioning key fields may also be grouping fields.

Two-step grouping with DB2 Multisystem

If the partitioning key is not included in the grouping fields, then grouping must be done using two-step grouping, because the like values of a field are not located on the same node. The following is an example of two-step grouping:

```
SQL:      SELECT JOB, AVG(SALARY)
           FROM EMPLOYEE
           GROUP BY JOB

OPNQRYF:  OPNQRYF FILE((EMPLOYEE)) FORMAT(GRPFMT2)
           GRPFLD(JOB)
           MAPFLD((AVGSAL '%AVG(SALARY)'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

In this example, note that for the group where JOB is ‘Clerk’, the value of ‘Clerk’ is on 2 different nodes in the EMPLOYEE distributed file. Grouping is implemented by first running grouping in parallel on all 3 nodes. This results in an initial grouping which is placed in a temporary file at the coordinator node. The query is modified, and the grouping is run again at the coordinator node to get the final set of grouping results.

Whole-file grouping (no group by fields) is always implemented using two steps.

If the query contains either a HAVING clause or the group selection expression (GRPSLT) parameter on the OPNQRYF command, all groups from the first grouping step are returned to the coordinator node. The HAVING clause or the GRPSLT parameter is then processed as part of second grouping step.

If the query contains a DISTINCT column (aggregate) function and two-step grouping is required, no grouping is done in the first step. Instead, all records are returned to the coordinator node, and all grouping is run at the coordinator node as part of the second step.

Grouping and joins with DB2 Multisystem

If the query contains a join, the partitioning key used to determine the type of grouping that can be implemented is based on any repartitioning of data that was required to implement the join.

In the following example, a repartitioned join is performed prior to the grouping, which results in a new partitioning key of MGRNO. Because MGRNO is now the partitioning key, grouping can be performed using one-step grouping.

```
SQL:      SELECT MGRNO, COUNT(*)
           FROM DEPARTMENT, EMPLOYEE
           WHERE MGRNO = EMPNO
           GROUP BY MGRNO

OPNQRYF:  OPNQRYF FILE((DEPARTMENT) (EMPLOYEE)) FORMAT(GRPFMT2)
           JFLD((MGRNO EMPNO *EQ))
           GRPFLD(MGRNO)
           MAPFLD((CNTMGR '%COUNT'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

In the following example, a repartitioned join is performed prior to the grouping, which results in a new partitioning key of EMPNO. Because EMPNO is now the partitioning key instead of WORKDEPT, grouping cannot be performed using one-step grouping.

```
SQL:      SELECT WORKDEPT, COUNT(*)
           FROM DEPARTMENT, EMPLOYEE
           WHERE MGRNO = EMPNO
           GROUP BY WORKDEPT

OPNQRYF:  OPNQRYF FILE((DEPARTMENT) (EMPLOYEE)) FORMAT(GRPFMT3)
           JFLD((MGRNO EMPNO *EQ))
           GRPFLD(WORKDEPT)
           MAPFLD((CNTDEPT '%COUNT'))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

Subquery support with DB2 Multisystem

Distributed files can be specified in subqueries. A subquery can include search conditions of its own, and these search conditions can, in turn, include subqueries. Therefore, an SQL statement can contain a hierarchy of subqueries. Those elements of the hierarchy that contain subqueries are at a higher level than the subqueries

they contain. Refer to SQL Programming Concepts  for more information about using subqueries.

Access plans with DB2 Multisystem

The access plans stored for queries that refer to distributed files are different from the access plans stored for local files. The access plan stored for a distributed query is the distributed access plan that contains information on how the query is split into multiple steps and on the nodes on which each step of the query is run. The information on how the step is implemented locally at each node is not saved in the access plan; this information is built at run time.

Reusable open data paths (ODPs) with DB2 Multisystem

Reusable open data paths (ODPs) have special considerations for distributed queries. Like most other aspects of distributed queries, ODPs have two levels: distributed and local.

The distributed ODP is the coordinating ODP. A distributed ODP associates the query to the user and controls the local ODPs. Local ODPs are located on each system involved in the query, and they take requests through the distributed ODP.

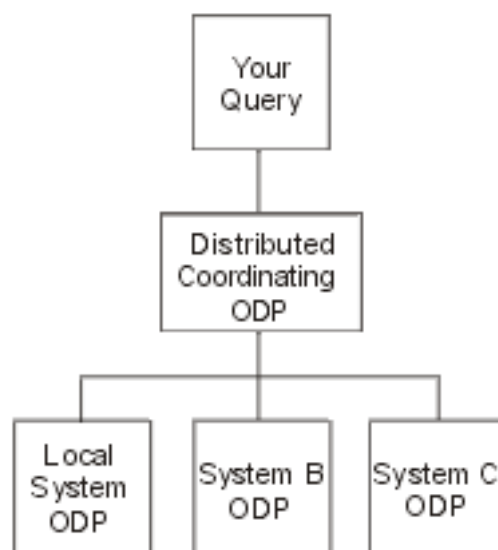


Figure 7. Example of an Open Data Path (ODP)

For example, if a request is made to perform an SQL FETCH, the request is made against the distributed ODP. The system then takes that request and performs the appropriate record retrieval against the local ODPs.

With distributed queries, it is possible for the distributed ODP to be reusable, yet for one or more of the local ODPs to be nonreusable; however, if the distributed query ODP is nonreusable, the local ODPs are always nonreusable. This is allowed so that:

- Each local system can decide the best way to open its local ODP (reusable versus nonreusable)
- Regardless of the local ODP methods, the distributed ODP can be opened as reusable as much as possible in order to maintain active resources, such as communications.

The system tries to make the distributed ODP reusable whenever possible, even when an underlying local ODP may not be reusable. If this occurs, the system handles the ODP refresh as follows:

- Cycles through each local ODP
- Performs a refresh of a reusable local ODP
- Performs a “hard” close and reopen of a nonreusable ODP
- Completes any remaining refresh of the distributed ODP itself that is needed

The distributed ODP is reusable more often than local ODPs, because the distributed ODP is not affected by some of the things that make local ODPs nonreusable, such as a host variable in a LIKE clause or the optimizer choosing nonreusable so that an index-from-index create can be performed. The cases that would make distributed ODPs nonreusable are a subset of those that affect local ODPs. This subset includes the following:

- The use of temporary files other than for sorting. These are called multi-step distributed queries, and the optimizer debug message CPI4343 is signalled for these cases.
- Library list changes, which can affect the files being queried.
- OVRDBF changes, which affects the files being queried.
- Value changes for special registers USER or CURRENT TIMEZONE.
- Job CCSID changes.
- The Reclaim Resources (RCLRSC) command being issued.

The reusability of the local ODP is affected by the same conditions that already exist for nondistributed query ODPs. Therefore, the same considerations apply to them as apply to local query ODPs.

Temporary result writer with DB2 Multisystem

Temporary result writers (temp writers) are system-initiated jobs that are always active. On the system, temp writers are paired jobs called QQQTEMP1 and QQQTEMP2. Temp writers handle requests from jobs that are running queries. These requests consist of a query (of the query step) to run and the name of a system temporary file to fill from the results of the query. The temp writer processes the request and fills the temporary file. This intermediate temporary file is then used by the requesting job to complete the original query.

The following example shows a query that requires a temporary result writer and the steps needed to process the query.

```
SQL:      SELECT COUNT(*)
          FROM DEPARTMENT a, EMPLOYEE b
          WHERE a.ADMRDEPT = b.WORKDEPT
          AND b.JOB = 'Manager'
```



```

OPNQRYF: OPNQRYF FILE((DEPARTMENT) (EMPLOYEE))
          FORMAT(FMTFILE)
          MAPFLD((CNTFLD '%COUNT'))
          JFLD((1/ADMRDEPT 2/WORKDEPT))
          QRYSLT('2/JOB = 'Manager')

```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

WORKDEPT is the partitioning key for EMPLOYEE, but ADMRDEPT is not the partitioning key for DEPARTMENT. Because the query must be processed in two steps, the optimizer splits the query into the following steps:

```

INSERT INTO SYS_TEMP_FILE
SELECT a.DEPTNAME, a.ADMRDEPT
FROM DEPARTMENT a

```

and

```

SELECT COUNT(*) FROM SYS_TEMP_FILE x, EMPLOYEE b
WHERE x.ADMRDEPT = b.WORKDEPT AND b.JOB = 'Manager'

```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

If a temp writer is allowed for the job (controlled by the Change Query Attributes (CHGQRYA) options), the optimizer does the following:

1. Creates the temporary file (SYS_TEMP_FILE) into library QRECOVERY
2. Sends the request that populates SYS_TEMP_FILE to the temp writer
3. Continues to finish opening the final query (while the temp writer is filling the temporary file)
4. Once the final query is opened, waits until the temp writer has finished filling the temporary file before returning control to its caller.

See “Changes to the change query attributes (CHGQRYA) command with DB2 Multisystem” on page 52 for information on this command.

For more information on using temporary result writers with DB2 Multisystem, see the following:

- “Temp writer job: Advantages with DB2 Multisystem”
- “Temp Writer Job: Disadvantages with DB2 Multisystem” on page 50
- “Control of the temp writer with DB2 Multisystem” on page 50

Temp writer job: Advantages with DB2 Multisystem

The advantage of using a temp writer job in processing a request is that the temp writer can process its request at the same time (in parallel) that the main job is processing another step of the query.

The following are some performance advantages of using a temp writer:

- The temp writer can use full SMP parallel support in completing its query step, while the main job can continue with more complex steps that do not lend themselves as easily to parallel processing (such as query optimization, index analysis, and so on).
- Because distributed file processing requires communications interaction, a considerable amount of time usually spent waiting for sending and receiving can be offloaded to the temp writer, which leaves the main job to do other work.

Temp Writer Job: Disadvantages with DB2 Multisystem

The temp writer also has disadvantages that must be considered when determining its usefulness for queries:

- The temp writer is a separate job. Consequently, it can encounter conflicts with the main job, such as:
 - The main job may have the file locked to itself. In this case, the temp writer cannot access the files and cannot complete its query step.
 - The main job may have created the distributed file under commitment control and has not yet committed the create. In this case, the temp writer cannot access the file.
- The temp writer may encounter a situation that it cannot handle in the same way as the main job. For example, if an inquiry message is signalled, the temp writer may have to cancel, whereas the main job can choose to ignore the message and continue on.
- Temp writers are shared by all jobs on the system. If several jobs have requests to the temp writers, the requests may queue up while the writers attempt to process them.

Note: The system is shipped with three, active temp writer job pairs.

- Attempting to analyze a query (through debug messages, for example) can be complicated if a temp writer is involved (because a step of the query is run in a separate job).

Note: The system does not allow the temp writer to be used for queries running under commitment control *CS or *ALL. This is because the main job may have records locked in the file that would cause the temp writer to be locked out of these records and not be able to finish.

Control of the temp writer with DB2 Multisystem

By default, queries do not use the temp writer. Temp writer usage, however, can be enabled by using the Change Query Attributes (CHGQRYA) command.

The asynchronous job usage (ASYN CJ) parameter on the CHGQRYA command is used to control the usage of the temp writer. The ASYN CJ parameter has the following applicable options:

- *DIST or *ANY allows the temp writer jobs to be used for queries involving distributed files.
- *LOCAL or *NONE prevents the temp writer from being used for queries of distributed files.

Optimizer messages with DB2 Multisystem

The OS/400[®] distributed query optimizer provides you with information messages on the current query processing when the job is in debug mode. These messages, which show how the distributed query was processed, are in addition to the existing optimizer messages. These messages appear for the Open Query File (OPNQRYF) command, DB2 UDB Query Manager and SQL Development Kit, interactive SQL, embedded SQL, and in any iSeries server high-level language (HLL). Every message appears in the job log; you only need to put your job into debug mode.

You can evaluate the performance of your distributed query by using the informational messages put in the job log by the database manager. The database

manager may send any of the following distributed messages or existing optimizer messages when appropriate. The ampersand variables (&1, &X) are replacement variables that contain either an object name or another substitution value when the message appears in the job log:

- CPI4341 Performing distributed query.
- CPI4342 Performing distributed join for query.
- CPI4343 Optimizer debug messages for distributed query step &1 of &2.
- CPI4345 Temporary distributed result file &4 built for query.

These messages provide feedback on how a distributed query was run and, in some cases, indicate the improvements that can be made to help the query run faster. The causes and user responses for the messages are paraphrased below. The actual message help is more complete and should be used when trying to determine the meaning and responses for each message.

A detailed explanation for each message follows:

CPI4341

Performing distributed query.

This message indicates that a single distributed file was queried and was not processed in multiple steps. This message lists the nodes of the file where the query was run.

CPI4342

Performing distributed join for query.

This message indicates that a distributed join occurred. This message also lists the nodes where the join was run as well as the files that were joined together.

CPI4343

Optimizer debug messages for distributed query step &1 of &2.

This message indicates that a distributed query was processed in multiple steps and lists the current step number. Following this message are all the optimizer messages for that step.

CPI4345

Temporary distributed result file &4 built for query.

This message indicates that a temporary distributed result file was created and lists a reason code as to why the temporary file was required. This message also shows the partitioning key that was used to create the file and the nodes that the temporary file was created on.

The following example shows you how to look at the distributed optimizer messages that were generated to determine how the distributed query was processed. The example uses the distributed files, EMPLOYEE and DEPARTMENT.

```
SQL:      SELECT A.EMPNO, B.MGRNO, C.MGRNO, D.EMPNO
          FROM   EMPLOYEE A, DEPARTMENT B, DEPARTMENT C, EMPLOYEE D
          WHERE  A.EMPNO=B.MGRNO
              AND B.ADMRDEPT=C.DEPTNO
              AND C.DEPTNO=D.WORKDEPT

OPNQRYF:  OPNQRYF FILE((EMPLOYEE) (DEPARTMENT) (DEPARTMENT) (EMPLOYEE))
          FORMAT(JFMT)
          JFLD((1/EMPNO 2/MGRNO *EQ)
              (2/ADMRDEPT 3/DEPTNO)
              (3/DEPTNO 4/WORKDEPT))
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

The following list of distributed optimizer messages is generated:

- CPI4343 Optimizer debug messages for distributed query step 1 of 4 follow:
 - CPI4345 Temporary distributed result file *QQTDF0001 built for query.
File B was directed into temporary file *QQTDF0001.
- CPI4343 Optimizer debug messages for distributed query step 2 of 4 follow:
 - CPI4342 Performing distributed join for query.
Files B, C and *QQTDF0001 were joined. This was a combination of a co-located join (between files B and C) and a directed join (with file *QQTDF0001).
 - CPI4345 Temporary distributed result file *QQTDF0002 built for query.
Temporary distributed file *QQTDF0002 was created to contain the result of joining files B, C and *QQTDF0001. This file was directed.
- CPI4343 Optimizer debug messages for distributed query step 3 of 4 follow:
 - CPI4345 Temporary distributed result file *QQTDF0003 built for query.
File A was directed into temporary file *QQTDF0003.
- CPI4343 Optimizer debug messages for distributed query step 4 of 4 follow:
 - CPI4342 Performing distributed join for query.
Files *QQTDF0002 and *QQTDF0003 were joined. This was a repartitioned join, because both files were directed before the join occurred.

Additional tools that you may want to use when tuning queries for performance include the CL commands Print SQL Information (PRTSQLINF), which applies to SQL programs and packages, and Change Query Attributes (CHGQRYA).

Changes to the change query attributes (CHGQRYA) command with DB2 Multisystem

The CHGQRYA command has two parameters that are applicable to distributed queries: ASYNCJ (asynchronous job usage) and APYRMT (apply remote):

- “Asynchronous job usage (ASYNCJ) parameter with DB2 Multisystem”
- “Apply remote (APYRMT) parameter” on page 53

Note: Unlike other parameters, ASYNCJ and APYRMT have no system values. If a value other than the default is necessary, the value must be changed for each job.

Asynchronous job usage (ASYNCJ) parameter with DB2 Multisystem

You can use the ASYNCJ parameter to control the usage of the temp writer. The ASYNCJ parameter has the following options:

- *ANY—allows the temp writer jobs to be used for database queries involving distributed files.
- *DIST—allows the temp writer jobs to be used for database queries involving distributed files.
- *LOCAL—allows the temp writer jobs to be used for queries of local files only. Although this option is allowed, currently there is no system support for using

temp writers for local query processing. *LOCAL was added to disable the temp writer for distributed queries, yet still allow communications to be performed asynchronously.

- *NONE—never use the temp writer. In addition, when distributed processing is performed, communications are performed synchronously. This can be very useful when analyzing queries, because it allows query debug messages from remote systems to be returned to the local system.

The following example shows you how to disable asynchronous job usage for distributed file processing:

```
CHGQRYA ASYNCJ(*LOCAL)
```

This command prevents asynchronous jobs from being used for queries involving distributed files.

The following example shows you how to completely disable asynchronous job usage:

```
CHGQRYA ASYNCJ(*NONE)
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

This command prevents asynchronous jobs from being used for any queries. In addition, for queries involving distributed files, communications to remote systems are done in a synchronous fashion.

The following example shows you how to use the CHGQRYA command in combination with the Start Debug (STRDBG) command to analyze a distributed query:

```
STRDBG UPDPDPROD(*YES)
CHGQRYA ASYNCJ(*NONE)
STRSQL
      SELECT COUNT(*) FROM EMPLOYEE A
```

The following debug messages are put into the job log:

```
Current connection is to relational database SYSA.
DDM job started.
Optimizer debug messages for distributed query step 1 of 2 follow:
Temporary distributed result file *QTD0001 built for query.
Following messages created on target system SYSB.
Arrival sequence access was used for file EMPLOYEE.
Arrival sequence access was used for file EMPLOYEE.
Optimizer debug messages for distributed query step 2 of 2 follow:
Arrival sequence access was used for file EMPLOYEE.
ODP created.
Blocking used for query.
```

See “Code Disclaimer Information” on page v for information pertaining to code examples.

Apply remote (APYRMT) parameter

You can use the APYRMT parameter to specify whether the other CHGQRYA options should be applied to the associated remote system jobs that are used in the processing of the distributed query requests:

- *YES—apply the CHGQRYA options to remote jobs. This requires that the remote job have authority to use the CHGQRYA command. Otherwise, an error is signalled to the remote job.
- *NO—apply the CHGQRYA options only locally.

The following example prevents the CHGQRYA options from being applied remotely:

```
CHGQRYA DEGREE(*NONE) APYRMT(*NO)
```

In this case, SMP parallel processing is prevented on the coordinator node, but the remote systems are allowed to choose their own parallel degree.

In addition to these parameters, you should be aware of how the parameter Query Time Limit (QRYTIMLMT) works. The time limit specified on the parameter is applied to each step (local and remote) of the distributed query; it is not applied to the entire query. Therefore, it is possible for one query step to encounter the time limit; whereas, another step may continue without problems. While the time limit option can be quite useful, it should be used with caution on distributed queries.

Summary of performance considerations

You should consider the following when developing queries that use distributed files:

1. For the OPNQRYF command and the query API (QQQQRY), specifying ALWCPYDTA(*OPTIMIZE) allows each node to choose an index or a sort to satisfy the ordering specified.
2. For the OPNQRYF command and the query API (QQQQRY), specifying ALWCPYDTA(*YES) or ALWCPYDTA(*NO) enforces that each node use an index that exactly matches the specified ordering fields. This is more restrictive than the way the optimizer processes ordering for nondistributed files.
3. Adding an ORDER BY clause to a DISTINCT select can return records faster by not requiring a final sort on the requesting system.
4. Including all of the fields of the partitioning key in the grouping fields generally results in one-step grouping, which performs better than two-step grouping.
5. Including all of the fields of the partitioning key in the join criteria generally results in a co-located distributed join.
6. Including all of the fields of the partitioning key in isolatable, equal record selection generally results in the query being processed on only one node.
7. Including any of the following scalar functions in isolatable, equal record selection generally results in the query being processed on only one node:
 - NODENAME
 - NODENUMBER
 - PARTITION


Bibliography


This bibliography lists the information referred to in this book:


- The Backup and Recovery topic provides information about setting up and managing:
 - Journaling, access path protection, and commitment control
 - User auxiliary storage pools (ASPs)
 - Disk protection (device parity, mirrored, and checksum)


Provides performance information about backup media and save/restore operations. Also includes advanced backup and recovery topics, such as using save-while-active support, saving and restoring to a different release, and programming tips and techniques.

- The Control Language topic in the Information Center provides a description of the iSeries server control language (CL) and its OS/400 commands. (Non-OS/400 commands are described in the respective licensed program publications.) Also provides an overview of *all* the CL commands for the iSeries server, and it describes the syntax rules needed to code them.

- The Database Programming  topic in the Information Center provides a detailed discussion of the iSeries server database organization, including information on how to create, describe, and update database files on the system. Also describes how to define files to the system using OS/400 data description specifications (DDS) keywords.

- The SQL Programming Concepts  topic in the Information Center provides information about how to use DB2 for AS/400 Query Manager and SQL Development Kit licensed program. Shows how to access data in a database library and to prepare, run, and test an application program that contains embedded SQL statements. Contains examples of SQL/400 statements and a description of the interactive SQL function. Describes common concepts and rules for using SQL/400 statements in COBOL/400, ILE COBOL/400, PL/I, ILE C/400, FORTRAN/400, RPG/400, ILE RPG/400, and REXX.

- The SQL Reference  topic in the Information Center provides information about how to use Structured Query Language/400 DB2/400 statements and gives details about the proper use of the statements. Examples of statements include syntax diagrams, parameters, and definitions. A list of SQL limits and a description of the SQL communication area (SQLCA) and SQL descriptor area (SQLDA) are also provided.

- The Distributed Database Programming  topic in the Information Center provides information on preparing and managing an iSeries server in a distributed relational database using the Distributed Relational Database Architecture (DRDA). It describes planning, setting up, programming, administering, and operating a distributed relational database on more than one server in a like-system environment.
- The *OptiConnect for OS/400*, SC41-5414-03 book describes OptiConnect support, which can connect multiple servers using a fiber optic cable. OptiConnect allows you to access intersystem databases more quickly and enables you to offload work to another server. Additional topics include configuration, installation, and operation information.
- The *Software Installation*, SC41-5120-06 book includes planning information and step-by-step instructions for the following procedures for installing the operating system and licensed programs:
 - Initial installation
 - Replacing the installed release with a new release
 - Adding additional licensed programs
 - Adding secondary languages
 - Changing the primary language of the system
- The OS/400 APIs topic in the Information Center provides information for the experienced programmer on how to use the application programming interfaces (APIs) to such OS/400 functions as:
 - Dynamic Screen Manager
 - Files (database, spooled, hierarchical)

- Journal and commitment control
- Message handling
- National language support
- Network management
- Objects
- Problem management
- Registration facility
- Security
- Software products
- Source debug
- UNIX**-type
- User-defined communications
- User interface
- Work management

Includes original program model (OPM), Integrated Language Environment (ILE), and UNIX-type APIs.

For information on DB2 UDB Symmetric Multiprocessing, see the SQL Programming Concepts topic and the Database Programming topic in the Information Center.

Index

Special characters

- (APYRMT) apply remote parameter 53
- (ASYN CJ) asynchronous job usage parameter 52
- *NODGRP system object 5

A

- access plan 47
- Add Logical File Member (ADDLFM) command 19
- Add Physical File Constraint (ADDPFCST) command 19
- Add Physical File Member (ADDPFM) command 19
- Add Physical File Trigger (ADDPFTRG) command 19
- Add RDB Directory Entries (ADDRDBDIRE) command 6
- adding systems to network redistribution issues 33
- ADDLFM (Add Logical File Member) command 19
- ADDPFCST (Add Physical File Constraint) command 19
- ADDPFM (Add Physical File Member) command 19
- ADDPFTRG (Add Physical File Trigger) command 19
- ADDRDBDIRE (Add RDB Directory Entries) command 6
- ALCOBJ (Allocate Object) command 19
- Allocate Object (ALCOBJ) command 19
- allow copy data (ALWCPYDTA) parameter 38
 - processing 39
- ALWCPYDTA (allow copy data) parameter 38
 - processing 39
- Apply Journalled Changes (APYJRNCHG) command
 - affects local file pieces only 18
- apply remote (APYRMT) parameter 53
- APYJRNCHG (Apply Journalled Changes) command
 - affects local file pieces only 18
- asynchronous job usage (ASYN CJ) parameter 52
- authority changes across node group inconsistencies 19
- authority errors 19

B

- benefits
 - DB2 Multisystem 2
- bibliography 55
- broadcast join 43
 - example 43

C

- Change Logical File (CHGLF) command 19
- Change Logical File Member (CHGLFM) command
 - SHARE parameter not allowed with distributed files 17
- Change Node Group Attributes (CHGNODGRPA) command 5, 10
- Change Object Owner (CHGOBJOWN) command 19
- Change Physical File (CHGPF) command 13, 19
 - data distribution, customizing 24
- Change Physical File Constraint (CHGPF CST) command 19
- Change Physical File Member (CHGPFM) command
 - SHARE parameter not allowed with distributed files 17
- Change Query Attributes (CHGQRYA) command
 - changes to 52
- CHGLF (Change Logical File) command 19
- CHGLFM (Change Logical File Member) command
 - SHARE parameter not allowed with distributed files 17
- CHGNODGRPA (Change Node Group Attributes) command 5, 10
- CHGOBJOWN (Change Object Owner) command 19
- CHGPF (Change Physical File) command 13, 19
 - data distribution, customizing 24
- CHGPF CST (Change Physical File Constraint) command 19
- CHGPFM (Change Physical File Member) command
 - SHARE parameter not allowed with distributed files 17
- Change Query Attributes (CHGQRYA) command
 - changes to 52
- CHGLF (Change Logical File) 19
- CHGLFM (Change Logical File Member)
 - SHARE parameter not allowed with distributed files 17
- CHGNODGRPA (Change Node Group Attributes) 5, 10
- CHGOBJOWN (Change Object Owner) 19
- CHGPF (Change Physical File) 13, 19
 - data distribution, customizing 24
- CHGPF CST (Change Physical File Constraint) 19
- CHGPFM (Change Physical File Member)
 - SHARE parameter not allowed with distributed files 17
- Change Query Attributes (CHGQRYA) changes to 52
- CHGLF (Change Logical File) 19
- CHGLFM (Change Logical File Member)
 - SHARE parameter not allowed with distributed files 17
- CHGNODGRPA (Change Node Group Attributes) 5, 10
- CHGOBJOWN (Change Object Owner) 19
- CHGPF (Change Physical File) 13, 19
 - data distribution, customizing 24
- CHGPF CST (Change Physical File Constraint) 19
- CHGPFM (Change Physical File Member)
 - SHARE parameter not allowed with distributed files 17
- CHGQRYA (Change Query Attributes) changes to 52
- Clear Physical File Member (CLRPFM) command 19
- CLRPFM (Clear Physical File Member) command 19
- co-located join 40
 - example 40
- command, CL
 - Add Logical File Member (ADDLFM) 19
 - Add Physical File Constraint (ADDPFCST) 19
 - Add Physical File Member (ADDPFM) 19
 - Add Physical File Trigger (ADDPFTRG) 19
- command, CL (*continued*)
 - Add RDB Directory Entries (ADDRDBDIRE) 6
 - ADDLFM (Add Logical File Member) 19
 - ADDPFCST (Add Physical File Constraint) 19
 - ADDPFM (Add Physical File Member) 19
 - ADDPFTRG (Add Physical File Trigger) 19
 - ADDRDBDIRE (Add RDB Directory Entries) 6
 - ALCOBJ (Allocate Object) 19
 - Allocate Object (ALCOBJ) 19
 - Apply Journalled Changes (APYJRNCHG)
 - affects local file pieces only 18
 - APYJRNCHG (Apply Journalled Changes)
 - affects local file pieces only 18
 - Change Logical File (CHGLF) 19
 - Change Logical File Member (CHGLFM)
 - SHARE parameter not allowed with distributed files 17
 - Change Node Group Attributes (CHGNODGRPA) 5, 10
 - Change Object Owner (CHGOBJOWN) 19
 - Change Physical File (CHGPF) 13, 19
 - data distribution, customizing 24
 - Change Physical File Constraint (CHGPF CST) 19
 - Change Physical File Member (CHGPFM)
 - SHARE parameter not allowed with distributed files 17
 - Change Query Attributes (CHGQRYA) changes to 52
 - CHGLF (Change Logical File) 19
 - CHGLFM (Change Logical File Member)
 - SHARE parameter not allowed with distributed files 17
 - CHGNODGRPA (Change Node Group Attributes) 5, 10
 - CHGOBJOWN (Change Object Owner) 19
 - CHGPF (Change Physical File) 13, 19
 - data distribution, customizing 24
 - CHGPF CST (Change Physical File Constraint) 19
 - CHGPFM (Change Physical File Member)
 - SHARE parameter not allowed with distributed files 17
 - CHGQRYA (Change Query Attributes) changes to 52
 - Clear Physical File Member (CLRPFM) 19

- command, CL (*continued*)
 - CLRPFM (Clear Physical File Member) 19
 - Copy (COPY) command
 - not allowed with distributed files 17
 - COPY (Copy) command
 - not allowed with distributed files 17
 - Copy File (CPYF) 19, 20
 - CPYF (Copy File) 19, 20
 - Create Duplicate Object (CRTDUPOBJ)
 - not allowed with distributed files 17
 - Create Logical File (CRTLFL) 19
 - Create Node Group (CRTNODGRP) 6
 - changing data partitioning 5
 - Create Physical File (CRTPF) 6, 13
 - CRTDUPOBJ (Create Duplicate Object)
 - not allowed with distributed files 17
 - CRTLFL (Create Logical File) 19
 - CRTNODGRP (Create Node Group) 6
 - changing data partitioning 5
 - CRTPF (Create Physical File) 6, 13
 - Deallocate Object (DLCOBJ) 19
 - Delete File (DLTF) 19
 - Delete Node Group (DLTNODGRP)
 - command 11
 - Display File Description (DSPFD) 18
 - Display Node Group (DSPNODGRP) 9
 - Display Object Description (DSPOBJD)
 - affects local file pieces only 18
 - Display Physical File Member (DSPPFM) 18
 - distributed file restrictions 17
 - DLCOBJ (Deallocate Object) 19
 - DLTF (Delete File) 19
 - DLTNODGRP (Delete Node Group)
 - command 11
 - DMPOBJ (Dump Object)
 - affects local file pieces only 18
 - DSPFD (Display File Description) 18
 - DSPNODGRP (Display Node Group) 9
 - DSPOBJD (Display Object Description)
 - affects local file pieces only 18
 - DSPPFM (Display Physical File Member) 18
 - Dump Object (DMPOBJ)
 - affects local file pieces only 18
 - End Journal Access Path (ENDJRNAP)
 - affects local file pieces only 18
 - End Journal Physical File Changes (ENDJRNPF) 19
 - ENDJRNAP (End Journal Access Path)
 - affects local file pieces only 18
 - ENDJRNPF (End Journal Physical File Changes) 19
 - Grant Object Authority (GRTOBJAUT) 19
 - GRTOBJAUT (Grant Object Authority) 19
- command, CL (*continued*)
 - Initialize Physical File Member (INZPFM)
 - not allowed with distributed files 17
 - INZPFM (Initialize Physical File Member)
 - not allowed with distributed files 17
 - list of
 - commands not allowed to run with distributed files 17
 - commands that affect all pieces of distributed files 19
 - commands that only affect local pieces of distributed files 18
 - Move Object (MOV OBJ)
 - not allowed with distributed files 17
 - MOV OBJ (Move Object)
 - not allowed with distributed files 17
 - Open Query File (OPNQRYF) 38
 - OPNQRYF (Open Query File) 38
 - POSDBF (Position Database File)
 - not allowed with distributed files 17
 - Position Database File (POSDBF)
 - not allowed with distributed files 17
 - Remove Journalized Changes (RMVJRNCHG)
 - affects local file pieces only 18
 - Remove Member (RMVM)
 - not allowed with distributed files 17
 - Remove Physical File Constraint (RMVPCST) 19
 - Remove Physical File Trigger (RMVPTFRG) 19
 - Rename Library (RNMLIB) command
 - not allowed with distributed files 17
 - Rename Member (RNMM)
 - not allowed with distributed files 17
 - Rename Object (RNMOBJ) 19
 - Reorganize Physical File Member (RGZPFM) 19
 - Restore Object (RSTOBJ)
 - affects local file pieces only 18
 - Revoke Object Authority (RVKOBJAUT) 19
 - RGZPFM (Reorganize Physical File Member) 19
 - RMVJRNCHG (Remove Journalized Changes)
 - affects local file pieces only 18
 - RMVM (Remove Member)
 - not allowed with distributed files 17
 - RMVPCST (Remove Physical File Constraint) 19
 - RMVPTFRG (Remove Physical File Trigger) 19
- command, CL (*continued*)
 - RNMLIB (Rename Library) command
 - not allowed with distributed files 17
 - RNMM (Rename Member)
 - not allowed with distributed files 17
 - RNMOBJ (Rename Object) 19
 - RSTOBJ (Restore Object)
 - affects local file pieces only 18
 - RVKOBJAUT (Revoke Object Authority) 19
 - Save Object (SAVOBJ)
 - affects local file pieces only 18
 - SAVOBJ (Save Object)
 - affects local file pieces only 18
 - Start Journal Access Path (STRJRNAP)
 - affects local file pieces only 18
 - Start Journal Physical File (STRJRNPF) 19
 - STRJRNAP (Start Journal Access Path)
 - affects local file pieces only 18
 - STRJRNPF (Start Journal Physical File) 19
 - Work with RDB Directory Entries (WRKRDBDIRE) command 6
 - WRKRDBDIRE (Work with RDB Directory Entries) command 6
 - commitment control 31
 - controlling
 - temp writer 50
 - conversation, protected 31
 - coordinator node
 - definition 28
 - Copy (COPY) command
 - not allowed with distributed files 17
 - COPY (Copy) command
 - not allowed with distributed files 17
 - Copy File (CPYF) command 19
 - using with distributed files 20
 - CPYF (Copy File) command 19
 - using with distributed files 20
 - Create Duplicate Object (CRTDUPOBJ)
 - command
 - not allowed with distributed files 17
 - Create Logical File (CRTLFL)
 - command 19
 - Create Node Group (CRTNODGRP)
 - command
 - changing data partitioning 5
 - using 6
 - Create Physical File (CRTPF)
 - command 6, 13
 - CREATE TABLE
 - node group example 13
 - CRTDUPOBJ (Create Duplicate Object)
 - command
 - not allowed with distributed files 17
 - CRTLFL (Create Logical File)
 - command 19
 - CRTNODGRP (Create Node Group)
 - command
 - changing data partitioning 5
 - using 6
 - CRTPF (Create Physical File)
 - command 6, 13

D

- data distribution
 - Change Physical File (CHGPF)
 - command 24
 - customizing 24
- data integrity 31
- data node 6
- data partitioning attribute, changing
 - example 10
- data types
 - that are partition compatible 39
- database file
 - creating as a distributed file 13
 - distributed across systems figure 1
- database system
 - using DB2 Multisystem to increase your system 33
- Database to Node Number Correlation figure 9
- DB2 Multisystem
 - benefits 2
 - installing 1
 - introduction to 1
 - journaling considerations 20
- Deallocate Object (DLCOBJ)
 - command 19
- default partitioning 5
- definition
 - coordinator node 28
 - distributed file 2
 - hashing 2
 - node 2
 - node group 2
 - node group object 2
 - partition compatible 39
 - partition map 2
 - partition number 2
 - partitioning 2
 - partitioning file 2
 - partitioning key 2
 - protected conversation 31
 - shared nothing cluster 1
 - temporary result file 36
 - visibility node 6
- Delete File (DLTF) command 19
- Delete Node Group (DLTNODGRP)
 - command
 - using 11
- directed join 41
 - example 41
- Display File Description (DSPFD)
 - command 18
- Display Node Group (DSPNODGRP)
 - command
 - using 9
- Display Node Group display
 - Database to Node Number Correlation figure 9
 - Partition Number to Node Number Correlation 9
- Display Object Description (DSPOBJD)
 - command
 - affects local file pieces only 18
- Display Physical File Member (DSPPFM)
 - command 18
- DISTINCT clause
 - implementation 39

- DISTINCT clause (*continued*)
 - optimization 39
- distributed file
 - creating 13
 - definition 2
 - displaying
 - local records 18
 - remote data 18
 - introduction 13, 16
 - restrictions 15
 - using Copy File (CPYF) command with 20
- distributed physical file
 - creating
 - example 13
 - using Create Physical File (CRTPF) command 13
 - using SQL CREATE TABLE statement 13
- distributed query
 - optimization overview 36
 - temporary result file 36
- distributed relational database network
 - defining local (*LOCAL) system 6
 - setting up 6
 - local (*LOCAL) system 6
- DLCOBJ (Deallocate Object)
 - command 19
- DLTF (Delete File) command 19
- DLTNODGRP (Delete Node Group)
 - command
 - using 11
- DMPOBJ (Dump Object) command
 - affects local file pieces only 18
- DSPFD (Display File Description)
 - command 18
- DSPNODGRP (Display Node Group)
 - command
 - using 9
- DSPOBJD (Display Object Description)
 - command
 - affects local file pieces only 18
- DSPPFM (Display Physical File Member)
 - command 18
- Dump Object (DMPOBJ) command
 - affects local file pieces only 18

E

- End Journal Access Path (ENDJRNAP)
 - command
 - affects local file pieces only 18
- End Journal Physical File Changes (ENDJRNPF) command 19
- ENDJRNAP (End Journal Access Path)
 - command
 - affects local file pieces only 18
- ENDJRNPF (End Journal Physical File Changes) command 19
- example
 - broadcast join 43
 - broadcast join query 43
 - co-located join 40
 - co-located join query 40
 - CREATE TABLE statement used to specify node group 13

- example (*continued*)
 - creating node group with specific partitioning 6
 - data partitioning attributes,
 - changing 10
 - directed join 41
 - directed join query 41
 - distributed physical file, creating 13
 - node group
 - creating 6
 - creating with default partitioning 6
 - deleting 11
 - displaying 9
 - performance improvement 31
 - repartitioned join 42
 - repartitioned join query 42
 - visibility node, using 6

F

- figure 9
 - Database to Node Number Correlation 9
 - distribution of database files 1
 - node group 2
 - open data path (ODP) 47
 - partition map 2
 - partitioning file 6
- file
 - partitioning
 - definition 2
- from record (FROMRCD) parameter 18
- FROMRCD (from record) parameter 18
- function
 - SQL
 - relative record numbering (RRN) 29
 - special registers 28

G

- Grant Object Authority (GRTOBJAUT)
 - command 19
- grouping
 - implementation 45
 - optimization 45
 - with joins 46
- grouping, one-step
 - optimization 45
- grouping, two-step
 - optimization 45
- GRTOBJAUT (Grant Object Authority)
 - command 19

H

- hash algorithm 5
- HASH scalar function
 - for Open Query File (OPNQRYF) CL command 26
 - for SQL 26
- hashing
 - definition 2

I

- Initialize Physical File Member (INZPFM) command
 - not allowed with distributed files 17
- introduction
 - node groups 5
- INZPFM (Initialize Physical File Member) command
 - not allowed with distributed files 17

J

- join
 - broadcast 43
 - example 43
 - co-located 40
 - example 40
 - directed 41
 - example 41
 - implementation 39
 - optimization 39, 44
 - repartitioned 42
 - example 42
 - with grouping 46
- join field
 - over partitioning keys 44
- journaling
 - considerations with DB2 Multisystem 20

K

- key, partitioning
 - definition 2

L

- local (*LOCAL) system
 - defining in distributed relational database network 6
- local record
 - displaying 18

M

- map, partition
 - definition 2
 - figure 2
- merge
 - record ordering optimization 38
- message
 - optimizer 50
- Move Object (MOV OBJ) command
 - not allowed with distributed files 17
- MOV OBJ (Move Object) command
 - not allowed with distributed files 17

N

- network, adding systems to 33
- node
 - data 6
 - definition 2
 - displaying 9

- node (*continued*)
 - visibility
 - definition 6
- node group
 - *NODGRP system object 5
 - authority change inconsistencies 19
 - changing 5
 - data partitioning attributes
 - examples 10
 - creating 6
 - examples 6
 - with specific partitioning 6
 - default partitioning 5
 - definition 2
 - deleting 11
 - example 11
 - displaying 9
 - example 9
 - information about 18
 - figure 2
 - introduction to 5
- node group (NODGRP) parameter 6, 13
- node group commands
 - tasks to complete before using 6
- node group object
 - definition 2
- NODENAME scalar function
 - for Open Query File (OPNQRYF) CL command 27
 - for SQL 27
- NODENUMBER scalar function
 - for Open Query File (OPNQRYF) CL command 28
 - for SQL 28
- NODGRP (node group) parameter 6, 13
- null-capable fields
 - in partitioning key 22
- number
 - partition
 - definition 2

O

- ODP (open data path)
 - distributed 47
 - figure 47
 - local 47
 - reusable 47
- one-step grouping 45
- open data path (ODP)
 - figure 47
 - reusable 47
- Open Query File (OPNQRYF)
 - command 38
 - scalar functions 25
- OPNQRYF (Open Query File) command 38
 - scalar functions 25
- OptiConnect
 - connectivity method 2
- optimization
 - DISTINCT clause 39
 - grouping 45
 - join 39, 44
 - of distributed queries 36
 - one-step grouping 45
 - overview 36

- optimization (*continued*)
 - record ordering 38
 - merge 38
 - sort 38
 - single file query 36
 - two-step grouping 45
 - UNION clause 39
- optimizer
 - messages 50
- ordering, record
 - optimization 38
 - merge 38
 - sort 38
- overview
 - optimization 36

P

- partition compatible
 - data types 39
 - definition 39
- partition map
 - definition 2
 - figure 2
- partition number 5
 - definition 2
- Partition Number to Node Number Correlation figure 9
- PARTITION scalar function
 - for Open Query File (OPNQRYF) CL command 25
 - for SQL 25
- partitioning 5
 - advantages of SMP 22
 - considerations for setting up 22
 - default partitioning 5
 - definition 2
 - keys 22
 - map 22
 - number 22
 - planning 22
- partitioning attributes, changing
 - example 10
- partitioning file
 - definition 2
 - file definition 6
- partitioning file (PTNFILE) parameter 5, 6
- partitioning file example
 - figure 6
- partitioning key
 - choosing 23
 - definition 2
 - null-capable fields 22
 - over join fields 44
- partitioning key (PTNKEY) parameter 13
- performance 31
 - improvement 31
 - example 31
 - improving query processing 31
 - query 35
 - summary for queries 54
 - Symmetric Multiprocessing (SMP) enhancements 31
- plan, access 47

POSDBF (Position Database File)
 command
 not allowed with distributed files 17
 Position Database File (POSDBF)
 command
 not allowed with distributed files 17
 processing
 allow copy data (ALWCPYDTA)
 parameter 39
 ALWCPYDTA (allow copy data)
 parameter 39
 distribute data (DSTDTA)
 parameter 39
 DSTDTA (distribute data)
 parameter 39
 protected conversation
 definition 31
 with commitment control 31
 PTNFILE (partitioning file) parameter 5,
 6
 PTNKEY (partitioning key)
 parameter 13

Q

query
 broadcast join 43
 example 43
 co-located join 40
 example 40
 directed join 41
 example 41
 join implementation 39
 join optimization 39
 optimization overview 36
 performance 35
 performance consideration
 summary 54
 performance improvements 31
 repartitioned join 42
 example 42
 temporary result file 36
 query optimizer 31
 query, single file
 implementation 36
 optimization 36

R

record ordering
 implementation 38
 merge optimization 38
 optimization 38
 sort optimization 38
 redistribution issues
 when adding systems to network 33
 related printed information 55
 relative record numbering (RRN)
 function 29
 remote data
 displaying 18
 remote location name (RMTLOCNAME)
 parameter 6
 Remove Journalized Changes
 (RMVJRNCHG) command
 affects local file pieces only 18

Remove Member (RMVM) command
 not allowed with distributed files 17
 Remove Physical File Constraint
 (RMVPF CST) command 19
 Remove Physical File Trigger
 (RMVPFTRG) command 19
 Rename Library (RNMLIB) command
 not allowed with distributed files 17
 Rename Member (RNMM) command
 not allowed with distributed files 17
 Rename Object (RNMOBJ) command 19
 Reorganize Physical File Member
 (RGZPFM) command 19
 repartitioned join 42
 example 42
 Restore Object (RSTOBJ) command
 affects local file pieces only 18
 restrictions
 CL commands not allowed to run
 with distributed files 17
 CL commands that affect all pieces of
 distributed files 19
 CL commands that only affect local
 pieces of distributed files 18
 distributed files 15
 reusable open data path (ODP) 47
 Revoke Object Authority (RVKOBJAUT)
 command 19
 RGZPFM (Reorganize Physical File
 Member) command 19
 RMTLOCNAME (remote location name)
 parameter 6
 RMVJRNCHG (Remove Journalized
 Changes) command
 affects local file pieces only 18
 RMVM (Remove Member) command
 not allowed with distributed files 17
 RMVPF CST (Remove Physical File
 Constraint) command 19
 RMVPFTRG (Remove Physical File
 Trigger) command 19
 RNMLIB (Rename Library) command
 not allowed with distributed files 17
 RNMM (Rename Member) command
 not allowed with distributed files 17
 RNMOBJ (Rename Object) command 19
 RRN (relative record numbering)
 function 29
 RSTOBJ (Restore Object) command
 affects local file pieces only 18
 RVKOBJAUT (Revoke Object Authority)
 command 19

S

Save Object (SAVOBJ) command
 affects local file pieces only 18
 SAVOBJ (Save Object) command
 affects local file pieces only 18
 scalability 31
 scalar function 25
 Open Query File (OPNQRYF) CL
 command
 HASH 26
 NODENAME 27
 NODENUMBER 28
 PARTITION 25

scalar function (*continued*)
 SQL
 HASH 26
 NODENAME 27
 NODENUMBER 28
 PARTITION 25
 shared nothing cluster 1
 definition 1
 single file query
 implementation 36
 optimization 36
 SMP (Symmetric Multiprocessing) 31
 special register function
 for SQL 28
 SQL
 scalar functions 25
 Start Journal Access Path (STRJRNAP)
 command
 affects local file pieces only 18
 Start Journal Physical File (STRJRNPF)
 command 19
 Start SQL (STRSQL) command 38
 STRJRNAP (Start Journal Access Path)
 command
 affects local file pieces only 18
 STRJRNPF (Start Journal Physical File)
 command 19
 STRSQL (Start SQL) command 38
 subquery
 description 47
 Symmetric Multiprocessing (SMP) 31
 system object
 *NODGRP 5
 systems, adding to network
 redistribution issues 33

T

temp writer 48
 advantages of using 49
 controlling 50
 disadvantages of using 50
 temporary result file
 definition 36
 temporary result writer 48
 two-phase commit protocols 31
 two-step grouping 45

U

UNION clause
 implementation 39
 optimization 39

V

visibility node
 definition 6
 example of how to use 6
 how to create 6

W

Work with RDB Directory Entries
 (WRKRDBDIRE) command 6

writer, temp 48
writer, temporary result 48
WRKRDBDIRE (Work with RDB
Directory Entries) command 6



Printed in U.S.A.