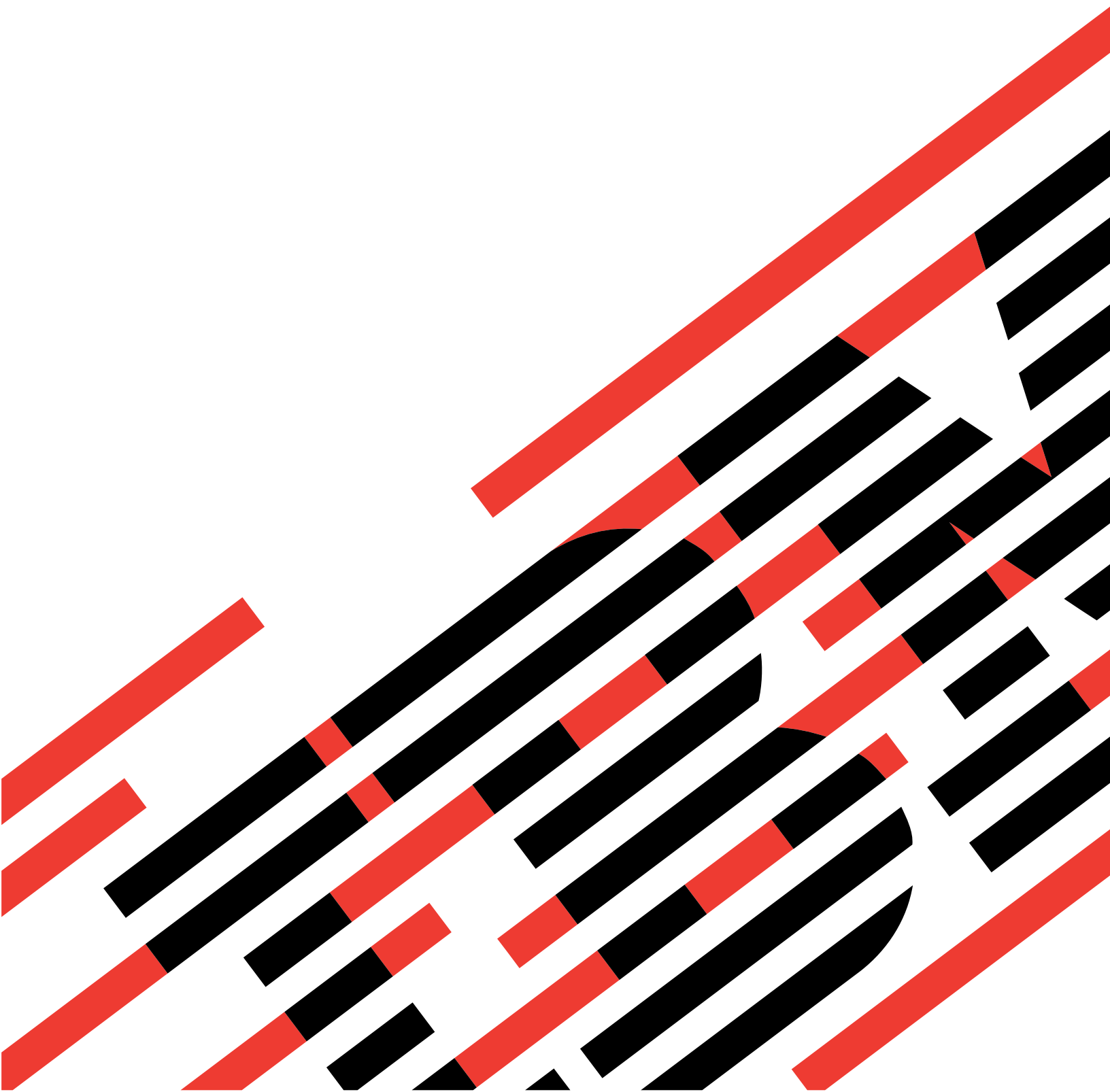




@server

iSeries

DB2 Universal Database for iSeries -
Database Performance and Query Optimization





@server

iSeries

DB2 Universal Database for iSeries -
Database Performance and Query Optimization

Contents

About DB2 UDB for iSeries Database Performance and Query Optimization	vii
Who should read the Database Performance and Query Optimization book	vii
Assumptions relating to SQL statement examples	viii
How to interpret syntax diagrams	viii
What's new for V5R2	ix
I Code disclaimer information	ix
Chapter 1. Database performance and query optimization: Overview	1
Creating queries	2
Chapter 2. Data access on DB2 UDB for iSeries: data access paths and methods	3
Table scan	3
Index	3
Encoded vector index	3
Data access: data access methods	3
Data access methods: Summary	5
Ordering query results	7
Enabling parallel processing for queries	7
Spreading data automatically	8
Table scan access method	8
Parallel table prefetch access method	10
Parallel table scan method	11
Index scan-key selection access method	13
Parallel index scan-key selection access method (available only when the DB2 UDB Symmetric Multiprocessing feature is installed)	14
Index scan-key positioning access method	15
Parallel index scan-key positioning access method (available only when the DB2 UDB Symmetric Multiprocessing feature is installed)	19
Index Only Access Method	21
Parallel table or index based preload access method	22
Index-from-index access method	22
Hashing access method	23
Bitmap processing method	24
Sort access method	28
Chapter 3. The DB2 UDB for iSeries query optimizer: Overview	31
How the query optimizer makes your queries more efficient	31
Optimizer decision-making rules	31
Cost estimation for queries	32
General query optimization tips	34
Access plan validation	34
Join optimization	35
Grouping optimization	50
Ordering optimization	54
View implementation	55
Chapter 4. Optimizing query performance using query optimization tools	59
Verify the performance of SQL applications	60
Examine query optimizer debug messages in the job log	60
Query optimization performance information messages	61
Query optimization performance information messages and open data paths	66
Gather information about embedded SQL statements with the PRTSQLINF command	67
Gather statistics about your queries with the database monitor	69

Start Database Monitor (STRDBMON) command	70
End Database Monitor (ENDDBMON) command	71
Database monitor performance rows	72
Query optimizer index advisor	72
Database monitor examples	73
Gather statistics about your queries with memory-resident database monitor APIs.	78
Memory-resident database monitor external API description	79
Memory-resident database monitor external table description	79
Sample SQL queries	80
Memory-resident database monitor row identification	80
I Monitoring your database performance using SQL Performance monitors in iSeries Navigator	80
I Creating an SQL performance monitor.	81
I Saving SQL performance monitor data (pausing a monitor)	82
I Analyzing SQL performance monitor data	82
View the effectiveness of your queries with Visual Explain	82
Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command	83
Control queries dynamically with the query options file QAQQINI	84
Specifying the QAQQINI file	85
Creating the QAQQINI query options file	85
Control long-running queries with the DB2 UDB for iSeries Predictive Query Governor	93
How the query governor works	94
Cancelling a query	95
Query governor implementation considerations	95
Query governor considerations for user applications: Setting the time limit	95
Controlling the default reply to the query governor inquiry message	95
Testing performance with the query governor	96
Examples of setting query time limits	96
Control parallel processing for queries	97
Controlling system wide parallel processing for queries	97
Controlling job level parallel processing for queries	98
I Analyzing queries with the Statistics Manager	100
I Statistics Manager APIs.	100
I Managing statistical information with iSeries Navigator	100
Query optimization tools: Comparison table	101
Chapter 5. Using indexes to speed access to large tables.	103
Coding for effective indexes: Avoid numeric conversions.	103
Coding for effective indexes: Avoid arithmetic expressions	104
Coding for effective indexes: Avoid character string padding	104
Coding for effective indexes: Avoid the use of like patterns beginning with % or _	104
Coding for effective indexes: Be aware of the instances where DB2 UDB for iSeries does not use an index.	105
Coding for effective indexes: Using indexes with sort sequence	106
Coding for effective indexes: Using indexes and sort sequence with selection, joins, or grouping	106
Coding for effective indexes: Ordering	106
Examples of indexes.	107
Index example: Equals selection with no sort sequence table	107
Index example: Equals selection with a unique-weight sort sequence table	108
Index example: Equal selection with a shared-weight sort sequence table	108
Index example: Greater than selection with a unique-weight sort sequence table.	108
Index example: Join selection with a unique-weight sort sequence table	108
Index example: Join selection with a shared-weight sort sequence table	109
Index example: Ordering with no sort sequence table.	109
Index example: Ordering with a unique-weight sort sequence table.	109
Index example: Ordering with a shared-weight sort sequence table.	110
Index example: Ordering with ALWCOPYDTA(*OPTIMIZE) and a unique-weight sort sequence table	110

Index example: Grouping with no sort sequence table	110
Index example: Grouping with a unique-weight sort sequence table	110
Index example: Grouping with a shared-weight sort sequence table	111
Index example: Ordering and grouping on the same columns with a unique-weight sort sequence table	111
Index example: Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table	111
Index example: Ordering and grouping on the same columns with a shared-weight sort sequence table	112
Index example: Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table	112
Index example: Ordering and grouping on different columns with a unique-weight sort sequence table	112
Index example: Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table	113
Index example: Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table	113
What are encoded vector indexes?	113
Chapter 6. Application design tips for database performance	117
Database application design tips: Use live data	117
Database application design tips: Reduce the number of open operations	118
Database application design tips: Retain cursor positions	120
Database application design tips: Retaining cursor positions for non-ILE program calls	120
Database application design tips: Retaining cursor positions across ILE program calls.	121
Database application design tips: General rules for retaining cursor positions for all program calls	121
Chapter 7. Programming techniques for database performance	123
Programming techniques for database performance: Use the OPTIMIZE clause	123
Programming techniques for database performance: Use FETCH FOR n ROWS.	124
Programming techniques for database performance: Improve SQL blocking performance when using FETCH FOR n ROWS	125
Programming techniques for database performance: Use INSERT n ROWS	125
Programming techniques for database performance: Control database manager blocking	125
Programming techniques for database performance: Optimize the number of columns that are selected with SELECT statements	126
Programming techniques for database performance: Eliminate redundant validation with SQL PREPARE statements	127
Programming techniques for database performance: Page interactively displayed data with REFRESH(*FORWARD)	127
Chapter 8. General DB2 UDB for iSeries performance considerations	129
Effects on database performance when using long object names	129
Effects of precompile options on database performance	129
Effects of the ALWCPYDTA parameter on database performance	130
Tips for using VARCHAR and VARGRAPHIC data types in databases	131
Appendix A. Database monitor: DDS	133
Database monitor physical file DDS	133
Optional database monitor logical file DDS.	140
Database monitor logical table 1000 - Summary Row for SQL Information	141
Database monitor logical table 3000 - Summary Row for Table Scan	152
Database monitor logical table 3001 - Summary Row for Index Used	157
Database monitor logical table 3002 - Summary Row for Index Created	163
Database monitor logical table 3003 - Summary Row for Query Sort	170
Database monitor logical table 3004 - Summary Row for Temp Table	174

Database monitor logical table 3005 - Summary Row for Table Locked	179
Database monitor logical table 3006 - Summary Row for Access Plan Rebuilt	182
Database monitor logical table 3007 - Summary Row for Optimizer Timed Out	185
Database monitor logical table 3008 - Summary Row for Subquery Processing	188
Database monitor logical table 3010 - Summary for HostVar & ODP Implementation	189
Database monitor logical table 3014 - Summary Row for Generic QQ Information	190
I Database monitor logical table 3015 - Summary Row for Statistics Information	197
Database monitor logical table 3018 - Summary Row for STRDBMON/ENDDDBMON	200
Database monitor logical table 3019 - Detail Row for Rows Retrieved	201
Database monitor logical table 3021 - Summary Row for Bitmap Created	202
Database monitor logical table 3022 - Summary Row for Bitmap Merge	205
Database monitor logical table 3023 - Summary for Temp Hash Table Created	208
Database monitor logical table 3025 - Summary Row for Distinct Processing	212
Database monitor logical table 3027 - Summary Row for Subquery Merge	213
Database monitor logical table 3028 - Summary Row for Grouping	217
Appendix B. Memory Resident Database Monitor: DDS	223
I External table description (QAQQRYI) - Summary Row for SQL Information	223
I External table description (QAQQTEXT) - Summary Row for SQL Statement	229
I External table description (QAQQ3000) - Summary Row for Arrival Sequence	229
I External table description (QAQQ3001) - Summary row for Using Existing Index	231
I External table description (QAQQ3002) - Summary Row for Index Created	233
I External table description (QAQQ3003) - Summary Row for Query Sort	235
I External table description (QAQQ3004) - Summary Row for Temporary Table	236
I External table description (QAQQ3007) - Summary Row for Optimizer Information	238
I External table description (QAQQ3008) - Summary Row for Subquery Processing	239
I External table description (QAQQ3010) - Summary Row for Host Variable and ODP Implementation	239
Index	241

About DB2 UDB for iSeries Database Performance and Query Optimization

This book explains to programmers and database administrators:

- How to use the tools and functions that are available in DB2 UDB for iSeries for getting the best performance out of your database applications
- How to run queries that make full use of the capabilities of the DB2 UDB for iSeries integrated database.

For more information on DB2 UDB for iSeries guidelines and examples for implementation in an application programming environment, see the following information in the Database and Files Systems category of the iSeries Information Center:

- SQL Reference
- SQL Programming Concepts
- SQL Programming with Host Languages
- SQL Call Level Interfaces (ODBC)
- Database Programming
- Query/400 Use
- ODBC
- SQLJ

Java Database Connectivity (JDBC) information can be found in the IBM® Developer Kit for Java™ under Programming in the iSeries™ Information Center.

For additional information on advanced database functions, see the *DATABASE 2/400 Advanced Database Functions* book, GG24-4249.

Who should read the Database Performance and Query Optimization book

This information is for programmers and database administrators who understand basic database applications and want to understand how to tune queries. This information shows how to use the available tools for debugging query performance problems.

You should be familiar with languages and interfaces, including the following:

- COBOL for iSeries
- ILE COBOL for iSeries
- iSeries PL/I
- ILE C for iSeries
- ILE C++
- VisualAge® C++ for iSeries
- REXX
- RPG III (part of RPG for iSeries)
- ILE RPG for iSeries
- Query/400
- The OPNQRYP command
- Call level interfaces (CLI)
- ODBC

- JDBC

Assumptions relating to SQL statement examples

The examples of queries that are shown in this book are based on the sample tables in Appendix A, "DB2 UDB for iSeries Sample Tables," of the SQL Programming Concepts book. For the SQL examples, assume the following:

- They are shown in the interactive SQL environment or they are written in ILE C or in COBOL. EXEC SQL and END-EXEC are used to delimit an SQL statement in a COBOL program. A description of how to use SQL statements in a COBOL program is provided in "Coding SQL Statements in COBOL Applications" of the SQL Programming with Host Languages book. A description of how to use SQL statements in an ILE C program is provided in "Coding SQL Statements in C Applications" of the SQL Programming with Host Languages book.
- Each SQL example is shown on several lines, with each clause of the statement on a separate line.
- SQL keywords are highlighted.
- Table names provided in Appendix A, "DB2 UDB for iSeries Sample Tables" of the SQL Programming Concepts use the collection CORPDATA. Table names that are not found in Appendix A, "DB2 UDB for iSeries Sample Tables," should use collections you create.
- Calculated columns are enclosed in parentheses, (), and brackets, [].
- The SQL naming convention is used.
- The APOST and APOSTSQL precompiler options are assumed although they are not the default options in COBOL. Character string constants within SQL and host language statements are delimited by apostrophes (').
- A sort sequence of *HEX is used, unless otherwise noted.
- The complete syntax of the SQL statement is usually not shown in any one example. For the complete description and syntax of any of the statements described in this guide, see the SQL Reference.

Whenever the examples vary from these assumptions, it is stated.

Because this guide is for the application programmer, most of the examples are shown as if they were written in an application program. However, many examples can be slightly changed and run interactively by using interactive SQL. The syntax of an SQL statement, when using interactive SQL, differs slightly from the format of the same statement when it is embedded in a program.

How to interpret syntax diagrams

Throughout this book, syntax is described using the structure defined as follows:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The **▶▶**— symbol indicates the beginning of a statement.

The **—▶** symbol indicates that the statement syntax is continued on the next line.

The **▶—** symbol indicates that a statement is continued from the previous line.

The **—▶▶** symbol indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the **▶—** symbol and end with the **—▶** symbol.

- Required items appear on the horizontal line (the main path).

▶▶—required_item—▶▶

- Optional items appear below the main path.

▶▶—required_item—optional_item—▶▶

If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.



- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.



If choosing one of the items is optional, the entire stack appears below the main path.



If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.



If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

What's new for V5R2

- | The major new features covered in this book include:
 - | • Arrival sequence joins
 - | • Redesigned the query engine
 - | • Statistics manager

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. Database performance and query optimization: Overview

The goal of database performance tuning is to minimize the response time of your queries and to make the best use of your server's resources by minimizing network traffic, disk I/O, and CPU time. This goal can only be achieved by understanding the logical and physical structure of your data, understanding the applications used on your server, and understanding how the many conflicting uses of your database may impact database performance.

The best way to avoid performance problems is to ensure that performance issues are part of your ongoing development activities. Many of the most significant performance improvements are realized through careful design at the beginning of the database development cycle. To most effectively optimize performance, you must identify the areas that will yield the largest performance increases over the widest variety of situations and focus your analysis on those areas.

- | In V5R2, DB2 UDB for iSeries redesigned the query engine, which may provide performance improvement
- | for many SQL read-only queries. A redbook will be published in late 2002 that will provide details on the
- | performance improvements, which types of queries can take advantage of the redesign, and how to aid
- | the optimizer in taking advantage of the new improvements.

Understanding access paths and the query optimizer

Since iSeries automatically manages many hardware resources, and uses a cost-based optimization formula to determine the most efficient access plan for running an SQL statement, it is important to know how the server determines the most efficient access method and what factors determine their selection by the server. These topics are covered in Data access methods. In addition, a clear understanding of the iSeries query optimizer will also help you design queries that leverage the query optimizer's cost estimation and decision-making rules.

Improving your queries

Once you are familiar with these concepts, you can incrementally improve your queries by reviewing the material found in the following topics:

Topic	Description
Optimizing query performance using query optimization tools	Describes how you can use query optimization tools to improve data retrieval times by gathering statistics about your queries or controlling the processing of your queries. With the results that these tools provide, you can then change the data access method chosen by the server or create the correct indexes and use them effectively.
Using indexes to speed access to large tables	Describes the index-based retrieval method for accessing tables and how to create effective indexes by avoiding such things as numeric conversions, arithmetic expressions, character string padding, and the use of like patterns.
Increasing database performance through application design	Describes how the correct design of user applications can improve performance. Application design considerations include parameter passing techniques, using live data, reducing the number of open operations, and retaining cursor positions.
Improving database performance using programming techniques	Describes how the correct programming techniques can improve performance. Among the techniques covered are: using the OPTIMIZE clause, using FETCH n ROWS, using INSERT n ROWS, controlling the database manager blocking, optimizing the number of columns selected with SELECT statements, eliminating redundant validation, and paging interactively displayed data.
General iSeries performance	Describes some general server considerations and how they affect the performance of your queries.

Creating queries

You can create queries through any of the following interfaces:

- SQL
- Open Query File (OPNQRYF) command
- Open database connectivity (ODBC)
- Query for iSeries
- Java Database Connectivity (JDBC)
- Call level interface (CLI)

The query optimizer optimizes all of the queries that you create using these interfaces.

Chapter 2. Data access on DB2 UDB for iSeries: data access paths and methods

This section introduces the data access methods that DB2 Universal Database for iSeries and the Licensed Internal Code use to process queries and access data. The data access methods are grouped into nonkeyed, keyed, and temporary result file access methods.

The iSeries fundamentally uses two methods to retrieve data specified in a query; through an index (keyed access methods) or directly through the table (nonkeyed access methods). These access methods can be combined in many ways to retrieve data. A data access method can be a table scan, an index, a combination of both, or an encoded vector index.

Table scan

A table scan, or arrival sequence, uses the order of rows as they are stored in the table to locate data that is specified in a query. Processing tables using the table scan is similar to processing sequential or direct files on traditional systems.

Index

An index, or keyed sequence access path, provides access to a table that is arranged according to the contents of key columns. The keyed sequence is the order in which rows are retrieved. The access path is automatically maintained whenever rows are added to or deleted from the table, or whenever the contents of the index columns are changed. The best examples of an index is an index that is created with the CREATE INDEX statement, or a keyed logical file that is created with the CRTLF command.


Columns that are good candidates for indexes are:

- Columns that are frequently referenced in row selection predicates.
- Columns that are frequently referenced in grouping or ordering.
- Columns that are used to join tables (see “Join optimization” on page 35).

Encoded vector index

An encoded vector index provides access to a database table by assigning codes to distinct key values and then representing these values in an array. The elements of the array can be 1, 2, or 4 bytes in length, depending on the number of distinct values that must be represented. Because of their compact size and relative simplicity, encoded vector indexes provide for faster scans that can be more easily processed in parallel.

You create encoded vector indexes by using the CREATE ENCODED VECTOR INDEX statement. See What are encoded vector indexes for information on the use and maintenance of encoded vector indexes.

For additional information about accelerating your queries with encoded vector indexes , go to the DB2 Universal Database for iSeries web pages.

Data access: data access methods

The Licensed Internal Code and DB2 Universal Database for iSeries share the work on access methods. The Licensed Internal Code does the low-level processing which includes selection, join functions, hashing, and index creation.

The query optimization process chooses the most efficient access method for each query and keeps this information in the access plan. The type of access is dependent on the number of rows, the expected number of page faults ¹, and other criteria.

You can use the tools and tips that are described later in this book to influence the way in which the query optimizer implements your queries.

The optimizer uses any of the following methods to retrieve data. See “Data access methods: Summary” on page 5 for a summary of these methods:

- Table scan method (a dataspace is an internal object that contains the data in a table)
- Parallel table prefetch method
- Index scan-key selection method
- Index scan-key positioning method
- Parallel table or index preload
- Index-from-index method
- Index only access method
- Hashing method
- Bitmap processing method
- Sort access method

Accessing data with the DB2 UDB Symmetric Multiprocessing methods

The DB2 UDB Symmetric Multiprocessing feature provides the optimizer with additional methods for retrieving data that include parallel processing.

Symmetrical multiprocessing (SMP) is a form of parallelism achieved on a single server where multiple processors (CPU and I/O processors) that share memory and disk resource work simultaneously towards achieving a single end result. This parallel processing means that the database manager can have more than one (or all) of the server processors working on a single query simultaneously. The performance of a CPU bound query can be significantly improved with this feature on multiple-processor servers by distributing the processor load across more than one processor on the server.

The following methods are available to the optimizer once the DB2 UDB Symmetric Multiprocessing feature has been installed on your server:

- Parallel table scan method
- Parallel index scan-key selection method
- Parallel index scan-key positioning method
- Parallel index only access method
- Parallel hashing method
- Parallel bitmap processing method

Additional considerations:

The following topics provide additional background information on the access methods:

- “Ordering query results” on page 7
- “Enabling parallel processing for queries” on page 7
- “Spreading data automatically” on page 8

1. An interrupt that occurs when a program refers to a 4K-byte page that is not in main storage.

Data access methods: Summary

The following table provides a summary of data management methods that are discussed in this book.

Table 1. Summary of data access methods

Access Method	Selection Process	Good When	Not Good When	Selected When	Advantages
"Table scan access method" on page 8	Reads all rows. Selection criteria applied to data in dataspace.	Approx. > 20% rows selected.	Approx. < 20% rows selected.	No ordering, grouping, or joining and approx. > 20% rows selected.	Minimizes page I/O through pre-fetching.
"Parallel table prefetch access method" on page 10	Data retrieved from auxiliary storage in parallel streams. Reads all rows. Selection criteria applied to data in dataspace.	Approx. > 20% rows selected. 1. Adequate active memory available. 2. Query would otherwise be I/O bound. 3. Data spread across multiple disk units.	Approx. < 20% rows selected. Query is CPU bound.	No ordering, grouping, or joining and approx. > 20% rows selected, and the server job has been configured to take advantage of I/O parallelism.	Minimizes wait time for page I/O through parallel table prefetching.
"Parallel table scan method" on page 11	Data read and selected in parallel tasks.	Approx. > 10% rows selected, large table. 1. Adequate active memory available. 2. Data spread across multiple disk units. 3. DB2 UDB Symmetric Multiprocessing installed. 4. Multi-processor server.	Approx. < 10% rows selected. Query is I/O bound on a uniprocessor server.	1. DB2 UDB Symmetric Multiprocessing installed. 2. CPU bound or running on a multiprocessor server.	Significant performance especially on multiprocessors.
"Index scan-key selection access method" on page 13	Selection criteria applied to index.	Ordering, grouping, and joining.	Large number of rows selected.	Index is required and cannot use index scan-key positioning method.	Dataspace accessed only for rows matching index scan-key selection criteria.
"Parallel index scan-key selection access method (available only when the DB2 UDB Symmetric Multiprocessing feature is installed)" on page 14	Selection criteria applied to index in parallel tasks.	Size of index is much less than the dataspace. DB2 UDB Symmetric Multiprocessing must be installed.	Large number of rows selected.	When ordering of results not required.	Better I/O overlap because parallel tasks perform the I/O. Can fully utilize multiprocessor servers.

Table 1. Summary of data access methods (continued)

Access Method	Selection Process	Good When	Not Good When	Selected When	Advantages
"Index scan-key positioning access method" on page 15	Selection criteria applied to range of index entries. Commonly used option.	Approx. < 20% rows selected.	Approx. > 20% rows selected.	Selection columns match left-most keys and approx. < 20% rows selected.	Index and dataspace accessed only for rows matching selection criteria.
"Parallel index scan-key positioning access method (available only when the DB2 UDB Symmetric Multiprocessing feature is installed)" on page 19	Selection criteria applied to range of index entries in parallel tasks.	Approx. < 20% rows selected. DB2 UDB Symmetric Multiprocessing must be installed.	Large number of rows selected.	<ol style="list-style-type: none"> 1. When ordering of results not required. 2. Selection columns match left-most keys and approx. < 20% rows selected. 	<ol style="list-style-type: none"> 1. Index and dataspace accessed only for rows matching selection criteria. 2. Better I/O overlap because parallel tasks perform the I/O. 3. Can fully utilize a multiprocessor servers.
"Index-from-index access method" on page 22	Key row positioning on permanent index. Builds temporary index over selected index entries.	Ordering, grouping and joining.	Approx. > 20% rows selected.	No existing index to satisfy ordering but existing index does satisfy selection and selecting approx. < 20% rows.	Index and dataspace accessed only for rows matching selection criteria.
"Sort access method" on page 28	Order data read using table scan processing or index scan-key positioning.	Approx. > 20% rows selected or large result set of rows.	Approx. < 20% rows selected or small result set of rows.	Ordering specified; either no index exists to satisfy the ordering or a large result set is expected.	See table scan and index scan-key positioning in this table.
"Index Only Access Method" on page 21	Done in combination with any of the other index access methods	All columns used in the query exist as key columns.	Approx. < 20% rows selected or small result set of rows.	All columns used in the query exist as key columns.	Reduced I/O to the dataspace.
"Parallel table or index based preload access method" on page 22	Index or table data loaded in parallel to avoid random access.	Excessive random activity would otherwise occur against the object and active memory is available to hold the entire object.	Active memory is already overcommitted.	Excessive random activity would result from processing the query and active memory is available which can hold the entire object.	Random page I/O is avoided which can improve I/O bound queries.

Table 1. Summary of data access methods (continued)

Access Method	Selection Process	Good When	Not Good When	Selected When	Advantages
"Hashing access method" on page 23(Parallel or non-parallel)	Rows with common correlated data having a common value.	Longer running grouping and join queries.	Short running queries.	Join or grouping specified.	Reduces random I/O when compared to index methods. If DB2 UDB Symmetric Multiprocessing is installed, possible exploitation of SMP parallelism.
"Bitmap processing method" on page 24	Key position/index scan-key selection used to build bitmap. Bitmap used to avoid touching rows in table.	Selection can be applied to index and either approx. >5% or approx. <25% rows selected or an OR operator is involved in selection that precludes the use of only one index.	Approx. >25% rows selected.	Indexes match selection criteria.	Reduces page I/O to the data space. Allows multiple indexes per table.

Ordering query results

You must specify an ORDER BY clause (or OPNQRYP KEYFLD parameter) to guarantee a particular ordering of the results. Before parallel access methods were available, the database manager processed table rows (and keyed sequences) in a sequential manner. This caused the sequencing of the results to be somewhat predictable (generally rows were retrieved in the order in which they were stored in the dataspace) even though ordering was not included in the original query request. Because parallel methods cause blocks of table rows and key values to be processed concurrently, the ordering of the retrieved results becomes more random and unpredictable.

An ORDER BY clause is the only way to guarantee the specific sequencing of the results. However, an ordering request should only be specified when absolutely required, because the sorting of the results can increase both CPU utilization and response time.

Enabling parallel processing for queries

You must enable parallel processing for queries when you submit them or when you code your applications. The optimizer does not automatically use parallelism as the chosen access method.

You can use the system-value QQRYPDEGREE, the query options file, or the DEGREE parameter on the Change Query Attributes (CHGQRYP) command to control the degree of parallelism that the query optimizer uses. See "Control parallel processing for queries" on page 97 for information on how to control parallel processing.

A set of database system tasks is created at server startup for use by the database manager. The database manager uses the tasks to process and retrieve data from different disk devices. Since these tasks can be run on multiple processors simultaneously, the elapsed time of a query can be reduced. Even though much of the I/O and CPU processing of a parallel query is done by the tasks, the accounting of the

I/O and CPU resources used are transferred to the application job. The summarized I/O and CPU resources for this type of application continue to be accurately displayed by the Work with Active Jobs (WRKACTJOB) command.

Spreading data automatically

DB2 Universal Database for iSeries automatically spreads the data across the disk devices available in the auxiliary storage pool (ASP) where the data is allocated. This ensures that the data is spread without user intervention. The spreading allows the database manager to easily process the blocks of rows on different disk devices in parallel.

Even though DB2 Universal Database for iSeries spreads data across disk devices within an ASP, sometimes the allocation of the data extents (contiguous sets of data) might not be spread evenly. This occurs when there is uneven allocation of space on the devices, or when a new device is added to the ASP. The allocation of the data space may be spread again by saving, deleting, and then restoring the table.

Table scan access method

All rows in the table are read. The selection criteria are applied to each row, and only the rows that match the criteria are returned to the calling application. The rows in the table are processed in no guaranteed order. If you want the result in a particular sequence, you must specify the ORDER BY clause (or OPNQRYF KEYFLD parameter).

Table scan can be efficient for the following reasons:

- It minimizes the number of page I/O operations because all rows in a given page are processed, and once the page is in main storage, the page is not retrieved again.
- The database manager can easily predict the sequence of pages from the dataspace for retrieval. For this reason, the database manager can schedule asynchronous I/O of the pages into main storage from auxiliary storage. This is commonly referred to as **prefetching**. This is done so that the page is available in main storage when the database manager needs to access the data.

Where the table scan access method is most effective

This selection method is good when a large percentage of the rows is to be selected. A large percentage is generally 20% or more.

Where the table scan access method is least effective

Table scan processing is not efficient when a small percentage of rows in the table will be selected. Because all rows in the table are examined, this leads to unnecessary use of I/O and processing unit resources.

Considerations for table scan access

Table scan processing can be adversely affected when rows are selected from a table that contains deleted rows. This is because the delete operation only marks rows as deleted. For table scan processing, the database manager reads all of the deleted rows, even though none of the deleted rows are ever selected. You should use the Reorganize Physical File Member (RGZPFM) CL command to eliminate deleted rows. By specifying REUSEDLT(*YES) on the physical file, you can also reuse the deleted row space. All SQL tables are created with REUSEDLT(*YES).

PRTSQLINF command messages

The messages created by the PRTSQLINF CL command to describe a query in an SQL program which is using the dataspace selection method would appear as follows:

SQL4010 Table scan access for table 1.

Selection algorithms for table scan access method

The Licensed Internal Code can use one of two algorithms for selection when a table scan is processed, derived-column selection and dataspace-only selection. The dataspace-only selection has two forms - dataspace looping and dataspace-only filtering. Dataspace looping processes large sets of records efficiently, while dataspace-only filtering is another step to eliminate records prior to derived operations.

All access methods use dataspace filtering, but dataspace looping is only used when a table scan is processing a high percentage of records.

The following pseudocode illustrates the *derived column selection algorithm*:

```
DO UNTIL END OF TABLE

1. Address the next (or first) row

2. Map all column values to an internal buffer, performing all derived
   operations.

3. Evaluate the selection criteria to a TRUE or FALSE value using
   the column values as they were copied to internal buffer.

4. IF the selection is TRUE
   THEN
   Copy the values from the internal buffer into the
   user's answer buffer.
   ELSE
   No operation
END
```

The *table-scan selection algorithm* is as follows:

```
DO UNTIL END OF TABLE

1. Calculate a search limit. This limit is usually the number of
   rows which are already in active memory, or have already
   had an I/O request done to be loaded into memory.

2. DO UNTIL (search limit reached
   or row selection criteria is TRUE)
   a. Address the next (or first) row

   b. Evaluate any selection criteria which does not
      require a derived value directly for the dataspace
      row.

   END

3. IF the selection is true
   THEN
   a. Map all column values to an internal buffer, performing all
      derived operations.

   b. Copy the values from the internal buffer into the
      user's answer buffer.
   ELSE
   No operation
END
```

The table-scan selection algorithm provides better performance than derived column selection for two reasons:

- Data movement and computations are only done on rows that are selected.

- The loop in step 2 of the table-scan selection algorithm is generated into an executable code burst. When a small percentage of rows is actually selected, DB2 Universal Database for iSeries will be running this small program until a row is found.

Guidelines for coding queries

No action is necessary for queries that use the table scan selection algorithm of the table scan access method. Any query interface can use this improvement. However, the following guidelines determine whether a selection predicate can be implemented as a dataspace selection:

- The optimizer always ensures that the operands for any selection item are compatible, therefore you can improve your queries by making sure the operands are compatible before processing the query.
- Neither operand of the predicate can be any kind of a derived value, function, substring, concatenation, or numeric expression.
- When both operands of a selection predicate are numeric columns, both columns must have the same type, scale, and precision; otherwise, one operand is mapped into a derived value. For example, a DECIMAL(3,1) must only be compared against another DECIMAL(3,1) column.
- When one operand of a selection predicate is a numeric column and the other is a constant or host variable, then the types must be the same and the precision and scale of the constant or host variable must be less than or equal to that of the column.
- Selection predicates involving packed decimal or numeric types of columns can only be done if the table was created by the SQL CREATE TABLE statement.
- A varying-length character column cannot be referenced in the selection predicate.
- When one operand of a selection predicate is a character column and the other is a constant or host variable, then the length of the host variable cannot be greater than that of the column.
- Comparison of character-column data must not require CCSID or keyboard shift translation.

It can be important to avoid derived-column selection because the reduction in CPU and response time for table scan selection can be large, in some cases as high as 70-80%. The queries that will benefit the most from dataspace only selection are those where less than 60% of the table is actually selected. The lower the percentage of rows selected, the more noticeable the performance benefit will be.

Parallel table prefetch access method

DB2 Universal Database for iSeries can also use *parallel table prefetch* processing to shorten the processing time that is required for long-running, I/O-bound table scan queries.

This method has the same characteristics as the table scan method; however, the I/O processing is done in parallel. This is accomplished by starting multiple input streams for the table to prefetch the data.

Where the parallel table prefetch access method is most effective

This method is most effective when the following are true:

- The data is spread across multiple disk devices.
- The optimizer has determined that the query will be I/O bound.
- There is an ample amount of main storage available to hold the data that is collected from every input stream.

DB2 Universal Database for iSeries data spreading

As mentioned previously, DB2 Universal Database for iSeries automatically spreads the data across the disk devices without user intervention, allowing the database manager to prefetch table data in parallel. The database manager uses tasks to retrieve data from different disk devices. Usually the request is for an entire extent (contiguous set of data). This improves performance because the disk device can use

smooth sequential access to the data. Because of this optimization, parallel prefetch can preload data to active memory faster than the SETOBJACC CL command.

Even though DB2 Universal Database for iSeries spreads data across disk devices within an ASP, sometimes the allocation of the dataspace extents may not be spread evenly. This occurs when there is uneven allocation of space on the devices or a new device is added to the ASP. The allocation of the dataspace can be respread by saving, deleting, and restoring the table.

How the query optimizer selects queries that use this method

The query optimizer selects the candidate queries which can take advantage of this type of implementation. The optimizer selects the candidates by estimating the CPU time required to process the query and comparing the estimate to the amount of time required for input processing. When the estimated input processing time exceeds the CPU time, the query optimizer indicates that the query may be implemented with parallel I/O.

If DB2 UDB Symmetric Multiprocessing is installed, then the query optimizer usually prefers the DB2 UDB Symmetric Multiprocessing parallel methods.

Processing requirements

Parallel table prefetch requires that input and output parallel processing must be enabled by the system value QQRDEGREE, by the query option file, or by the DEGREE parameter on the Change Query Attributes (CHGQRYA) command. See “Control parallel processing for queries” on page 97 for information on how to control parallel processing. Because queries being processed with parallel table prefetch aggressively use main storage and disk I/O resources, the number of queries that use parallel table prefetch should be limited and controlled. Parallel prefetch uses multiple disk arms, but it makes little use of multiple CPUs for any given query. Parallel prefetch I/O will use I/O resources intensely. Allowing a parallel prefetch query on a server with an overcommitted I/O subsystem may intensify the over-commitment problem.

You should run the job in a shared storage pool with the *CALC paging option because this causes more efficient use of active memory. DB2 Universal Database for iSeries uses the automated system tuner to determine how much memory this process is allowed to use. At run-time, the Licensed Internal Code will allow parallel table prefetch to be used only if the memory statistics indicate that it will not overcommit the memory resources. For more information on the paging option, see the Automatic System Tuning section of the Work Management topic.

Parallel table prefetch requires that enough memory be available to cache the data that is being retrieved by the multiple input streams. For large tables, the typical extent size is 1 MB. This means that 2 MB of memory must be available to use two input streams concurrently. Increasing the amount of available memory in the pool allows more input streams to be used. If plenty of memory is available, the entire dataspace for the table may be loaded into active memory when the query is opened.

PRTSQLINF command messages

The messages created by the PRTSQLINF command to describe a query in an SQL program which is using the parallel table prefetch access method would appear as follows:

```
SQL4023 Parallel dataspace prefetch used.
```

Parallel table scan method

DB2 Universal Database for iSeries can use this parallel access method to shorten the processing time that is required for long-running table scan queries when the DB2 UDB Symmetric Multiprocessing feature is installed. The parallel table scan method reduces the I/O processing time like the parallel table prefetch access method. In addition, if running on a server that has more than one processor, this method can

reduce the elapsed time of a query by splitting the table scan processing into tasks that can be run on the multiple processors simultaneously. All selection and column processing is performed in the task. The application's job schedules the work requests to the tasks and merges the results into the result buffer that is returned to the application.

Where the parallel table scan access method is most effective

This method is most effective when the following are true:

- The data is spread across multiple disk devices.
- The server has multiple processors that are available.
- There is an ample amount of main storage available to hold the data buffers and result buffers.
- When used for large tables in an OLAP or batch environment.

How the query optimizer selects queries that use this method

As mentioned previously, DB2 Universal Database for iSeries automatically spreads the data across the disk devices without user intervention, allowing the database manager to prefetch table data in parallel. This allows each task to concentrate on its share of the striped data stored away. This way there is no contention on any of the tasks to gain access to the data and perform their portion of the query.

The query optimizer selects the candidate queries that can take advantage of this type of implementation. The optimizer selects the candidates by estimating the CPU time required to process the query and comparing the estimate to the amount of time required for input processing. The optimizer reduces its estimated elapsed time for table scan based on the number of tasks it calculates should be used. It calculates the number of tasks based on the number of processors in the server, the amount of memory available in the job's pool, and the current value of the DEGREE query attribute. If the parallel table scan is the fastest access method, it is then chosen.

Processing requirements

Parallel table scan requires that SMP parallel processing must be enabled either by the system value QQRDEGREE, the query option file, or by the DEGREE parameter on the Change Query Attributes (CHGQRYA) command. See "Control parallel processing for queries" on page 97 for information on how to control parallel processing.

Parallel table scan cannot be used for queries that require any of the following:

- Specification of the *ALL commitment control level.
- Nested loop join implementation. See "Nested loop join implementation" on page 35.
- Backward scrolling. For example, parallel table scan cannot normally be used for queries defined by the Open Query File (OPNQRYF) command, which specify ALWCOPYDTA(*YES) or ALWCOPYDTA(*NO), because the application might attempt to position to the last row and retrieve previous rows. SQL-defined queries that are not defined as scrollable can use this method. Parallel table scan can be used during the creation of a temporary result, such as a sort or hash operation, no matter what interface was used to define the query. OPNQRYF can be defined as not scrollable by specifying the *OPTIMIZE parameter value for the ALWCOPYDTA parameter, which enables the usage of most of the parallel access methods.
- Restoration of the cursor position. For instance, a query requiring that the cursor position be restored as the result of the SQL ROLLBACK HOLD statement or the ROLLBACK CL command. SQL applications using a commitment control level other than *NONE should specify *ALLREAD as the value for precompiler parameter ALWBLK to allow this method to be used.
- Update or delete capability.

You should run the job in a shared storage pool with the *CALC paging option, as this will cause more efficient use of active memory. For more information on the paging option see the Automatic System Tuning section of the Work Management topic in the iSeries Information Center.

Parallel table scan requires active memory to buffer the data that is being retrieved, and to separate result buffers for each task. A typical total amount of memory that is needed for each task is about 2 megabytes. For example, about 8 megabytes of memory must be available in order to use 4 parallel table scan tasks concurrently. Increasing the amount of available memory in the pool allows more input streams to be used. Queries that access tables with large varying length character columns, or queries that generate result values that are larger than the actual row length of the table might require more memory for each task.

The performance of parallel table scan can be severely limited if numerous row locking conflicts or data mapping errors occur.

Index scan-key selection access method

This access method requires indexes. The entire index is read, and any selection criteria that references the key columns of the index are applied against the index. The advantage of this method is that the dataspace is only accessed to retrieve rows that satisfy the selection criteria applied against the index. Any additional selection not performed through the index scan-key selection method is performed at the dataspace level.

The index scan-key selection access method can be very expensive if the search condition applies to a large number of rows because:

- The whole index is processed.
- For every key selected from the index, a random I/O to the dataspace occurs.

How the query optimizer selects queries that use this method

Normally, the optimizer would choose to use table scan processing when the search condition applies to a large number of rows. The optimizer only chooses the index scan-key selection method if less than 20% of the keys are selected or if an operation forces the use of an index. Operations that might force the use of an index include:

- Ordering
- Grouping
- Joining

In these cases, the optimizer may choose to create a temporary index rather than use an existing index. When the optimizer creates a temporary index, it uses a 64K page size for primary dials and an 8K page size for secondary dials. An index created using the SQL CREATE INDEX statement uses 64K page size. For indexes that are created using the CRTLF command, or for SQL indexes created before V4R5M0, the index size is normally 16K.

The optimizer also processes as much of the selection as possible while building the temporary index. Nearly all temporary indexes built by the optimizer are select/omit or sparse indexes. Finally, the optimizer can use multiple parallel tasks when creating the index. The page size difference, corresponding performance improvement from swapping in fewer pages, and the ability to use parallel tasks to create the index may be enough to overcome the overhead cost of creating an index. Dataspace selection is used for building of temporary indexes.

If index scan-key selection access method is used because the query specified ordering (an index was required) the query performance might be improved by using the following parameters to allow the ordering to be done with the query sort.

- For SQL, the following combinations of precompiler parameters:
 - ALWCPYDTA(*OPTIMIZE), ALWBLK(*ALLREAD), and COMMIT(*CHG or *CS)

- ALWCPYDTA(*OPTIMIZE) and COMMIT(*NONE)
- For OPNQRYF, the following parameters:
 - *ALWCPYDTA(*OPTIMIZE) and COMMIT(*NO)
 - ALWCPYDTA(*OPTIMIZE) and COMMIT(*YES) and the commitment control level is started with a commit level of *NONE, *CHG, or *CS

When a query specifies a select/omit index and the optimizer decides to build a temporary index, all of the selection from the select/omit index is put into the temporary index after any applicable selection from the query.

Parallel index scan-key selection access method (available only when the DB2 UDB Symmetric Multiprocessing feature is installed)

For the parallel index scan-key selection access method, the possible key values are logically partitioned. Each partition is processed by a separate task just as in the index scan-key selection access method. The number of partitions processed concurrently is determined by the query optimizer. Because the keys are not processed in order, this method cannot be used by the optimizer if the index is being used for ordering. Key partitions that contain a larger portion of the existing keys from the index are further split as processing of other partitions complete.

Where the parallel index scan-key selection access method is most effective

The following example illustrates a query where the optimizer could choose the index scan-key selection method:

```
CREATE INDEX X1 ON EMPLOYEE(LASTNAME,WORKDEPT)

DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE WORKDEPT = 'E01'
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
          QRYSLT('WORKDEPT *EQ ''E01''')
```

If the optimizer chooses to run this query in parallel with a degree of four, the following might be the logical key partitions that get processed concurrently:

LASTNAME values	LASTNAME values
leading character	leading character
partition start	partition end
'A'	'F'
'G'	'L'
'M'	'S'
'T'	'Z'

If there were fewer keys in the first and second partition, processing of those key values would complete sooner than the third and fourth partitions. After the first two partitions are finished, the remaining key values in the last two might be further split. The following shows the four partitions that might be processed after the first and second partition are finished and the splits have occurred:

LASTNAME values	LASTNAME values
leading character	leading character
partition start	partition end
'O'	'P'
'Q'	'S'
'V'	'W'
'X'	'Z'

Processing requirements

Parallel index scan-key selection cannot be used for queries that require any of the following:

- Specification of the *ALL commitment control level.
- Nested loop join implementation. See “Nested loop join implementation” on page 35.
- Backward scrolling. For example, parallel index scan-key selection cannot be used for queries defined by the Open Query File (OPNQRYF) command which specify ALWCPYDTA(*YES) or ALWCPYDTA(*NO), because the application might attempt to position to the last row and retrieve previous rows. OPNQRYF can be defined as not scrollable by specifying the *OPTIMIZE parameter value for the ALWCPYDTA parameter, which enables the usage of most of the parallel access methods. SQL defined queries that are not defined as scrollable can use this method. Parallel index scan-key selection can be used during the creation of a temporary result, such as a sort or hash operation, no matter what interface was used to define the query.
- Restoration of the cursor position (for instance, a query requiring that the cursor position be restored as the result of the SQL ROLLBACK HOLD statement or the ROLLBACK CL command). SQL applications using a commitment control level other than *NONE should specify *ALLREAD as the value for precompiler parameter ALWBLK to allow this method to be used.
- Update or delete capability.

You should run the job in a shared pool with *CALC paging option as this will cause more efficient use of active memory. For more information on the paging option see the Automatic System Tuning section of the Work Management topic in the iSeries Information Center.

Parallel index scan-key selection requires that SMP parallel processing be enabled either by the system value QQRVDEGREE, the query options file, or by the DEGREE parameter on the Change Query Attributes (CHGQRYA) command. See “Control parallel processing for queries” on page 97 for information on how to control parallel processing.

Index scan-key positioning access method

This access method is very similar to the index scan-key selection access method. They both require a keyed sequence index. In the index scan-key selection access method, processing starts at the beginning of the index and continues to the end; all keys are paged in. In the index scan-key positioning access method, selection is against the index directly on a range of keys that match some or all of the selection criteria. Only those keys from this range are paged in and any remaining index selection is performed by the index scan-key selection method. Any selection not performed through index scan-key positioning or index scan-key selection is performed at the dataspace level. Because index scan-key positioning only retrieves a subset of the keys in the index, the performance of the index scan-key positioning method is better than the performance of the index scan-key selection method.

Where the index scan-key positioning access method is most efficient

The index scan-key positioning method is most efficient when a small percentage of rows are to be selected (less than approximately 20%). If more than approximately 20% of the rows are to be selected, the optimizer generally chooses to:

- Use table scan processing (if index is not required)
- Use index scan-key selection (if an index is required)
- Use query sort routine (if conditions apply)

How the query optimizer selects queries that use this method

For queries that do not require an index (no ordering, grouping, or join operations), the optimizer tries to find an existing index to use for index scan-key positioning. If no existing index can be found, the optimizer stops trying to use keyed access to the data because it is faster to use table scan processing than it is to build an index and then perform index scan-key positioning.

The following example illustrates a query where the optimizer could choose the index scan-key positioning method:

```
CREATE INDEX X1 ON EMPLOYEE(WORKDEPT)

DECLARE BROWSE2 CURSOR FOR
  SELECT * FROM EMPLOYEE
  WHERE WORKDEPT = 'E01'
  OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
  QRYSLT('WORKDEPT *EQ ''E01''')
```

In this example, the database support uses *X1* to position to the first index entry with the *WORKDEPT* value equal to 'E01'. For each key equal to 'E01', it randomly accesses the dataspace² and selects the row. The query ends when the index scan-key selection moves beyond the key value of E01.

Note that for this example all index entries processed and rows retrieved meet the selection criteria. If additional selection is added that cannot be performed through index scan-key positioning (such as selection columns which do not match the first key columns of an index over multiple columns) the optimizer uses index scan-key selection to perform as much additional selection as possible. Any remaining selection is performed at the dataspace level.

The messages created by the PRTSQLINF CL command to describe this query in an SQL program would appear as follows:

```
SQL4008 Index X1 used for table 1.
SQL4011 Key row positioning used on table 1.
```

The index scan-key positioning access method has additional processing capabilities. One such capability is to perform range selection across several values. For example:

```
CREATE INDEX X1 EMPLOYEE(WORKDEPT)

DECLARE BROWSE2 CURSOR FOR
  SELECT * FROM EMPLOYEE
  WHERE WORKDEPT BETWEEN 'E01' AND 'E11'
  OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
  QRYSLT('WORKDEPT *EQ %RANGE(''E01'' ''E11'')')
```

In the previous example, the database support positions to the first index entry equal to value 'E01' and rows are processed until the last index entry for 'E11' is processed.

PRTSQLINF command messages

The messages created by PRTSQLINF CL command to describe this query in an SQL program would appear as follows:

```
SQL4008 Index X1 used for table 1.
SQL4011 Key row positioning used on table 1.
```

Multi-range index scan-key positioning

2. random accessing occurs because the keys may not be in the same sequence as the rows in the dataspace

A further extension of this access method, called *multi-range* index scan-key positioning, is available. It allows for the selection of rows for multiple ranges of values for the first key columns of an index over multiple columns.

```
CREATE INDEX X1 ON EMPLOYEE(WORKDEPT)

DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE WORKDEPT BETWEEN 'E01' AND 'E11'
OR WORKDEPT BETWEEN 'A00' AND 'B01'
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
QRYSLT('WORKDEPT *EQ %RANGE(''E01'' ''E11'')
*OR WORKDEPT *EQ %RANGE(''A00'' ''B01'')')
```

In the previous example, the positioning and processing technique is used twice, once for each range of values.

The messages created by PRTSQLINF CL command to describe this query in an SQL program would appear as follows:

```
SQL4008 Index X1 used for table 1.
SQL4011 Key row positioning used on table 1.
```

All of the index scan-key positioning examples have so far only used one key, the left-most key, of the index. Index scan-key positioning also handles more than one key (although the keys must be contiguous to the left-most key).

```
CREATE INDEX X2
ON EMPLOYEE(WORKDEPT, LASTNAME, FIRSTNME)

DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE WORKDEPT = 'D11'
AND FIRSTNME = 'DAVID'
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
QRYSLT('WORKDEPT *EQ ''D11''
*AND FIRSTNME *EQ ''DAVID'')
```

Because the two selection keys (WORKDEPT and FIRSTNME) are not contiguous, there is no multiple key position support for this example. Therefore, only the WORKDEPT = 'D11' part of the selection can be applied against the index (single key index scan-key positioning). While this may be acceptable, it means that the processing of rows starts with the first key of 'D11' and then uses index scan-key selection to process the FIRSTNME = 'DAVID' against all 9 entries with WORKDEPT key value = 'D11'.

By creating the following index, X3, the above example query would run using multiple keys to do the index scan-key positioning.

```
CREATE INDEX X3
ON EMPLOYEE(WORKDEPT, FIRSTNME, LASTNAME)
```

Multiple key index scan-key positioning support can apply both pieces of selection as index scan-key positioning. This improves performance considerably. A starting value is built by concatenating the two selection values into 'D11DAVID' and selection is positioned to the index entry whose left-most two keys have that value.

The messages created by the PRTSQLINF CL command when used to describe this query in an SQL program would look like this:

```
SQL4008 Index X3 used for table 1.
SQL4011 Key row positioning used on table 1.
```

This next example shows a more interesting use of multiple index scan-key positioning.

```
CREATE INDEX X3 ON EMPLOYEE(WORKDEPT,FIRSTNME)

DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE WORKDEPT = 'D11'
AND FIRSTNME IN ('DAVID','BRUCE','WILLIAM')
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
QRYSLT('WORKDEPT *EQ 'D11''
*AND FIRSTNME *EQ %VALUES('DAVID' 'BRUCE'
'WILLIAM'))')
```

The query optimizer analyzes the WHERE clause and rewrites the clause into an equivalent form:

```
DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE (WORKDEPT = 'D11' AND FIRSTNME = 'DAVID')
OR (WORKDEPT = 'D11' AND FIRSTNME = 'BRUCE')
OR (WORKDEPT = 'D11' AND FIRSTNME = 'WILLIAM')
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
QRYSLT('(WORKDEPT *EQ 'D11'' *AND FIRSTNME *EQ
'DAVID'))
*OR (WORKDEPT *EQ 'D11'' *AND FIRSTNME *EQ 'BRUCE')
*OR (WORKDEPT *EQ 'D11'' *AND FIRSTNME *EQ 'WILLIAM'))')
```

In the rewritten form of the query there are actually 3 separate ranges of key values for the concatenated values of WORKDEPT and FIRSTNME:

Index X3 Start value	Index X3 Stop value
'D11DAVID'	'D11DAVID'
'D11BRUCE'	'D11BRUCE'
'D11WILLIAM'	'D11WILLIAM'

Index scan-key positioning is performed over each range, significantly reducing the number of keys selected to just 3. All of the selection can be accomplished through index scan-key positioning.

The complexity of this range analysis can be taken to a further degree in the following example:

```
DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE (WORKDEPT = 'D11'
AND FIRSTNME IN ('DAVID','BRUCE','WILLIAM'))
OR (WORKDEPT = 'E11'
AND FIRSTNME IN ('PHILIP','MAUDE'))
OR (FIRSTNME BETWEEN 'CHRISTINE' AND 'DELORES'
AND WORKDEPT IN ('A00','C01'))
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
QRYSLT('(WORKDEPT *EQ 'D11''
*AND FIRSTNME *EQ %VALUES('DAVID' 'BRUCE' 'WILLIAM'))')
```

```

*OR (WORKDEPT *EQ 'E11'
*AND FIRSTNME *EQ %VALUES('PHILIP' 'MAUDE'))
*OR (FIRSTNME *EQ %RANGE('CHRISTINE' 'DELORES'))
*AND WORKDEPT *EQ %VALUES('A00' 'C01'))')

```

The query optimizer analyzes the WHERE clause and rewrites the clause into an equivalent form:

```

DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE (WORKDEPT = 'D11' AND FIRSTNME = 'DAVID')
      OR (WORKDEPT = 'D11' AND FIRSTNME = 'BRUCE')
      OR (WORKDEPT = 'D11' AND FIRSTNME = 'WILLIAM')
      OR (WORKDEPT = 'E11' AND FIRSTNME = 'PHILIP')
      OR (WORKDEPT = 'E11' AND FIRSTNME = 'MAUDE')
      OR (WORKDEPT = 'A00' AND FIRSTNME BETWEEN
          'CHRISTINE' AND 'DELORES')
      OR (WORKDEPT = 'C01' AND FIRSTNME BETWEEN
          'CHRISTINE' AND 'DELORES')
OPTIMIZE FOR 99999 ROWS

```

OPNQRYF example:

```

OPNQRYF FILE((EMPLOYEE))
QRYSLT(' (WORKDEPT *EQ 'D11' *AND FIRSTNME *EQ
'DAVID')
*OR (WORKDEPT *EQ 'D11' *AND FIRSTNME *EQ 'BRUCE')
*OR (WORKDEPT *EQ 'D11' *AND FIRSTNME *EQ
'WILLIAM')
*OR (WORKDEPT *EQ 'E11' *AND FIRSTNME *EQ 'PHILIP')
*OR (WORKDEPT *EQ 'E11' *AND FIRSTNME *EQ 'MAUDE')
*OR (WORKDEPT *EQ 'A00' *AND
FIRSTNME *EQ %RANGE('CHRISTINE' 'DELORES'))
*OR (WORKDEPT *EQ 'C01' *AND
FIRSTNME *EQ %RANGE('CHRISTINE' 'DELORES'))')

```

In the query there are actually 7 separate ranges of key values for the concatenated values of WORKDEPT and FIRSTNME:

Index X3 Start value	Index X3 Stop value
'D11DAVID'	'D11DAVID'
'D11BRUCE'	'D11BRUCE'
'D11WILLIAM'	'D11WILLIAM'
'E11MAUDE'	'E11MAUDE'
'E11PHILIP'	'E11PHILIP'
'A00CHRISTINE'	'A00DELORES'
'C01CHRISTINE'	'C01DELORES'

Index scan-key positioning is performed over each range. Only those rows whose key values fall within one of the ranges are returned. All of the selection can be accomplished through index scan-key positioning. This significantly improves the performance of this query.

Parallel index scan-key positioning access method (available only when the DB2 UDB Symmetric Multiprocessing feature is installed)

Using the parallel index scan-key positioning access method, the existing key ranges are processed by separate tasks concurrently in separate database tasks. The number of concurrent tasks is controlled by the optimizer. The query will start processing the key ranges of the query up to the degree of parallelism being used. As processing of those ranges completes, the next ones on the list are started. As processing for a range completes and there are no more ranges in the list to process, ranges that still have keys left to process are split, just as in the parallel index scan-key selection method. The database manager attempts to keep all of the tasks that are being used busy, each processing a separate key range. Whether using the single value, range of values, or multi-range index scan-key positioning, the ranges can be further partitioned and processed simultaneously. Because the keys are not processed in order, this method can not be used by the optimizer if the index is being used for ordering.

How the query optimizer uses this method

Consider the following example if the SQL statement is run using parallel degree of four.

```
DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE (WORKDEPT = 'D11' AND FIRSTNME = 'DAVID')
      OR (WORKDEPT = 'D11' AND FIRSTNME = 'BRUCE')
      OR (WORKDEPT = 'D11' AND FIRSTNME = 'WILLIAM')
      OR (WORKDEPT = 'E11' AND FIRSTNME = 'PHILIP')
      OR (WORKDEPT = 'E11' AND FIRSTNME = 'MAUDE')
      OR (WORKDEPT = 'A00' AND FIRSTNME BETWEEN
          'CHRISTINE' AND 'DELORES')
      OR (WORKDEPT = 'C01' AND FIRSTNME BETWEEN
          'CHRISTINE' AND 'DELORES')

OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
QRYSLT(' (WORKDEPT *EQ 'D11' *AND FIRSTNME *EQ 'DAVID')
*OR (WORKDEPT *EQ 'D11' *AND FIRSTNME *EQ 'BRUCE')
*OR (WORKDEPT *EQ 'D11' *AND FIRSTNME *EQ 'WILLIAM')
*OR (WORKDEPT *EQ 'E11' *AND FIRSTNME *EQ 'PHILIP')
*OR (WORKDEPT *EQ 'E11' *AND FIRSTNME *EQ 'MAUDE')
*OR (WORKDEPT *EQ 'A00' *AND
FIRSTNME *EQ %RANGE('CHRISTINE' 'DELORES'))
*OR (WORKDEPT *EQ 'C01' *AND
FIRSTNME *EQ %RANGE('CHRISTINE' 'DELORES'))')
```

The key ranges the database manager starts with are as follows:

	Index X3 Start value	Index X3 Stop value
Range 1	'D11DAVID'	'D11DAVID'
Range 2	'D11BRUCE'	'D11BRUCE'
Range 3	'D11WILLIAM'	'D11WILLIAM'
Range 4	'E11MAUDE'	'E11MAUDE'
Range 5	'E11PHILIP'	'E11PHILIP'
Range 6	'A00CHRISTINE'	'A00DELORES'
Range 7	'C01CHRISTINE'	'C01DELORES'

Ranges 1 to 4 are processed concurrently in separate tasks. As soon as one of those four completes, range 5 is started. When another range completes, range 6 is started, and so on. When one of the four ranges in progress completes and there are no more new ones in the list to start, the remaining work left in one of the other key ranges is split and each half is processed separately.

Processing requirements

Parallel index scan-key positioning cannot be used for queries that require any of the following:

- Specification of the *ALL commitment control level.
- Nested loop join implementation. See “Nested loop join implementation” on page 35.
- Backward scrolling. For example, parallel index scan-key positioning cannot be used for queries defined by the Open Query File (OPNQRYF) command, which specify ALWCPYDTA(*YES) or ALWCPYDTA(*NO), because the application might attempt to position to the last row and retrieve previous rows. SQL-defined queries that are not defined as scrollable can use this method. Parallel index scan-key positioning can be used during the creation of a temporary result, such as a sort or hash operation, no matter what interface was used to define the query. OPNQRYF can be defined as not scrollable by specifying the *OPTIMIZE parameter value for the ALWCPYDTA parameter, which enables the usage of most of the parallel access methods.
- Restoration of the cursor position. For instance, a query requiring that the cursor position be restored as the result of the SQL ROLLBACK HOLD statement or the ROLLBACK CL command. SQL applications

using a commitment control level other than *NONE should specify *ALLREAD as the value for precompiler parameter ALWBLK to allow this method to be used.

- Update or delete capability.

You should run the job in a shared pool with the *CALC paging option as this will cause more efficient use of active memory. For more information on the paging option see the Automatic System Tuning section of the Work Management topic in the iSeries Information Center.

Parallel index scan-key selection requires that SMP parallel processing be enabled either by the system value QQRVDEGREE, by the query options file PARALLEL_DEGREE option, or by the DEGREE parameter on the Change Query Attributes (CHGQRYA) command. See “Control parallel processing for queries” on page 97 for information on how to control parallel processing.

Index Only Access Method

The index-only access method can be used in conjunction with any of the index scan-key selection or index scan-key positioning access methods, including the parallel options for these methods. (The parallel options are available only when the DB2 UDB Symmetric Multiprocessing feature is installed.) The processing for the selection does not change from what has already been described for these methods.

However, all of the data is extracted from the index rather than performing a random I/O to the data space. The index entry is then used as the input for any derivation or result mapping that might have been specified on the query.

Where the index-only method is most effective

The optimizer chooses this method when:

- All of the columns that are referenced within the query can be found within a permanent index or within the key columns of a temporary index that the optimizer has decided to create.
- The data values must be able to be extracted from the index and returned to the user in a readable format; in other words, none of the key columns that match the query columns have:
 - Absolute value specified
 - Alternative collating sequence or sort sequence specified
 - Zoned or digit force specified
- The query does not use a left outer join or an exception join.
- For non-SQL users, no variable length or null capable columns can require key feedback.

The following example illustrates a query where the optimizer could choose to perform index only access.

```
CREATE INDEX X2
ON EMPLOYEE(WORKDEPT, LASTNAME, FIRSTNME)

DECLARE BROWSE2 CURSOR FOR
SELECT FIRSTNME FROM EMPLOYEE
WHERE WORKDEPT = 'D11'
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
QRYSLT('WORKDEPT *EQ 'D11''')
```

In this example, the database manager uses X2 to position to the index entries for WORKDEPT='D11' and then extracts the value for the column FIRSTNME from those entries.

Note that the index key columns do not have to be contiguous to the leftmost key of the index for index only access to be performed. Any key column in the index can be used to provide data for the index only query. The index is used simply as the source for the data so the database manager can finish processing the query after the selection has been completed.

Note: Index only access is implemented on a particular table, so it is possible to perform index only access on some or all of the tables of a join query.

PRTSQLINF command messages

The messages created by the PRTSQLINF command to describe this query in an SQL program are as follows:

```
SQL4008 Index X2 used for table 1.  
SQL4011 Key row positioning used on table 1.  
SQL4022 Index only access used on table 1.
```

Parallel table or index based preload access method

Some queries implemented with index scan-key selection can require a lot of random I/O in order to access the dataspace. Because of this, a high percentage of the data in the dataspace is referenced. DB2 Universal Database for iSeries attempts to avoid this random I/O by initiating index- or table-based preload when query processing begins. The entire table or index is loaded into active memory in parallel as is done for parallel table prefetch. This requires that you have enough memory in the pool to load the entire object.

After the table or index is loaded into memory, random access to the data is achieved without further I/O. The DB2 Universal Database for iSeries cost-based query optimizer recognizes the queries and objects that benefit from table or index preloads if I/O parallel processing has been enabled. See “Control parallel processing for queries” on page 97 for information on how to control parallel processing. If DB2 UDB Symmetric Multiprocessing is installed, then the query optimizer usually prefers the DB2 UDB Symmetric Multiprocessing parallel methods.

The parallel preload method can be used with any of the other data access methods. The preload is started when the query is opened and control is returned to the application before the preload is finished. The application continues fetching rows using the other database access methods without any knowledge of preload.

Index-from-index access method

The database manager can build a temporary index from an existing index without having to read all of the rows in the dataspace. Generally speaking, this selection method is one of the most efficient. The temporary index that is created contains entries for only those rows that meet the selection predicates. This is similar to the index created by a select/omit logical file (commonly referred to as a sparse index).

Where the index-from-index access method is most effective

The optimizer chooses this method when:

- The query requires an index because it uses grouping, ordering, or join processing.
- A permanent index exists that has selection columns as the left-most keys and the left-most keys are very selective (i.e., index scan key positioning can be used).
- The selection columns are not the same as the ordering, grouping, or join-to columns.

How the query optimizer uses this method

To use the index-from-index access method, the database manager:

1. Uses index scan-key positioning on the permanent index with the query selection criteria

2. Builds index entries in the new temporary index using selected entries.
3. Key columns of the temporary index match the grouping, ordering or join columns.

The result is an index containing entries in the required keyed sequence (grouping, ordering, join) for rows that match the selection criteria.

A common index-from-index access method example follows:

```
CREATE INDEX X1 ON EMPLOYEE(WORKDEPT)

DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE WORKDEPT = 'D11'
ORDER BY LASTNAME
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
QRYSLT('WORKDEPT *EQ 'D11''')
KEYFLD((LASTNAME))
```

For this example, a temporary select/omit index is created with the primary key column LASTNAME. It contains index entries for only those rows where *WORKDEPT = 'D11'* assuming less than approximately 20% of the entries are selected.

PRTSQLINF command messages

The messages created by the PRTSQLINF CL command to describe this query in an SQL program are as follows:

```
SQL4012 Index created from index X1 for table 1.
SQL4011 Key row positioning used on table 1.
```

Alternatives to the index-from-index method

Rather than using the index-from-index access method, you can use the query sort routine. See “Sort access method” on page 28 for more information.

This decision is based on the number of rows to be retrieved.

Hashing access method

The hashing access method provides an alternative method for those queries (groupings and joins) that must process data in a grouped or correlated manner. Indexes are used to sort and group the data and are effective in some cases for implementing grouping and join query operations. However, if the optimizer had to create a temporary index for that query, extra processor time and resources are used when creating this index before the requested query can be run.

Where the hashing access method is most effective

The hashing access method can complement indexes or serve as an alternative. For each selected row, the specified grouping or join value in the row is run through a hashing function. The computed hash value is then used to search a specific partition of the hash table. A hash table is similar to a temporary work table, but has a different structure that is logically partitioned based on the specified query. If the row’s source value is not found in the table, then this marks the first time that this source value has been encountered in the database table. A new hash table entry is initialized with this first-time value and additional processing is performed based on the query operation. If the row’s source value is found in the table, the hash table entry for this value is retrieved and additional query processing is performed based

on the requested operation (such as grouping or joining). The hash method can only correlate (or group) identical values; the hash table rows are not guaranteed to be sorted in ascending or descending order.

Where this method can be used

The hashing method can be used only when the ALWCOPYDTA(*OPTIMIZE) option has been specified unless a temporary result is required, since the hash table built by the database manager is a temporary copy of the selected rows.

How this method works

The hashing algorithm allows the database manager to build a hash table that is well-balanced, given that the source data is random and distributed. The hash table itself is partitioned based on the requested query operation and the number of source values being processed. The hashing algorithm then ensures that the new hash table entries are distributed evenly across the hash table partitions. This balanced distribution is necessary to guarantee that scans in different partitions of the hash tables are processing the same number of entries. If one hash table partition contains a majority of the hash table entries, then scans of that partition are going to have to examine the majority of the entries in the hash table. This is not very efficient.

Since the hash method typically processes the rows in a table sequentially, the database manager can easily predict the sequence of memory pages from the database table needed for query processing. This is similar to the advantages of the table scan access method. The predictability allows the database manager to schedule asynchronous I/O of the table pages into main storage (also known as pre-fetching). Pre-fetching enables very efficient I/O operations for the hash method leading to improved query performance.

In contrast, query processing with a keyed sequence access method causes a random I/O to the database table for every key value examined. The I/O operations are random since the keyed-order of the data in the index does not match the physical order of the rows in the database table. Random I/O can reduce query performance because it leads to unnecessary use of I/O and processor unit resources.

An index can also be used by the hash method to process the table rows in keyed order. The index can significantly reduce the number of table rows that the hash method has to process. This can offset the random I/O costs associated with indexes.

The hash table creation and population takes place before the query is opened. Once the hash table has been completely populated with the specified database rows, the hash table is used by the database manager to start returning the results of the queries. Additional processing might be required on the resulting hash table rows, depending on the requested query operations.

Since blocks of table rows are automatically spread, the hashing access method can also be performed in parallel so that several groups of rows are being hashed at the same time. This shortens the amount of time it takes to hash all the rows in the database table.

If the DB2 UDB Symmetric Multiprocessing feature is installed, the hashing methods can be performed in parallel.

Bitmap processing method

As the name implies, this method generates bitmaps that are used during access to the data space. The bitmap processing method is used to:

- Minimize the random I/O that occurs on a data space when using an index in conjunction with the key position or when using the index scan-key selection method.
- Schedule the I/O more efficiently by skipping large sections of data within the dataspace when the skip-sequential method is used.

- Allow multiple indexes to be used to access a particular dataspace.

How the bitmap processing method works

In this method, the optimizer chooses one or more indexes to be used to aid in selecting rows from the dataspace. Temporary bitmaps are allocated (and initialized), one for each index. Each bitmap contains one bit for each row in the underlying data space. For each index, index scan-key row positioning and index scan-key row selection methods are used to apply selection criteria when initializing the bitmap.

For each index entry selected, the bit associated with that row is set to 1 (that is, turned on). The data space is not accessed. When the processing of the index is complete, the bitmap contains the information on which rows are to be selected from the underlying data space. This process is repeated for each index. If two or more indexes are used, the temporary bitmaps are logically ANDed and ORed together to obtain one resulting bitmap. Once the resulting bitmap is built, it is used to avoid mapping in rows from the dataspace unless they are selected by the query. It is used to help schedule the selection of rows from the dataspace or to provide another level of filtering prior to the underlying dataspace being accessed.

The indexes used to generate the bitmaps are not actually used to access the selected rows. For this reason, they are called **tertiary indexes**. Conversely, indexes used to access the final rows are called **primary indexes**. Primary indexes are used for ordering, grouping, joining, and for selection when no bitmap is used.

Where the method is used

Bitmaps are always preprocessed before the optimizer starts to process the query through the primary access method. The bitmap processing method is used in conjunction with primary access methods table scan, index scan-key row positioning, or index scan-key row selection. Bitmap processing, like parallel table prefetch and parallel table/index preload, does not actually select the rows from the data space; it simply assists the primary methods.

If the bitmap is used in conjunction with the table scan method, the bitmap initiates skip-sequential processing. The table scan (and parallel table scan) uses the bitmap to "skip over" pages with no selected rows (i.e., no bits in the bitmap are set to 1). This has several advantages:

- No CPU processing is used to process nonselected rows.
- I/O is minimized and the memory is not filled with the contents of the entire data space.

Example: Bitmap processing method used in conjunction with table scan method

The following example illustrates a query where the query optimizer chooses the bitmap processing method in conjunction with the table scan:

```
CREATE INDEX IX1 ON EMPLOYEE (WORKDEPT)
CREATE INDEX IX2 ON EMPLOYEE (SALARY)
```

```
DECLARE C1 CURSOR FOR
  SELECT * FROM EMPLOYEE
  WHERE WORKDEPT = 'E01' OR SALARY>50000
  OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
  QRYSLT('WORKDEPT *EQ 'E01' *OR SALARY > 50000')
```

In this example, both indexes IX1 and IX2 are used. The database manager first generates a bitmap from the results of applying selection WORKDEPT = 'E01' against index IX1 (using index scan-key positioning). The database manager then generates a bitmap from the results of applying selection SALARY>50000 against index IX2 (again using index scan-key positioning).

Next, the database manager combines these two bitmaps into one resulting bitmap by logically ORing the individual bitmaps together. Finally, a table scan is initiated. The table scan uses the bitmap to skip through the data space rows, retrieving only those selected by the bitmap. This improves performance by skipping over large portions of data.

This example also shows an additional capability provided with bitmap processing (use of an index for ANDed selection was already possible but bitmap processing now allows more than one index). When using bitmap processing, multiple index usage is possible with selections where OR is the major Boolean operator.

The messages created by the PRTSQLINF command when used to describe this query would look like:

```
SQL4010 Table scan for table 1.
SQL4032 Index IX1 used for bitmap processing of table 1.
SQL4032 Index IX2 used for bitmap processing of table 1.
CPI4329 Arrival sequence access was used for file EMPLOYEE.
CPI4388 2 access path(s) used for bitmap processing of file EMPLOYEE.
```

Example: Bitmap processing used in conjunction with the index scan-key positioning access method

If the bitmap is used in conjunction with either the index scan-key row positioning or index scan-key row selection method, it implies that the bitmap (generated from tertiary indexes) is being used to aid a primary index access. The following example illustrates a query where bitmap processing is used in conjunction with the index scan-key positioning for a primary index:

```
CREATE INDEX PIX ON EMPLOYEE (LASTNAME)
CREATE INDEX TIX1 ON EMPLOYEE (WORKDEPT)
CREATE INDEX TIX2 ON EMPLOYEE (SALARY)
```

```
DECLARE C1 CURSOR FOR
  SELECT * FROM EMPLOYEE
  WHERE WORKDEPT = 'E01' OR SALARY>50000
  ORDER BY LASTNAME
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE))
  QRYSLT('WORKDEPT *EQ 'E01'' *OR SALARY > 50000')
  KEYFLD(LASTNAME)
```

In this example, indexes TIX1 and TIX2 are used in bitmap processing. The database manager first generates a bitmap from the results of applying selection WORKDEPT = 'E01' against index TIX1 (using index scan-key positioning). It then generates a bitmap from the results of applying selection SALARY>50000 against index TIX2 (again using index scan-key positioning).

The database manager then combines these two bitmaps into one bitmap using OR logic. An index scan-key selection method is initiated using (primary) index PIX. For each entry in index PIX, the bitmap is checked. If the entry is selected by the bitmap, then the data space row is retrieved and processed.

The messages created by the PRTSQLINF CL command, when used to describe this query, would look like:

```
SQL4008 Index PIX used for table 1.
SQL4032 Index TIX1 used for bitmap processing of table 1.
CPI4328 Access path of file PIX was used by query.
CPI4338 2 access path(s) used for bitmap processing of file EMPLOYEE.
```

Example: Bitmap processing used in conjunction with join queries

Bitmap processing can be used for join queries, as well. Since bitmap processing is on a per-table basis, each table of a join can independently use or not use bitmap processing.

The following example illustrates a query where bitmap processing is used against the second table of a join query but not on the first table:

```
CREATE INDEX EPIX ON EMPLOYEE(EMPNO)
CREATE INDEX TIX1 ON EMPLOYEE(WORKDEPT)
CREATE INDEX TIX2 ON EMPLOYEE(SALARY)
DECLARE C1 CURSOR FOR
SELECT * FROM PROJECT, EMPLOYEE
WHERE RESEMP=EMPNO AND
(WORKDEPT='E01' OR SALARY>50000)
```

Using the OPNQRYF command:

```
OPNQRYF FILE((PROJECT) (EMPLOYEE)) FORMAT(RESULTFILE)
          JFLD((1/RESPEMP 2/EMPNO))
          QRYSLT('2/WORKDEPT=''E01'' *OR 2/SALARY>50000')
```

In this example, the optimizer decides that the join order is table PROJECT to table EMPLOYEE. Table scan is used on table PROJECT. For table EMPLOYEE, index EPIX is used to process the join (primary index). Indexes TIX1 and TIX2 are used in bitmap processing.

The database manager positions to the first row in table PROJECT. It then performs the join using index EPIX. Next, it generates a bitmap from the results of applying selection WORKDEPT='E01' against index TIX1 (using index scan-key positioning). It then generates a bitmap from the results of applying selection SALARY>50000 against index TIX2 (again using index scan-key positioning).

Next, the database manager combines these two bitmaps into one bitmap using OR logic. Finally, the entry that EPIX is currently positioned to is checked against the bitmap. The entry is either selected or rejected by the bitmap. If the entry is selected, the rows are retrieved from the underlying data space. Next, index EPIX is probed for the next join row. When an entry is found, it is compared against the bitmap and either selected or rejected. Note that the bitmap was generated only once (the first time it was needed) and is just reused after that.

The query optimizer debug messages put into the job log would look like:

```
CPI4327 File PROJECT processed in join position 1.
CPI4326 File EMPLOYEE processed in join position 2.
CPI4338 2 access path(s) used for bitmap processing of file EMPLOYEE.
```

Bitmap processing and composite key indexes

Bitmap processing alleviates some of the problems associated with having composite key indexes (multiple key columns in one index).

For example, given an SQL query:

```
DECLARE C1 CURSOR FOR
SELECT * FROM EMPLOYEE
WHERE WORKDEPT='D11' AND
FIRSTNAME IN ('DAVID', 'BRUCE', 'WILLIAM')
```

Or the same query using the OPNQRYF command:

```
OPNQRYF FILE((EMPLOYEE))
          QRYSLT('WORKDEPT=''D11'' *AND
          FIRSNME = %VALUES(''DAVID'' ''BRUCE'' ''WILLIAM'')')
```

An index with keys (WORKDEPT, FIRSTNAME) would be the best index to use to satisfy this query. However, two indexes, one with a key of WORKDEPT and the other with a key of FIRSNME could be used in bitmap processing, with their resulting bitmaps ANDed together and table scan used to retrieve the result.

With the bitmap processing method, you can create several indexes, each with only one key column, and have the optimizer use them as general purpose indexes for many queries. You can avoid problems involved with trying to determine the best composite key indexes for all queries being performed against a table. Bitmap processing, in comparison to using a multiple key-column index, allows more ease of use, but at some cost to performance.

Note: Keep in mind that you will always achieve the best performance by using composite key indexes.

Considerations for bitmap processing

Some additional points regarding bitmap processing:

- As long as the DB2 UDB Symmetric Multiprocessing feature is installed, you can use parallel processing whenever you use bitmap processing. In this case, the bitmap is built from the results of performing either parallel index scan-key positioning or parallel index scan-key selection on the tertiary index.
- Bitmaps are preprocessed at the first I/O request for the query. Therefore, the first row fetched may take longer to retrieve than subsequent rows.
- Bitmaps, by their nature, contain static selection, which is controlled by ALWCPYDTA. Once the bitmap is generated, it will not select any new or modified rows. Therefore, bitmap processing is used only when ALWCPYDTA(*OPTIMIZE) is specified.

For example, suppose that an OPNQRYF statement specifying (QRYSLT('QUANTITY >5')) is opened using bitmap processing and the first row is read. While updates to the table will not cause a record to be selected, if you reverse this example where the bitmap was generated before the update and the update caused the record not to be selected, the optimizer will ensure that you do not get this record because of duplicate selection that is applied to the underlying dataspace once it is retrieved. Through a separate database operation, all rows where QUANTITY is equal to 4 are updated so QUANTITY is equal to 10. Since the bitmap was already built (during the first row fetch from the OPNQRYF open identifier), these updated rows will not be retrieved on subsequent fetches through the OPNQRYF open identifier.

The exception to this is when the query contains grouping or one or more aggregate functions (for example, SUM, COUNT, MIN, MAX), in which case static data is already being made.

- The query optimizer does not use bitmap processing for a query that is insert, update, or delete-capable. For OPNQRYF, you must set the OPTION parameter to *INP and the SEQONLY parameter to *YES. There must not be any overrides to SEQONLY(*NO)).

Sort access method

The sort access method provides an alternative method for those queries that must process data in an ordered method (ORDER BY). An index can be used to sort the data, and is effective in some cases for implementing ordering. However, if the optimizer had to create a temporary index for that query, it would use extra processor time and resources when creating the index before the requested query can be run.

Where this method can be used

The optimizer chooses this method in the following circumstances:

- For SQL (see “Effects of the ALWCPYDTA parameter on database performance” on page 130) when you specify either of the following precompile options:
 - ALWCPYDTA(*OPTIMIZE), ALWBLK(*ALLREAD), and COMMIT(*CHG or *CS)
 - ALWCPYDTA(*OPTIMIZE) and COMMIT(*NONE)
- For OPNQRYF, when you specify either of the following options:
 - ALWCPYDTA(*OPTIMIZE) and COMMIT(*NO)
 - ALWCPYDTA(*OPTIMIZE) and COMMIT(*YES) and commitment control is started with a commit level of *NONE, or *CHG, or *CS

- If a temporary result is required prior to the ordering function.
- If the number of order by keys exceeds 120 or the combined length of the sort keys exceeds 2000 bytes.

How this method works

The sort algorithm reads the rows into a sort space and sorts the rows based on the specified ordering keys. The rows are then returned to the user from the ordered sort space.

Chapter 3. The DB2 UDB for iSeries query optimizer: Overview

This overview of the query optimizer provides guidelines for designing queries that will perform and will use server resources more efficiently. This overview covers queries that are optimized by the query optimizer and includes interfaces such as SQL, OPNQRYF, APIs (QQQRY), ODBC, and Query/400 queries. Whether or not you apply the guidelines, the query results will still be correct.

Note: The information in this overview is complex. You might find it helpful to experiment with an iSeries server as you read this information to gain a better understanding of the concepts.

When you understand how DB2 Universal Database for iSeries processes queries, it is easier to understand the performance impacts of the guidelines discussed in this overview. There are two major components of DB2 Universal Database for iSeries query processing:

- How the server accesses data. See “Data access: data access methods” on page 3.
These methods are the algorithms that are used to retrieve data from the disk. The methods include index usage and row selection techniques. In addition, parallel access methods are available with the DB2 UDB Symmetric Multiprocessing operating system feature.
- Query optimizer. See “How the query optimizer makes your queries more efficient”.
The query optimizer identifies the valid techniques which could be used to implement the query and selects the most efficient technique.

How the query optimizer makes your queries more efficient

The optimizer is an important part of DB2 Universal Database for iSeries because the optimizer:

- Makes the key decisions which affect database performance.
- Identifies the techniques which could be used to implement the query.
- Selects the most efficient technique.

Data manipulation statements such as SELECT specify only what data the user wants, not how to retrieve that data. This path to the data is chosen by the optimizer and stored in the access plan. This topic covers the techniques employed by the query optimizer for performing this task including:

- “Cost estimation for queries” on page 32
- “Access plan validation” on page 34
- “Join optimization” on page 35
- “Grouping optimization” on page 50

Optimizer decision-making rules

The optimizer uses a general set of guidelines to choose the best method for accessing data. The optimizer:

- Determines the default filter factor for each predicate in the selection clause.
- Extracts attributes of the table from internally stored information.
- Performs a key range estimate to determine the true filter factor of the predicates when the selection predicates match the left-most keys of an index.
- Determines the cost of table scan processing if an index is not required.
- Determines the cost of creating an index over a table if an index is required. This index is created by performing either a table scan or creating an index-from-index.
- Determines the cost of using a sort routine or hashing method if selection conditions apply and an index is required.

- For each index available, generally in the order of most recently created to oldest, the optimizer does the following until its time limit is exceeded:
 - Extracts attributes of the index from internally stored statistics.
 - Determines if the index meets the selection criteria.
 - Determines the cost of using the index by using the estimated page faults and the predicate filter factors to help determine the cost.
 - Compares the cost of using this index with the previous cost (current best).
 - Picks the cheaper one.
 - Continues to search for best index until its time limit is exceeded or no more indexes.

The *time limit* controls how much time the optimizer spends choosing an implementation. It is based on how much time was spent so far and the current best implementation cost found. The idea is to prevent the optimizer from spending more time optimizing the query than it would take to actually execute the query. Dynamic SQL queries are subject to the optimizer time restrictions. Static SQL queries optimization time is not limited. For OPNQRYF, if you specify OPTALLAP(*YES), the optimization time is not limited.

For small tables, the query optimizer spends little time in query optimization. For large tables, the query optimizer considers more indexes. Generally, the optimizer considers five or six indexes (for each table of a join) before running out of optimization time. Because of this, it is normal for the optimizer to spend longer lengths of time analyzing queries against larger tables.

Cost estimation for queries

At run time, the optimizer chooses an optimal access method for the query by calculating an *implementation cost* based on the current state of the database. The optimizer models the access cost of each of the following:

- Reading rows directly from the table (table scan processing)
- Reading rows through an index (using either index scan-key selection or index scan-key positioning)
- Creating an index directly from the dataspace
- Creating an index from an existing index (index-from-index)
- Using the query sort routine or hashing method (if conditions are satisfied)
- Using bitmap processing

The cost of a particular method is the sum of:

- The start-up cost
- The cost associated with the given optimization mode.
 - **Costs associated with optimization modes when using SQL:** For SQL, the precompile option ALWCPYDTA and the OPTIMIZE FOR n ROWS clause indicate to the query optimizer the optimization goal to be achieved.
 - The optimizer can optimize SQL queries with one of two goals:
 1. Minimize the time required to retrieve the first buffer of rows from the table. This goal biases the optimization towards not creating an index.

Either a data scan or an existing index is preferred. This mode can be specified in two ways:

 - a. The OPTIMIZE FOR n ROWS allows the users to specify the number of rows they expect to retrieve from the query.

The optimizer uses this value to determine the percentage of rows that will be returned and optimizes accordingly. A small value instructs the optimizer to minimize the time required to retrieve the first n rows.
 - b. Specifying ALWCPYDTA(*NONE) or (*YES) as a precompiler parameter allows the optimizer to minimize the time that it requires to retrieve the first 3% of the resulting rows.

This option is effective only if the OPTIMIZE FOR n ROWS was not specified.

2. Minimize the time to process the whole query assuming that all selected rows are returned to the application. This option does not bias the optimizer to any particular access method. This mode can be specified in two ways:

- a. The OPTIMIZE FOR n ROWS allows the users to specify the number of rows they expect to retrieve from the query.

The optimizer uses this value to determine the percentage of rows that will be returned and optimizes accordingly. A value greater than or equal to the expected number of resulting rows instructs the optimizer to minimize the time required to run the entire query.

- b. ALWCPYDTA(*OPTIMIZE) specified as a precompiler parameter.

This option is effective only if the OPTIMIZE FOR n ROWS is not specified.

– **Costs associated with optimization modes when using OPNQRYF:**

- The cost associated with the given optimization parameter (*FIRSTIO, *ALLIO, or *MINWAIT).
 - *FIRSTIO — Minimize the time required to retrieve the first buffer of rows from the table. Biases the optimization toward not creating an index. Either a data scan or an existing index is preferred. When *FIRSTIO is selected, users may also pass in the number of rows they expect to retrieve from the query. The optimizer uses this value to determine the percentage of rows that will be returned and optimizes accordingly. A small value would minimize the time required to retrieve the first n rows, similar to *FIRSTIO. A large value would minimize the time to retrieve all n rows, similar to *ALLIO.
 - *ALLIO — Minimize the time to process the whole query assuming that all query rows are read from the table. This option does not bias the optimizer to any particular access method.

Note: If you specify ALWCPYDTA(*OPTIMIZE) and the query optimizer decides to use the sort routine, your query resolves according to the *ALLIO optimize parameter.

- *MINWAIT—Minimize delays when reading rows from the table. Minimize I/O time at the expense of open time. This option biases optimization toward either creating a temporary index or performing a sort. Either an index is created or an existing index is used.
- The cost of any index creations
- The cost of the expected number of page faults to read the rows and the cost of processing the expected number of rows.

Page faults and number of rows processed may be predicted by statistics the optimizer can obtain from the database objects, including:

- Table size
- Row size
- Index size
- Key size

Page faults can also be greatly affected if index only access can be performed, thus eliminating any random input and output to the dataspace.

A weighted measure of the expected number of rows to process is based on what the relational operators in the row selection predicates, *default filter factors*, are likely to retrieve:

- 10% for equal
- 33% for less-than, greater-than, less-than-equal-to, or greater-than-equal-to
- 90% for not equal
- 25% for BETWEEN range (OPNQRYF %RANGE)
- 10% for each IN list value (OPNQRYF %VALUES)

Key range estimate is a method the optimizer uses to gain more accurate estimates of the number of expected rows to be selected from one or more selection predicates. The optimizer estimates by applying the selection predicates against the left-most keys of an existing index. The default filter factors can then be further refined by the estimate based on the key range. If an index exists whose left-most keys match columns used in row selection predicates, that index can be used to estimate the number of entries that match the selection criteria. The estimate of the number of entries is based on the number of pages and key density of the machine index and is done without actually accessing the entries. Creating indexes over columns that are used in selection predicates can significantly help optimization.

Page faults and the number of rows processed are dependent on the type of access the optimizer chooses. Refer to Chapter 2, “Data access on DB2 UDB for iSeries: data access paths and methods” on page 3 for more information on access methods.

General query optimization tips

Here are some tips to help your queries run as fast as possible:

- Create indexes whose leftmost key columns match your selection predicates to help supply the optimizer with selectivity values (key range estimates).
- For join queries, create indexes that match your join columns to help the optimizer determine the average number of matching rows.
- Minimize extraneous mapping by specifying only columns of interest on the query. For example, specify only the columns you need to query on the SQL SELECT statement instead of specifying SELECT *. Also, you should specify FOR FETCH ONLY if the columns do not need to be updated.
- If your queries often use table scan access method, use the RGZPFM (Reorganize Physical File Member) command to remove deleted rows from tables or the CHGPF (Change Physical File) REUSEDLT (*YES) command to reuse deleted rows.

For embedded SQL, consider using the following precompile options:

- Specify ALWCOPYDTA(*OPTIMIZE) to allow the query optimizer to create temporary copies of data so better performance can be obtained.
- Use CLOSWLCSR(*ENDJOB) or CLOSQLCSR(*ENDACTGRP) to allow open data paths to remain open for future invocations.
- Specify DLYPRP(*YES) to delay SQL statement validation until an OPEN, EXECUTE, or DESCRIBE statement is run. This option improves performance by eliminating redundant validation.
- Use ALWBLK(*ALLREAD) to allow row blocking for read-only cursors.

For OPNQRYF (Open Query File) queries, consider using the following parameters:

- Use ALWCOPYDTA(*OPTIMIZE) to let the query optimizer create temporary copies of data if it can obtain better performance by doing so.
- Use OPTIMIZE(*FIRSTIO) to bias the optimizer to use an existing index instead of creating a temporary index.

Access plan validation

An access plan is a control structure that describes the actions necessary to satisfy each query request. An access plan contains information about the data and how to extract it. For any query, whenever optimization occurs, the query optimizer develops an optimized plan of how to access the requested data. The information is kept in what is called a mini plan. The mini plan, along with the query definition template (QDT), is used to interface with the optimizer and make an access plan.

- For dynamic SQL, an access plan is created, but the plan is not saved. A new access plan is created each time the PREPARE statement runs.
- For an iSeries program that contains static embedded SQL, the access plan is saved in the *associated space* of the program or package that contains embedded SQL statements.

- For OPNQRYF, an access plan is created but is not saved. A new access plan is created each time the OPNQRYF command is processed.
- For Query/400, an access plan is saved as part of the query definition object.

Join optimization

A join operation is a complex function that requires special attention in order to achieve good performance. This section describes how DB2 Universal Database for iSeries implements join queries and how optimization choices are made by the query optimizer. It also describes design tips and techniques which help avoid or solve performance problems. Among the topics discussed are:

- Nested loop join implementation
 - Index nested loop join
 - Arrival sequence nested loop join
- Hash joins
- Cost estimation and index selection for join secondary dials
- Tips for improving the performance of join queries

Nested loop join implementation

DB2 Universal Database for iSeries provides a **nested loop** join method. For this method, the processing of the tables in the join are ordered. This order is called the **join order**. The first table in the final join order is called the **primary table**. The other tables are called **secondary tables**. Each join table position is called a **dial**. The nested loop will be implemented either using an index on secondary tables or a table scan (arrival sequence) on the secondary tables. In general, the join will be implemented using an index. The join may be implemented with a table scan when the secondary table is a user-defined table function.

Index nested loop join implementation: During the join, DB2 Universal Database for iSeries:

1. Accesses the first primary table row selected by the predicates local to the primary table.
2. Builds a key value from the join columns in the primary table.
3. Uses index scan-key positioning to locate the first row that satisfies the join condition for the first secondary table using an index with keys matching the join condition or local row selection columns of the secondary table.
4. Applies bitmap selection, if applicable.
5. Determines if the row is selected by applying any remaining selection local to the first secondary dial. If the secondary dial row is not selected then the next row that satisfies the join condition is located. Steps 1 through 5 are repeated until a row that satisfies both the join condition and any remaining selection is selected from all secondary tables
6. Returns the result join row.
7. Processes the last secondary table again to find the next row that satisfies the join condition in that dial.

During this processing, when no more rows that satisfy the join condition can be selected, the processing backs up to the logical previous dial and attempts to read the next row that satisfies its join condition.
8. Ends processing when all selected rows from the primary table are processed.

Nested loop join characteristics:

Note the following characteristics of a nested loop join:

- If ordering or grouping is specified and all the columns are over a single table and that table is eligible to be the primary, then that table becomes the primary table and is processed with an index over the table.
- If ordering and grouping is specified on two or more tables, DB2 Universal Database for iSeries breaks the processing of the query into two parts:

1. Perform the join selection omitting the ordering or grouping processing and write the result rows to a temporary work table. This allows the optimizer to consider any table of the join query as a candidate for the primary table.
2. The ordering or grouping processing is then performed on the data in the temporary work table.

The query optimizer might also decide to break the query into these two parts to improve performance when the SQL ALWCPYDTA(*OPTIMIZE) precompiler parameter or the OPNQRYF KEYFLD, and ALWCPYDTA(*OPTIMIZE) parameters are specified.

- All rows that satisfy the join condition from each secondary dial are located using an index. Rows are retrieved from secondary tables in random sequence. This random disk I/O time often accounts for a large percentage of the processing time of the query. Since a given secondary dial is searched once for each row selected from the primary and the preceding secondary dials that satisfy the join condition for each of the preceding secondary dials, a large number of searches may be performed against the later dials. Any inefficiencies in the processing of the later dials can significantly inflate the query processing time. This is the reason why attention to performance considerations for join queries can reduce the run-time of a join query from hours to minutes.
- Again, all selected rows from secondary dials are accessed through an index. If an efficient index cannot be found, a temporary index is created. Some join queries build temporary indexes over secondary dials even when an index exists for all of the join keys. Because efficiency is very important for secondary dials of longer running queries, the query optimizer may choose to build a temporary index which contains only entries which pass the local row selection for that dial. This preprocessing of row selection allows the database manager to process row selection in one pass instead of each time rows are matched for a dial.

| **Arrival sequence nested loop join implementation using table scan:**

| During the join, DB2 Universal Database for iSeries:

- | 1. Accesses the first primary table row selected by the predicates local to the primary table.
- | 2. Scans the secondary to locate the first row that satisfies the join condition for the first secondary table using the table scan to match the join condition or local row selection columns of the secondary table.
- | 3. Determines if the row is selected by applying any remaining selection local to the first secondary dial. If the secondary dial row is not selected then the next row that satisfies the join condition is located. Steps 1 through 3 are repeated until a row that satisfies both the join condition and any remaining selection is selected from all secondary tables.
- | 4. Returns the result join row.
- | 5. Processes the last secondary table again to find the next row that satisfies the join condition in that dial. During this processing, when no more rows that satisfy the join condition can be selected, the processing backs up to the logical previous dial and attempts to read the next row that satisfies its join condition.
- | 6. Ends processing when all selected rows from the primary table are processed.

Hash join

The hash join method is similar to nested loop join. Instead of using indexes to locate the matching rows in a secondary table, however, a hash temporary result table is created that contains all of the rows selected by local selection against the table. The structure of the hash table is such that rows with the same join value are loaded into the same hash table partition (clustered). The location of the rows for any given join value can be found by applying a hashing function to the join value.

Advantages of hash joins over nested loop joins

Hash join has several advantages over nested loop join:

- The structure of a hash temporary result table is simpler than that of an index, so less CPU processing is required to build and probe a hash table.

- The rows in the hash result table contain all of the data required by the query so there is no need to access the dataspace of the table with random I/O when probing the hash table.
- Like join values are clustered, so all matching rows for a given join value can usually be accessed with a single I/O request.
- The hash temporary result table can be built using SMP parallelism.
- Unlike indexes, entries in hash tables are not updated to reflect changes of column values in the underlying table. The existence of a hash table does not affect the processing cost of other updating jobs in the server.

Queries that cannot use hash join

Hash join cannot be used for queries that:

- Perform subqueries unless all subqueries in the query can be transformed to inner joins.
- Perform a UNION or UNION ALL.
- Perform left outer or exception join.
- Use a DDS created join logical file.
- Require live access to the data as specified by the *NO or *YES parameter values for the ALWCOPYDATA precompiler parameter. Hash join is used only for queries running with ALWCOPYDATA(*OPTIMIZE). This parameter can be specified either on precompiler commands, the STRSQL CL command, or the OPNQRYF CL command. The Client Access/400 ODBC driver and Query Management driver always uses this mode.
- Hash join can be used with OPTIMIZE(*YES) if a temporary result is required to run the query.
- Hash join cannot be used for queries involving physical files or tables that have read triggers.
- Require that the cursor position be restored as the result of the SQL ROLLBACK HOLD statement or the ROLLBACK CL command. For SQL applications using commitment control level other than *NONE, this requires that *ALLREAD be specified as the value for the ALWBLK precompiler parameter.

Hash join and parallel processing

The query attribute DEGREE, which can be changed by using the Change Query attribute CL command (CHGQRYA), does not enable or disable the optimizer from choosing to use hash join. However, hash join queries can use SMP parallelism if the query attribute DEGREE is set to either *OPTIMIZE, *MAX, or *NBRTASKS.

Types of queries that can effectively use hash join

Hash join is used in many of the same cases where a temporary index would have been built. Join queries which are most likely to be implemented using hash join are those where either:

- All rows in the various tables of the join are involved in producing result rows.
- Significant non-join selection is specified for the tables of the join which reduces the number of rows in the tables that are involved with the join result.

Example of hash join that processes all rows

The following is an example of a join query that would process all of the rows from the queried tables:

```
SELECT *
  FROM EMPLOYEE, EMP_ACT
 WHERE EMPLOYEE.EMPNO = EMP_ACT.EMPNO
OPTIMIZE FOR 99999999 ROWS
```

OPNQRYF example :

```
OPNQRYF FILE((EMPLOYEE EMP_ACT)) FORMAT(FORMAT1)
  JFLD((1/EMPNO 2/EMPNO *EQ))
  ALWCPYDTA(*OPTIMIZE)
```

This query is implemented using the following steps:

1. A temporary hash table is built over table EMP_ACT with a key of EMPNO. This occurs when the query is opened.
2. For each row retrieved from the EMPLOYEE table, the temporary hash table will be probed for any matching join values.
3. For each matching row found, a result row is returned.

The messages created by the PRTSQLINF CL command to describe this hash join query in an SQL program would appear as follows:

```
SQL402A Hashing algorithm used to process join.
SQL402B Table EMPLOYEE used in hash join step 1.
SQL402B Table EMP_ACT used in hash join step 2.
```

Example of hash join on query that is limited by local selection

The following is an example of a join query that would have the queried tables of the join queried significantly reduced by local selection:

```
SELECT EMPNO, LASTNAME, DEPTNAME
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.WORKDEPT = DEPARTMENT.DEPTNO
AND EMPLOYEE.HIREDATE BETWEEN 1996-01-30 AND 1995-01-30
AND DEPARTMENT.DEPTNO IN ('A00', 'D01', 'D11', 'D21', 'E11')
OPTIMIZE FOR 99999999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE DEPARTMENT))
  FORMAT(FORMAT2)
  QRYSLT('1/HIREDATE *EQ %RANGE(''1996-01-30'' ''1995-01-30'')
  *AND 2/DEPTNO *EQ %VALUES(''A00'' ''D01'' ''D11'' ''D21''
  ''E11'')
  JFLD((1/WORKDEPT 2/DEPTNO *EQ))
  ALWCPYDTA(*OPTIMIZE)
```

This query is implemented using the following steps:

1. A temporary hash table is built over table DEPARTMENT with key values of DEPTNO containing rows matching the selection predicate, DEPTNO IN ('A00', 'D01', 'D11', 'D21', 'E11'). This occurs when the query is opened.
2. For each row retrieved from the EMPLOYEE table matching the selection predicate, HIREDATE BETWEEN 1996-01-30 and 1995-01-30, the temporary hash table will be probed for the matching join values.
3. For each matching row found, a result row is returned.

The messages created by the PRTSQLINF CL command to describe this hash join query in an SQL program would appear as follows:

```
SQL402A Hashing algorithm used to process join.
SQL402B Table EMPLOYEE used in hash join step 1.
SQL402B Table DEPARTMENT used in hash join step 2.
```

Example of hash join on query where ordering, grouping, non-equal selection, or result columns are selected

When ordering, grouping, non-equal selection specified with operands derived from columns of different tables, or result columns are derived from columns of different tables, the hash join processing will be done and the result rows of the join will be written to a temporary table. Then, as a second step, the query will be completed using the temporary table.

The following is an example of a join query with selection specified with operands derived from columns of different tables:

```
SELECT EMPNO, LASTNAME, DEPTNAME
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.WORKDEPT = DEPARTMENT.DEPTNO
AND EMPLOYEE.EMPNO > DEPARTMENT.MGRNO
OPTIMIZE FOR 99999999 ROWS
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE DEPARTMENT)
FORMAT(FORMAT2)
JFLD((1/WORKDEPT 2/DEPTNO *EQ) (1/EMPNO 2/MGRNO
*GT))
```

This query is implemented using the following steps:

1. A temporary hash table is built over table DEPARTMENT with a key of DEPTNO. This occurs when the query is opened.
2. For each row retrieved from the EMPLOYEE table, the temporary hash table will be probed for the matching join values.
3. For each matching row found, a result row is written to a temporary table.
4. After all of the join result rows are written to the temporary table, rows that are selected by EMPNO > MGRNO are read from the temporary table and returned to the application.

The messages created by the PRTSQLINF CL command to describe this hash join query in an SQL program would appear as follows:

```
SQL402A Hashing algorithm used to process join.
SQL402B Table EMPLOYEE used in hash join step 1.
SQL402B Table DEPARTMENT used in hash join step 2.
SQL402C Temporary result table created for hash join query.
```

Join optimization algorithm

The query optimizer must determine the join columns, join operators, local row selection, index usage, and dial ordering for a join query.

The join columns and join operators depend on the:

- Join column specifications of the query
- Join order
- Interaction of join columns with other row selection
- Index used.

Join specifications which are not implemented for the dial are either deferred until they can be processed in a later dial or, if an inner join was being performed for this dial, processed as row selection.

For a given dial, the only join specifications which are usable as join columns for that dial are those being joined to a *previous* dial. For example, for the second dial the only join specifications that can be used to satisfy the join condition are join specifications which reference columns in the primary dial. Likewise, the third dial can only use join specifications which reference columns in the primary and the second dials and so on. Join specifications which reference later dials are deferred until the referenced dial is processed.

For any given dial, only one type of join operator is normally implemented. For example, if one inner join specification has a join operator of '=' and the other has a join operator of '>', the optimizer attempts to implement the join with the '=' operator. The '>' join specification is processed as row selection after a matching row for the '=' specification is found. In addition, multiple join specifications that use the same operator are implemented together.

Note: For OPNQRYF, only one type of join operator is allowed for either a left outer or an exception join. That is, the join operator for all join conditions must be the same.

When looking for an existing index to access a secondary dial, the query optimizer looks at the left-most key columns of the index. For a given dial and index, the join specifications which use the left-most key columns can be used. For example:

```
DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE, EMP_ACT
WHERE EMPLOYEE.EMPNO = EMP_ACT.EMPNO
AND EMPLOYEE.HIREDATE = EMP_ACT.EMSTDATE
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
| OPNQRYF FILE((EMPLOYEE, EMP_ACT)) FORMAT(FORMAT1)
| JFLD((1/EMPNO 2/EMPNO *EQ)(1/HIREDATE 2/EMSTDATE
| *EQ))
```

For the index over EMP_ACT with key columns EMPNO, PROJNO, and EMSTDATE, the join operation is performed only on column EMPNO. After the join is performed, index scan-key selection is done using column EMSTDATE.

The query optimizer also uses local row selection when choosing the best use of the index for the secondary dial. If the previous example had been expressed with a local predicate as:

```
DECLARE BROWSE2 CURSOR FOR
SELECT * FROM EMPLOYEE, EMP_ACT
WHERE EMPLOYEE.EMPNO = EMP_ACT.EMPNO
AND EMPLOYEE.HIREDATE = EMP_ACT.EMSTDATE
AND EMP_ACT.PROJNO = '123456'
OPTIMIZE FOR 99999 ROWS
```

OPNQRYF example:

```
| OPNQRYF FILE((EMPLOYEE, EMP_ACT)) FORMAT(FORMAT2)
| QRYSLT('2/PROJNO *EQ '123456''')
| JFLD((1/EMPNO 2/EMPNO *EQ)(1/HIREDATE 2/EMSTDATE
| *EQ))
```

the index with key columns EMPNO, PROJNO, and EMSTDATE are fully utilized by combining join and selection into one operation against all three key columns.

When creating a temporary index, the left-most key columns are the usable join columns in that dial position. All local row selection for that dial is processed when selecting entries for inclusion into the temporary index. A temporary index is similar to the index created for a select/omit keyed logical file. The temporary index for the previous example would have key columns of EMPNO and EMSTDATE.

```
| Since the OS/400® query optimizer attempts a combination of join and local row selection when
| determining access path usage, it is possible to achieve almost all of the same advantages of a temporary
| index by use of an existing index. In the above example, using either implementation, an existing index
| may be used or a temporary index may be created. A temporary index would have been built with the local
| row selection on PROJNO applied during the index's creation; the temporary index would have key
| columns of EMPNO and EMSTDATE (to match the join selection). If, instead, an existing index was used
| with key columns of EMPNO, PROJNO, EMSTDATE (or PROJNO, EMP_ACT, EMSTDATE or
| EMSTDATE, PROJNO, EMP_ACT or ...) the local row selection could be applied at the same time as the
```

- | join selection (rather than prior to the join selection, as happens when the temporary index is created, or
- | after the join selection, as happens when only the first key column of the index matches the join column).

The implementation using the existing index is more likely to provide faster performance because join and selection processing are combined without the overhead of building a temporary index. However, the use of the existing index may have just slightly slower I/O processing than the temporary index because the local selection is run many times rather than once. In general, it is a good idea to have existing indexes available with key columns for the combination of join columns and columns using equal selection as the left-most keys.

Join order optimization

The join order is fixed if any join logical files are referenced. The join order is also fixed if the OPNQRYF JORDER(*FILE) parameter is specified or the query options file (QAQQINI) FORCE_JOIN_ORDER parameter is *YES. Otherwise, the following join ordering algorithm is used to determine the order of the tables:

1. Determine an access method for each individual table as candidates for the primary dial.
2. Estimate the number of rows returned for each table based on local row selection.
If the join query with row ordering or group by processing is being processed in one step, then the table with the ordering or grouping columns is the primary table.
3. Determine an access method, cost, and expected number of rows returned for each join combination of candidate tables as primary and first secondary tables.

The join order combinations estimated for a four table inner join would be:

1-2 2-1 1-3 3-1 1-4 4-1 2-3 3-2 2-4 4-2 3-4 4-3

4. Choose the combination with the lowest join cost.
If the cost is nearly the same, then choose the combination which selects the fewest rows.
5. Determine the cost, access method, and expected number of rows for each remaining table joined to the previous secondary table.
6. Select an access method for each table that has the lowest cost for that table.
7. Choose the secondary table with the lowest join cost.
If the cost is nearly the same, choose the combination which selects the fewest rows.
8. Repeat steps 4 through 7 until the lowest cost join order is determined.

Note: After dial 32, the optimizer uses a different method to determine file join order, which may not be the lowest cost.

When a query contains a left or right outer join or a right exception join, the join order is not fixed. However, all from-columns of the ON clause must occur from dials previous to the left or right outer or exception join. For example:

```
FROM A INNER JOIN B ON A.C1=B.C1
LEFT OUTER JOIN C ON B. C2=C.C2
```

The allowable join order combinations for this query would be:

1-2-3, 2-1-3, or 2-3-1

Right outer or right exception joins are implemented as left outer and left exception, respectively with files flipped. For example:

```
FROM A RIGHT OUTER JOIN B ON A.C1=B.C1
```

is implemented as B LEFT OUTER JOIN A ON B.C1=A.C1. The only allowed join order is 2-1.

When a join logical file is referenced or the join order is forced to the specified table order, the query optimizer loops through all of the dials in the order specified, and determines the lowest cost access methods.

Cost estimation and index selection for join secondary dials

In step 3 on page 41 and in step 5 on page 41, the query optimizer has to estimate a cost and choose an access method for a given dial combination. The choices made are similar to those for row selection except that an index must be used.

As the query optimizer compares the various possible access choices, it must assign a numeric cost value to each candidate and use that value to determine the implementation which consumes the least amount of processing time. This costing value is a combination of CPU and I/O time and is based on the following assumptions:

- Table pages and index pages must be retrieved from auxiliary storage. For example, the query optimizer is not aware that an entire table may be loaded into active memory as the result of a SETOBJACC CL command. Usage of this command may significantly improve the performance of a query, but the query optimizer does not change the query implementation to take advantage of the memory resident state of the table.
- The query is the only process running on the server. No allowance is given for server CPU utilization or I/O waits which occur because of other processes using the same resources. CPU related costs are scaled to the relative processing speed of the server running the query.
- The values in a column are uniformly distributed across the table. For example, if 10% of the rows in a table have the same value, then it is assumed that every tenth row in the table contains that value.
- The values in a column are independent from the values in any other columns in a row. For example, if a column named A has a value of 1 in 50% of the rows in a table and a column named B has a value of 2 in 50% of the rows, then it is expected that a query which selects rows where A = 1, and B = 2 selects 25% of the rows in the table.

The main factors of the join cost calculations for secondary dials are the number of rows selected in all previous dials and the number of rows which match, on average, each of the rows selected from previous dials. Both of these factors can be derived by estimating the number of matching rows for a given dial.

When the join operator is something other than equal, the expected number of matching rows is based on the following default filter factors:

- 33% for less-than, greater-than, less-than-equal-to, or greater-than-equal-to
- 90% for not equal
- 25% for BETWEEN range (OPNQRYF %RANGE)
- 10% for each IN list value (OPNQRYF %VALUES)

For example, when the join operator is less-than, the expected number of matching rows is $.33 * (\text{number of rows in the dial})$. If no join specifications are active for the current dial, the cartesian product is assumed to be the operator. For cartesian products, the number of matching rows is every row in the dial, unless local row selection can be applied to the index.

When the join operator is equal, the expected number of rows is the average number of duplicate rows for a given value.

The iSeries performs index maintenance (insertion and deletion of key values in an index) and maintains a running count of the number of unique values for the given key columns in the index. These statistics are bound with the index object and are always maintained. The query optimizer uses these statistics when it is optimizing a query. Maintaining these statistics adds no measurable amount of overhead to index maintenance. This statistical information is only available for indexes which:

- Contain no varying length character keys.

Note: If you have varying length character columns used as join columns, you can create an index which maps the varying length character column to a fixed character key using the CTRL F CL command. An index that contains fixed length character keys defined over varying length data supplies average number of duplicate values statistics.

- Were created or rebuilt on an iSeries server on which Version 2 Release 3 or a later version is installed.

Note: The query optimizer can use indexes created on earlier versions of OS/400 to estimate if the join key values have a high or low average number of duplicate values. If the index is defined with only the join keys, the estimate is done based on the size of the index. In many cases, additional keys in the index cause matching row estimates through that index to not be valid. The performance of some join queries may be improved by rebuilding these indexes.

Average number of duplicate values statistics are maintained only for the first 4 left-most keys of the index. For queries which specify more than 4 join columns, it might be beneficial to create multiple additional indexes so that an index can be found with *average number of duplicate values* statistics available within the 4 left-most key columns. This is particularly important if some of the join columns are somewhat unique (*low average number of duplicate values*).

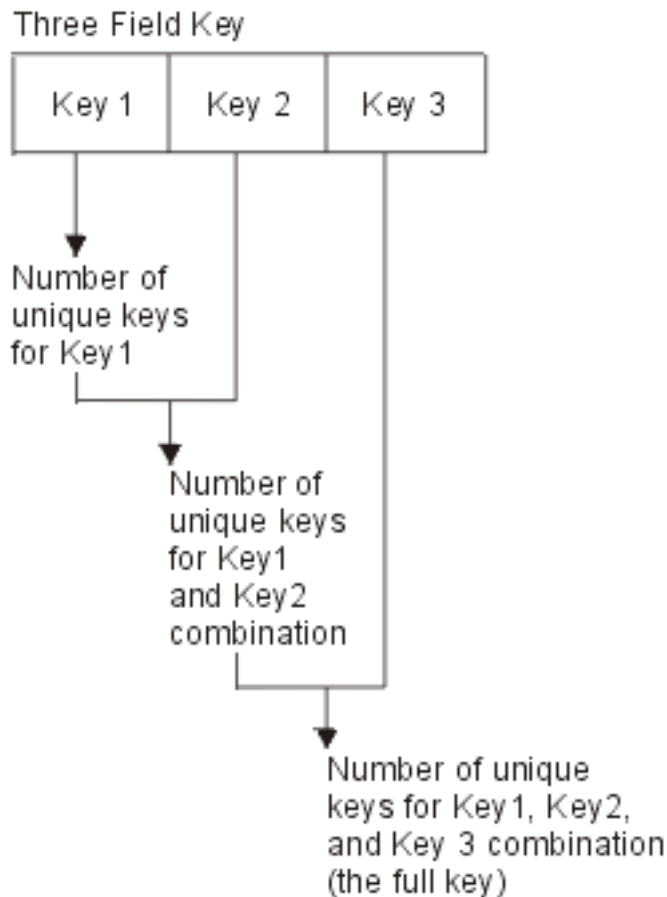


Figure 1. Average number of duplicate values of a 3 key index

These statistics are maintained as part of index rebuild and creation.

Using the average number of duplicate values for equal joins or the default filter value for the other join operators, we now have the number of matching rows. The following formula is used to compute the number of join rows from previous dials.

$$NPREV = R_p * M_2 * FF_2 * \dots * M_n * FF_n \dots$$

NPREV

The number of join rows from all previous dials.

R_p The number of rows selected from the primary dial.

M₂ The number of matching rows for dial 2.

FF₂ Filtering reduction factor for predicates local to dial 2 that are not already applied using M₂ above.

M_n The number of matching rows for dial n.

FF_n Filtering reduction factor for predicates local to dial n that are not already applied using M_n above.

Note: Multiply the pair of matching rows (M_n) and filter reduction filter factors (FF_n) for each secondary dial preceding the current dial.

Now that it has calculated the number of join rows from previous dials, the optimizer is ready to generate a cost for the access method.

Temporary index or hash temporary result table from table: The first access method choice analyzed by the query optimizer is building a *temporary index or hash temporary result table from the table*. The basic formula for costing access of a join secondary dial through a temporary index built from the table or hash table follows:

$$JSCOST = CRTDSI + NPREV * ((MATCH * FF * KeyAccess) + (MATCH * FF * FCost)) * FirstIO$$

JSCOST

Join Secondary cost

CRTDSI

Cost to build the temporary index or a hash temporary result table

NPREV

The number of join rows from all previous dials

MATCH

The number of matching rows (usually average duplicates)

KeyAccess

The cost to access a key in an index or a hash table

FF

The filtering factor for local predicates of this dial (excluding selection performed on earlier dials because of transitive closure)

FCost

The cost to access a row from the table

FirstIO

A reduction ratio to reduce the non-startup cost because of an optimization goal to optimize for the first buffer retrieval. For more information, see “Cost estimation for queries” on page 32.

This secondary dial access method is used if no usable index is found or if the temporary index or hash table performs better than any existing index. This method can be better than using any existing index because the row selection is completed when the index or hash table is created if any of the following are true:

- The number of matches (MATCH) is high.
- The number of join rows from all previous dials (NPREV) is high.

- There is some filtering reduction (FF < 100%).

Temporary index or hash table from index: The basic cost formula for this access method choice is the same as that of using a temporary index or hash table built from a table, with one exception. The cost to build the temporary index, CRTDSI, is calculated to include the selection of the rows through an existing index. This access method is used for join secondary dial access for the same reason. However, the creation from an index might be less costly.

Use an existing index: The final access method is to use an existing index. The basic formula for costing access of a join secondary dial through an existing index is:

$$\text{JSCOST} = \text{NPREV} * ((\text{MATCH} * \text{KeyAccess}) + (\text{MATCH} * \text{FCost})) * \text{FirstIO}$$

JSCOST

Join Secondary cost

NPREV

The number of join rows from all previous dials

MATCH

The number of matching keys which will be found in this index (usually average duplicates)

KeyAccess

The cost to access a key in an index

FCost The cost to access a row from the table

FirstIO

A reduction ratio to reduce the non-startup cost because of an optimization goal to optimize for the first buffer retrieval. For more information, see "Cost estimation for queries" on page 32.

If I/O optimization is used (that is, OPNQRYF OPTIMIZE(*FIRSTIO)), this is a likely access method because the entire cost is reduced. Also, if the number of join rows from all previous dials (NPREV), and the number of matching keys (MATCH) is low, this may be the most efficient method.

The query optimizer considers using an index which only has a subset of the join columns as the left-most leading keys when:

- It is able to determine from the average number of duplicate values statistics that the average number of rows with duplicate values is quite low.
- The number of rows being selected from the previous dials is small.

Predicates generated through transitive closure

For join queries, the query optimizer may do some special processing to generate additional selection. When the set of predicates that belong to a query logically infer extra predicates, the query optimizer generates additional predicates. The purpose is to provide more information during join optimization.

Example of predicates being added because of transitive closure:

```
SELECT * FROM EMPLOYEE, EMP_ACT
WHERE EMPLOYEE.EMPNO = EMP_ACT.EMPNO
AND EMPLOYEE.EMPNO = '000010'
```

The optimizer will modify the query to be:

```
SELECT * FROM EMPLOYEE, EMP_ACT
WHERE EMPLOYEE.EMPNO = EMP_ACT.EMPNO
AND EMPLOYEE.EMPNO = '000010'
AND EMP_ACT.EMPNO = '000010'
```

OPNQRYF example:

```
OPNQRYF FILE((EMPLOYEE EMP_ACT)) FORMAT(FORMAT1)
  QRYSLT('1/EMPNO *EQ ''000010''')
  JFLD((1/EMPNO 2/EMPNO *EQ))
```

The optimizer will modify the query to be:

```
OPNQRYF FILE((EMPLOYEE EMP_ACT)) FORMAT(FORMAT1)
  QRYSLT('1/EMPNO *EQ ''000010'' *AND
  2/EMPNO *EQ ''000010''')
  JFLD((1/EMPNO 2/EMPNO *EQ))
```

The following rules determine which predicates are added to other join dials:

- The dials affected must have join operators of equal.
- The predicate is **isolatable**, which means that a false condition from this predicate would omit the row.
- One operand of the predicate is an equal join column and the other is a constant or host variable.
- The predicate operator is not LIKE or IN (OPNQRYF %WLDCRD, %VALUES, or *CT).
- The predicate is not connected to other predicates by OR.
- The join type for the dial is an inner join.

The query optimizer generates a new predicate, whether or not a predicate already exists in the WHERE clause (OPNQRYF QRYSLT parameter).

Some predicates are redundant. This occurs when a previous evaluation of other predicates in the query already determines the result that predicate provides. Redundant predicates can be specified by you or generated by the query optimizer during predicate manipulation. Redundant predicates with predicate operators of =, >, >=, <, <=, or BETWEEN (OPNQRYF *EQ, *GT, *GE, *LT, *LE, or %RANGE) are merged into a single predicate to reflect the most selective range.

Multiple join types for a query

Even though multiple join types (inner, left outer, right outer, left exception, and right exception) can be specified in the query using the JOIN syntax, the iSeries Licensed Internal Code can only support one join type of inner, left outer, or left exception join type for the entire query. This requires the optimizer to determine what the overall join type for the query should be and to reorder files to achieve the correct semantics.

Note: This section does not apply to OPNQRYF.

The optimizer will evaluate the join criteria along with any row selection that may be specified in order to determine the join type for each dial and for the entire query. Once this information is known the optimizer will generate additional selection using the relative row number of the tables to simulate the different types of joins that may occur within the query.

Since null values are returned for any unmatched rows for either a left outer or an exception join, any isolatable selection specified for that dial, including any additional join criteria that may be specified in the WHERE clause, will cause all of the unmatched rows to be eliminated (unless the selection is for an IS NULL predicate). This will cause the join type for that dial to be changed to an inner join (or an exception join) if the IS NULL predicate was specified.

In the following example a left outer join is specified between the tables EMPLOYEE and DEPARTMENT. In the WHERE clause there are two selection predicates that also apply to the DEPARTMENT table.

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
FROM CORPDATA.EMPLOYEE XXX LEFT OUTER JOIN CORPDATA.DEPARTMENT YYY
  ON XXX.WORKDEPT = YYY.DEPTNO
  LEFT OUTER JOIN CORPDATA.PROJECT ZZZ
  ON XXX.EMPNO = ZZZ.RESPEMP
WHERE XXX.EMPNO = YYY.MGRNO AND
  YYY.DEPTNO IN ('A00', 'D01', 'D11', 'D21', 'E11')
```

The first selection predicate, XXX.EMPNO = YYY.MGRNO, is an additional join condition that will be added to the join criteria and evaluated as an "inner join" join condition. The second is an isolatable selection predicate that will eliminate any unmatched rows. Either one of these selection predicates will cause the join type for the DEPARTMENT table to be changed from a left outer join to an inner join.

Even though the join between the EMPLOYEE and the DEPARTMENT table was changed to an inner join the entire query will still need to remain a left outer join to satisfy the join condition for the PROJECT table.

Note: Care must be taken when specifying multiple join types since they are supported by appending selection to the query for any unmatched rows. This means that the number of resulting rows that satisfy the join criteria can become quite large before any selection is applied that will either select or omit the unmatched rows based on that individual dial's join type.

For more information on how to use the JOIN syntax see either Joining Data from More Than One Table in the SQL Programming Concepts book or the SQL Reference book.

Sources of join query performance problems

The optimization algorithms described above benefit most join queries, but the performance of a few queries may be degraded. This occurs when:

- An index is not available which provides average number of duplicate values statistics for the potential join columns.

Note: "Cost estimation and index selection for join secondary dials" on page 42 provides suggestions on how to avoid the restrictions about indexes statistics or create additional indexes over the potential join columns if they do not exist.

- The query optimizer uses default filter factors to estimate the number of rows being selected when applying local selection to the table because indexes do not exist over the selection columns.

Creating indexes over the selection columns allows the query optimizer to make a more accurate filtering estimate by using key range estimates.

- The particular values selected for the join columns yield a significantly greater number of matching rows than the average number of duplicate values for all values of the join columns in the table (i.e. the data is not uniformly distributed).

Use DDS to build a logical file with an index with select/omit specifications matching the local row selection. This provides the query optimizer with a more accurate estimate of the number of matching rows for the keys which are selected.

Note: The optimizer can better determine from the select/omit index that the data is not uniformly distributed.

- The query optimizer makes the wrong assumption about the number of rows which will be retrieved from the answer set.

For SQL programs, specifying the precompile option ALWCPYDTA(*YES) makes it more likely that the queries in that program will use an existing index. Likewise, specifying ALWCPYDTA(*OPTIMIZE) makes it more likely that the queries in that program will create a temporary index. The SQL clause OPTIMIZE FOR n ROWS can also be used to influence the query optimizer.

For the OPNQRYF command, the wrong performance option for the OPTIMIZE keyword may have been specified. Specify *FIRSTIO to make the use of an existing index more likely. Specify *ALLIO to make the creation of a temporary index more likely.

Tips for improving the performance of join queries

If you are looking at a join query which is performing poorly or you are about to create a new application which uses join queries, the following checklist may be useful.

Table 2. Checklist for Creating an Application that Uses Join Queries

What to Do	How It Helps
Check the database design. Make sure that there are indexes available over all of the join columns and/or row selection columns. If using CRTLF, make sure that the index is not shared.	This gives the query optimizer a better opportunity to select an efficient access method because it can determine the average number of duplicate values. Many queries may be able to use the existing index to implement the query and avoid the cost of creating a temporary index.
Check the query to see whether some complex predicates should be added to other dials to allow the optimizer to get a better idea of the selectivity of each dial.	Since the query optimizer does not add predicates for predicates connected by OR or non-isolatable predicates, or predicate operators of LIKE or IN, modifying the query by adding these predicates may help.
Create an index which includes Select/Omit specifications which match that of the query using CRTLF CL command.	This step helps if the statistical characteristics are not uniform for the entire table. For example, if there is one value which has a high duplication factor and the rest of the column values are unique, then a select/omit index allows the optimizer to skew the distribution of values for that key and make the right optimization for the selected values.
Specify ALWCPYDTA(*OPTIMIZE) or ALWCPYDTA(*YES)	<p>If the query is creating a temporary index, and you feel that the processing time would be better if the optimizer only used the existing index, specify ALWCPYDTA(*YES).</p> <p>If the query is not creating a temporary index, and you feel that the processing time would be better if a temporary index was created, specify ALWCPYDTA(*OPTIMIZE).</p> <p>Alternatively, specify the OPTIMIZE FOR n ROWS to inform the optimizer of the application has intention to read every resulting row. To do this set n to a large number. You could also set n to a small number before ending the query.</p>
For OPNQRYF, specify OPTIMIZE(*FIRSTIO) or OPTIMIZE(*ALLIO)	If the query is creating a temporary index and you feel that the processing time would be better if it would only use the existing index, then specify OPTIMIZE(*FIRSTIO). If the query is not creating a temporary index and you feel that the processing time would be better if a temporary index was created then specify OPTIMIZE(*ALLIO).

Table 2. Checklist for Creating an Application that Uses Join Queries (continued)

What to Do	How It Helps
<p>Use a join logical file or use the query options file (QAQQINI) FORCE_JOIN_ORDER parameter of *YES. OPNQRYF users can specify JORDER(*FILE).</p>	<p>A join in which one table is joined with all secondary tables consecutively is sometimes called a star join. In the case of a star join where all secondary join predicates contain a column reference to a particular table, there may be performance advantages if that table is placed in join position one. In Example A, all tables are joined to table EMPLOYEE. The query optimizer can freely determine the join order. The query should be changed to force EMPLOYEE into join position one by using the query options file (QAQQINI) FORCE_JOIN_ORDER parameter of *YES or OPNQRYF JORDER(*FILE) as shown in example B. Note that in these examples the join type is a join with no default values returned (this is an inner join.). The reason for forcing the table into the first position is to avoid random I/O processing. If EMPLOYEE is not in join position one, every row in EMPLOYEE could be examined repeatedly during the join process. If EMPLOYEE is fairly large, considerable random I/O processing occurs resulting in poor performance. By forcing EMPLOYEE to the first position, random I/O processing is minimized.</p> <p>Example A: Star join query</p> <pre> DECLARE C1 CURSOR FOR SELECT * FROM DEPARTMENT, EMP_ACT, EMPLOYEE, PROJECT WHERE DEPARTMENT.DEPTNO=EMPLOYEE.WORKDEPT AND EMP_ACT.EMPNO=EMPLOYEE.EMPNO AND EMPLOYEE.WORKDEPT=PROJECT.DEPTNO </pre> <p>Example B: Star join query with order forced via FORCE_JOIN_ORDER</p> <pre> DECLARE C1 CURSOR FOR SELECT * FROM EMPLOYEE, DEPARTMENT, EMP_ACT, PROJECT WHERE DEPARTMENT.DEPTNO=EMPLOYEE.WORKDEPT AND EMP_ACT.EMPNO=EMPLOYEE.EMPNO AND EMPLOYEE.WORKDEPT=PROJECT.DEPTNO </pre> <p>Example A: Star join query (OPNQRYF)</p> <pre> OPNQRYF FILE((DEPARTMENT EMP_ACT EMPLOYEE PROJECT)) FORMAT(FORMAT1) JFLD((1/DEPTNO 3/WORKDEPT *EQ) (2/EMPNO 3/EMPNO *EQ) (3/WORKDEPT 4/DEPTNO *EQ)) </pre> <p>Example B: Star join query (OPNQRYF) with JORDER(*FILE) parameter</p> <pre> OPNQRYF FILE((EMPLOYEE DEPARTMENT EMP_ACT PROJECT)) FORMAT(FORMAT1) JFLD((2/DEPTNO 1/WORKDEPT *EQ) (3/EMPNO 1/EMPNO *EQ) (1/WORKDEPT 4/DEPTNO *EQ)) JORDER(*FILE) </pre> <p>Note: Specifying columns from EMPLOYEE in the ORDER BY clause (OPNQRYF KEYFLD parameter) may also have the effect of placing EMPLOYEE in join position 1. This allows the query optimizer to choose the best order for the remaining tables.</p>
<p>Specify ALWCOPYDTA(*OPTIMIZE) to allow the query optimizer to use a sort routine.</p>	<p>In the cases where ordering is specified and all key columns are from a single dial, this allows the query optimizer to consider all possible join orders.</p>
<p>Specify join predicates to prevent all of the rows from one table from being joined to every row in the other table.</p>	<p>This improves performance by reducing the join fan-out. Every secondary table should have at least one join predicate that references one of its columns as a 'join-to' column.</p>

Tips for improving performance when selecting data from more than two tables

If the select-statement you are considering accesses two or more tables, all the recommendations suggested in Chapter 5, “Using indexes to speed access to large tables” on page 103 apply. The following suggestion is directed specifically to select-statements that access several tables. For joins that involve more than two tables, you might want to provide redundant information about the join columns. If you give the optimizer extra information to work with when requesting a join. It can determine the best way to do the join. The additional information might seem redundant, but is helpful to the optimizer. For example, instead of coding:

```
EXEC SQL
  DECLARE EMPACTDATA CURSOR FOR
  SELECT LASTNAME, DEPTNAME, PROJNO, ACTNO
     FROM CORPDATA.DEPARTMENT, CORPDATA.EMPLOYEE,
          CORPDATA.EMP_ACT
     WHERE DEPARTMENT.MGRNO = EMPLOYEE.EMPNO
          AND EMPLOYEE.EMPNO = EMP_ACT.EMPNO
END-EXEC.
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE(CORPDATA/DEPARTMENT CORPDATA/EMPLOYEE CORPDATA/EMP_ACT)
  FORMAT(FORMAT1)
  JFLD((1/MGRNO 2/EMPNO *EQ) (2/EMPNO 3/EMP_ACT *EQ))
```

Provide the optimizer with a little more data and code:

```
EXEC SQL
  DECLARE EMPACTDATA CURSOR FOR
  SELECT LASTNAME, DEPTNAME, PROJNO, ACTNO
     FROM CORPDATA.DEPARTMENT, CORPDATA.EMPLOYEE,
          CORPDATA.EMP_ACT
     WHERE DEPARTMENT.MGRNO = EMPLOYEE.EMPNO
          AND EMPLOYEE.EMPNO = EMP_ACT.EMPNO
          AND DEPARTMENT.MGRNO = EMP_ACT.EMPNO
END-EXEC.
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE(CORPDATA/DEPARTMENT CORPDATA/EMPLOYEE CORPDATA/EMP_ACT)
  FORMAT(FORMAT1)
  JFLD((1/MGRNO 2/EMPNO *EQ) (2/EMPNO 3/EMP_ACT *EQ)
        (1/MGRNO 3/EMPNO *EQ))
```

Grouping optimization

This section describes how DB2 Universal Database for iSeries implements grouping techniques and how optimization choices are made by the query optimizer. The query optimizer has two choices for implementing grouping: the hash implementation or the index implementation.

Grouping hash implementation

This technique uses the base hash access method to perform grouping or summarization of the selected table rows. For each selected row, the specified grouping value is run through the hash function. The computed hash value and grouping value are used to quickly find the entry in the hash table corresponding to the grouping value. If the current grouping value already has a row in the hash table, the hash table entry is retrieved and summarized (updated) with the current table row values based on the requested grouping column operations (such as SUM or COUNT). If a hash table entry is not found for the current grouping value, a new entry is inserted into the hash table and initialized with the current grouping value.

The time required to receive the first group result for this implementation will most likely be longer than other grouping implementations because the hash table must be built and populated first. Once the hash

table is completely populated, the database manager uses the table to start returning the grouping results. Before returning any results, the database manager must apply any specified grouping selection criteria or ordering to the summary entries in the hash table.

Where the grouping hash method is most effective

The grouping hash method is most effective when the consolidation ratio is high. The **consolidation ratio** is the ratio of the selected table rows to the computed grouping results. If every database table row has its own unique grouping value, then the hash table will become too large. This in turn will slow down the hashing access method.

The optimizer estimates the consolidation ratio by first determining the number of unique values in the specified grouping columns (that is, the expected number of groups in the database table). The optimizer then examines the total number of rows in the table and the specified selection criteria and uses the result of this examination to estimate the consolidation ratio.

Indexes over the grouping columns can help make the optimizer's ratio estimate more accurate. Indexes improve the accuracy because they contain statistics that include the average number of duplicate values for the key columns.

The optimizer also uses the expected number of groups estimate to compute the number of partitions in the hash table. As mentioned earlier, the hashing access method is more effective when the hash table is well-balanced. The number of hash table partitions directly affects how entries are distributed across the hash table and the uniformity of this distribution.

The hash function performs better when the grouping values consist of columns that have non-numeric data types, with the exception of the integer (binary) data type. In addition, specifying grouping value columns that are not associated with the variable length and null column attributes allows the hash function to perform more effectively.

Index grouping implementation

This implementation utilizes the index scan-key selection or index scan-key positioning access methods to perform the grouping. An index is required that contains all of the grouping columns as contiguous leftmost key columns. The database manager accesses the individual groups through the index and performs the requested summary functions.

Since the index, by definition, already has all of the key values grouped together, the first group result can be returned in less time than the hashing method. This is because of the temporary result that is required for the hashing method. This implementation can be beneficial if an application does not need to retrieve all of the group results or if an index already exists that matches the grouping columns.

When the grouping is implemented with an index and a permanent index does not already exist that satisfies grouping columns, a temporary index is created. The grouping columns specified within the query are used as the key columns for this index.

Optimizing grouping by eliminating grouping columns

All of the grouping columns are evaluated to determine if they can be removed from the list of grouping columns. Only those grouping columns that have isolatable selection predicates with an equal operator specified can be considered. This guarantees that the column can only match a single value and will not help determine a unique group. This processing is done to allow the optimizer to consider more indexes to implement the query and to reduce the number of columns that will be added as key columns to a temporary index or hash table.

The following example illustrates a query where the optimizer could eliminate a grouping column.


```

DECLARE DEPTEMP CURSOR FOR
SELECT EMPNO, LASTNAME, WORKDEPT
FROM CORPDATA.EMPLOYEE
WHERE EMPNO = '000190'
GROUP BY EMPNO, LASTNAME, WORKDEPT

```

OPNQRYF example:

```

OPNQRYF FILE(EMPLOYEE) FORMAT(FORMAT1)
QRYSLT('EMPNO *EQ ''000190''')
GRPFLD(EMPNO LASTNAME WORKDEPT)

```

In this example, the optimizer can remove EMPNO from the list of grouping columns because of the EMPNO = '000190' selection predicate. An index that only has LASTNAME and WORKDEPT specified as key columns can be considered to implement the query and if a temporary index or hash is required then EMPNO will not be used.

Note: Even though EMPNO can be removed from the list of grouping columns, the optimizer might still choose to use that index if a permanent index exists with all three grouping columns.

Optimizing grouping by removing read triggers

For queries involving physical files or tables with read triggers, group by triggers will always involve a temporary file prior to the group by processing, and will therefore slow down these queries.

Note: Read triggers are added when the ADDPFTRG command has been used on the table with TRGTIME (*AFTER) and TRGEVENT (*READ).

The query will run faster if the read trigger is removed (RMVPFTRG TRGTIME (*AFTER) TRGEVENT (*READ)).

Optimizing grouping by adding additional grouping columns

The same logic that is applied to removing grouping columns can also be used to add additional grouping columns to the query. This is only done when you are trying to determine if an index can be used to implement the grouping.

The following example illustrates a query where the optimizer could add an additional grouping column.

```

CREATE INDEX X1 ON EMPLOYEE
(LASTNAME, EMPNO, WORKDEPT)

DECLARE DEPTEMP CURSOR FOR
SELECT LASTNAME, WORKDEPT
FROM CORPDATA.EMPLOYEE
WHERE EMPNO = '000190'
GROUP BY LASTNAME, WORKDEPT

```

OPNQRYF example:

```

OPNQRYF FILE ((EMPLOYEE)) FORMAT(FORMAT1)
QRYSLT('EMPNO *EQ ''000190''')
GRPFLD(LASTNAME WORKDEPT)

```

For this query request, the optimizer can add EMPNO as an additional grouping column when considering X1 for the query.

Optimizing grouping by using index skip key processing

Index Skip Key processing can be used when grouping with the keyed sequence implementation algorithm which uses an existing index. The index skip key processing algorithm:

1. Uses the index to position to a group and

2. finds the first row matching the selection criteria for the group, and if specified the first non-null MIN or MAX value in the group
3. Returns the group to the user
4. "Skip" to the next group and repeat processing

This will improve performance by potentially not processing all index key values for a group.

Index skip key processing can be used:

- For single table queries using the keyed sequence grouping implementation when:
 - There are no column functions in the query, or
 - There is only a single MIN or MAX column function in the query and the operand of the MIN or MAX is the next key column in the index after the grouping columns. There can be no other grouping functions in the query. For the MIN function, the key column must be an ascending key; for the MAX function, the key column must be a descending key. If the query is whole table grouping, the operand of the MIN or MAX must be the first key column.

Example 1, using SQL:

```
CREATE INDEX IX1 ON EMPLOYEE (SALARY DESC)
```

```
DECLARE C1 CURSOR FOR  
  SELECT MAX(SALARY) FROM EMPLOYEE;
```

Example 1, using the OPNQRYF command:

```
OPNQRYF FILE(EMPLOYEE) FORMAT(FORMAT1)  
  MAPFLD((MAXSAL '%MAX(SALARY)'))
```

The query optimizer will chose to use the index IX1. The SLIC runtime code will scan the index until it finds the first non-null value for SALARY. Assuming that SALARY is not null, the runtime code will position to the first index key and return that key value as the MAX of salary. No more index keys will be processed.

Example 2, using SQL:

```
CREATE INDEX IX2 ON EMPLOYEE (DEPT, JOB,SALARY)
```

```
DECLARE C1 CURSOR FOR  
  SELECT DEPT, MIN(SALARY)  
  FROM EMPLOYEE  
  WHERE JOB='CLERK'  
  GROUP BY DEPT
```

Example 2, using the OPNQRYF command:

```
OPNQRYF FILE(EMPLOYEE) FORMAT(FORMAT2)  
  QRYSLT('JOB *EQ ''CLERK''')  
  GRPFLD((DEPT))  
  MAPFLD((MINSAL '%MIN(SALARY)'))
```

The query optimizer will chose to use Index IX2. The SLIC runtime code will position to the first group for DEPT where JOB equals 'CLERK' and will return the SALARY. The code will then skip to the next DEPT group where JOB equals 'CLERK'.

- For join queries:
 - All grouping columns must be from a single table.
 - For each dial there can be at most one MIN or MAX column function operand that references the dial and no other column functions can exist in the query.
 - If the MIN or MAX function operand is from the same dial as the grouping columns, then it uses the same rules as single table queries.

- If the MIN or MAX function operand is from a different dial then the join column for that dial must join to one of the grouping columns and the index for that dial must contain the join columns followed by the MIN or MAX operand.

Example 1, using SQL:

```
CREATE INDEX IX1 ON DEPARTMENT(DEPTNAME)

CREATE INDEX IX2 ON EMPLOYEE(WORKDEPT, SALARY)

DECLARE C1 CURSOR FOR
  SELECT DEPTNAME, MIN(SALARY)
  FROM DEPARTMENT, EMPLOYEE
  WHERE DEPARTMENT.DEPTNO=EMPLOYEE.WORKDEPT
  GROUP BY DEPARTMENT.DEPTNO;
```

Example 1, using the OPNQRYF command:

```
OPNQRYF FILE(DEPARTMENT EMPLOYEE) FORMAT(FORMAT1)
  JFLD((1/DEPTNO 2/WORKDEPT *EQ))
  GRPFLD((1/DEPTNO))
  MAPFLD((MINSAL '%MIN(SALARY)'))
```

Ordering optimization

This section describes how DB2 Universal Database for iSeries implements ordering techniques, and how optimization choices are made by the query optimizer. The query optimizer can use either index ordering or a sort to implement ordering.

Sort Ordering implementation

The sort algorithm reads the rows into a sort space and sorts the rows based on the specified ordering keys. The rows are then returned to the user from the ordered sort space.

Index Ordering implementation

The index ordering implementation requires an index that contains all of the ordering columns as contiguous leftmost key columns. The database manager accesses the individual rows through the index in index order, which results in the rows being returned in order to the requester.

This implementation can be beneficial if an application does not need to retrieve all of the ordered results, or if an index already exists that matches the ordering columns. When the ordering is implemented with an index, and a permanent index does not already exist that satisfies ordering columns, a temporary index is created. The ordering columns specified within the query are used as the key columns for this index.

Optimizing ordering by eliminating ordering columns

All of the ordering columns are evaluated to determine if they can be removed from the list of ordering columns. Only those ordering columns that have isolatable selection predicates with an equal operator specified can be considered. This guarantees that the column can match only a single value, and will not help determine in the order.

This processing is done to allow the optimizer to consider more indexes as it implements the query, and to reduce the number of columns that will be added as key columns to a temporary index. The following SQL example illustrates a query where the optimizer could eliminate an ordering column.

```
DECLARE DEPTEMP CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  WHERE EMPNO = '000190'
  ORDER BY EMPNO, LASTNAME, WORKDEPT
```

OPNQRYF example:

```
OPNQRYF FILE(EMPLOYEE) FORMAT(FORMAT1)
  QRYSLT('EMPNO *EQ '000190''')
  KEYFLD(EMPNO LASTNAME WORKDEPT)
```

In this example, the optimizer can remove EMPNO from the list of ordering columns because of the EMPNO = '000190' selection predicate. An index that has only LASTNAME and WORKDEPT specified as key columns can be considered to implement the query; if a temporary index is required, then EMPNO will not be used.

Note: Even though EMPNO can be removed from the list of ordering columns, the optimizer might still choose to use that index if a permanent index exists with all three ordering columns.

Optimizing ordering by adding additional ordering columns

The same logic that is applied to removing ordering columns can also be used to add additional grouping columns to the query. This is done only when you are trying to determine if an index can be used to implement the ordering.

The following example illustrates a query where the optimizer could add an additional ordering column.

```
CREATE INDEX X1 ON EMPLOYEE (LASTNAME, EMPNO, WORKDEPT)
```

```
DECLARE DEPTEMP CURSOR FOR
  SELECT LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  WHERE EMPNO = '000190'
  ORDER BY LASTNAME, WORKDEPT
```

OPNQRYF example:

```
OPNQRYF FILE ((EMPLOYEE)) FORMAT(FORMAT1)
  QRYSLT('EMPNO *EQ '000190''')
  KEYFLD(LASTNAME WORKDEPT)
```

For this query request, the optimizer can add EMPNO as an additional ordering column when considering X1 for the query.

View implementation

Views are implemented by the query optimizer using one of two methods:

- The optimizer combines the query select statement with the select statement of the view (view composite)
- The optimizer places the results of the view in a temporary table and then replaces the view reference in the query with the temporary table (view materialization)

This also applies to nested table expressions and common table expressions except where noted.

View composite implementation

The view composite implementation takes the query select statement and combines it with the select statement of the view to generate a new query. The new, combined select statement query is then run directly against the underlying base tables.

This single, composite statement is the preferred implementation for queries containing views, since it requires only a single pass of the data.

Examples:

```
CREATE VIEW D21EMPL AS
  SELECT * FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT='D21'
```

Using SQL:

```

SELECT LASTNAME, FIRSTNME, SALARY
FROM D21EMPL
WHERE JOB='CLERK'

```

Using OPNQRYF:

```

OPNQRYF FILE(D21EMPL)
FORMAT(FORMAT1)
QRYSLT('JOB *EQ ''CLERK''')

```

The query optimizer will generate a new query that looks like the following example:

```

SELECT LASTNAME, FIRSTNME, SALARY
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT='D21' AND JOB='CLERK'

```

The query contains the columns selected by the user's query, the base tables referenced in the query, and the selection from both the view and the user's query.

Note: The new composite query that the query optimizer generates is not visible to users. Only the original query against the view will be seen by users and database performance tools.

View materialization implementation

The view materialization implementation runs the query of the view and places the results in a temporary result table. The view reference in the user's query is then replaced with the temporary table, and the query is run against the temporary result table.

View materialization is done whenever it is not possible to create a view composite. The following types of queries require view materialization:

- The outermost select of the view contains grouping, the query contains grouping, and refers to a column derived from a column function in the view in the HAVING or select-list.
- The query is a join and the outermost select of the view contains grouping or DISTINCT.
- The outermost select of the view contains DISTINCT, and the query has UNION, grouping, or DISTINCT and one of the following:
 - Only the query has a shared weight NLSS table
 - Only the view has a shared weight NLSS table
 - Both the query and the view have a shared weight NLSS table, but the tables are different.
- The query contains a column function and the outermost select of the view contains a DISTINCT
- The view does not contain an access plan. This can occur when a view references a view and a view composite cannot be created because of one of the reasons listed above. This does not apply to nested table expressions and common table expressions.

Since a temporary result table is created, access methods that are allowed with ALWCPYDTA(*OPTIMIZE) may be used to implement the query. These methods include hash grouping, hash join, and bitmaps.

Examples:

```

CREATE VIEW AVGSALVW AS
SELECT WORKDEPT, AVG(SALARY) AS AVGSAL
FROM CORPDATA.EMPLOYEE
GROUP BY WORKDEPT

```

SQL example:

```

SELECT D.DEPTNAME, A.AVGSAL
FROM CORPDATA.DEPARTMENT D, AVGSALVW A
WHERE D.DEPTNO=A.WORKDEPT

```

OPNQRYF example:

```
OPNQRYP FILE(CORPDATA/DEPARTMENT AVGSALVW)
  FORMAT(FORMAT1)
  JFLD((1/DEPTNO 2/WORKDEPT *EQ))
```

In this case, a view composite cannot be created since a join query references a grouping view. The results of AVGSALVW are placed in a temporary result table (*QUERY0001). The view reference AVGSALVW is replaced with the temporary result table. The new query is then run. The generated query looks like the following:

```
SELECT D.DEPTNAME, A.AVGSAL
  FROM CORPDATA.DEPARTMENT D, *QUERY0001 A
 WHERE D.DEPTNO=A.WORKDEPT
```

Note: The new query that the query optimizer generates is not visible to users. Only the original query against the view will be seen by users and database performance tools.

Whenever possible, isolatable selection from the query, except subquery predicates, is added to the view materialization process. This results in smaller temporary result tables and allows existing indexes to be used when materializing the view. This will not be done if there is more than one reference to the same view or common table expression in the query. The following is an example where isolatable selection is added to the view materialization:

```
SELECT D.DEPTNAME,A.AVGSAL
  FROM CORPDATA.DEPARTMENT D, AVGSALVW A
 WHERE D.DEPTNO=A.WORKDEPT
        A.WORKDEPT LIKE 'D%' AND AVGSAL>10000
```

OPNQRYP example:

```
OPNQRYP FILE(CORPDATA/DEPARTMENT AVGSALVW)
  FORMAT(FORMAT1)
  JFLD((1/DEPTNO 2/WORKDEPT *EQ))
  QRYSLT('1/WORKDEPT *EQ %WLDCRD(''D*'') *AND 2/AVGSAL *GT 10000')
```

The isolatable selection from the query is added to view resulting in a new query to generate the temporary result table:

```
SELECT WORKDEPT, AVG(SALARY) AS AVGSAL
  FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT LIKE 'D%'
 GROUP BY WORKDEPT
 HAVING AVG(SALARY)>10000
```

Chapter 4. Optimizing query performance using query optimization tools

You can use query optimization tools to improve data retrieval time. Use the results of the tools to:

- Change the data access method chosen by the server. See “Data access methods: Summary” on page 5.
- Create the correct indexes and use them effectively. See Chapter 5, “Using indexes to speed access to large tables” on page 103.

Query optimization is an iterative process. Do the following as needed to optimize your queries.

Gather statistics about your queries

There are various ways to gather statistics about your queries. The following is a sampling of the ways that statistics can be gathered:

- “Verify the performance of SQL applications” on page 60
- “Examine query optimizer debug messages in the job log” on page 60
- “Gather information about embedded SQL statements with the PRTSQLINF command” on page 67
- “Gather statistics about your queries with the database monitor” on page 69
- “Gather statistics about your queries with memory-resident database monitor APIs” on page 78
- “View the effectiveness of your queries with Visual Explain” on page 82
- “Monitoring your database performance using SQL Performance monitors in iSeries Navigator” on page 80

Control the processing of your queries:

- “Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command” on page 83
- “Control queries dynamically with the query options file QAQQINI” on page 84
- “Control long-running queries with the DB2 UDB for iSeries Predictive Query Governor” on page 93
- “Control parallel processing for queries” on page 97

Comparing the different tools:

You may want to check out the “Query optimization tools: Comparison table” on page 101 to learn:

- What information each tool can yield about your query
- When in the process a specific tool can analyze your query
- The tasks each tool can perform to improve your query

For additional tips and techniques:

If you are experienced with query optimization, you may want to refer to a list of “General query optimization tips” on page 34.

Also, the following topics provide programming tips and techniques for optimizing your applications for query performance:

- Chapter 6, “Application design tips for database performance” on page 117
- Chapter 7, “Programming techniques for database performance” on page 123
- Chapter 8, “General DB2 UDB for iSeries performance considerations” on page 129

Verify the performance of SQL applications

You can verify the performance of an SQL application by using the following commands:

DSPJOB

You can use the Display Job (DSPJOB) command with the OPTION(*OPNF) parameter to show the indexes and tables being used by an application that is running in a job.

You can also use DSPJOB with the OPTION(*JOBLOCK) parameter to analyze object and row lock contention. It displays the objects and rows that are locked and the name of the job holding the lock.

Specify the OPTION(*CMTCTL) parameter on the DSPJOB command to show the isolation level that the program is running, the number of rows being locked during a transaction, and the pending DDL functions. The isolation level displayed is the default isolation level. The actual isolation level, used for any SQL program, is specified on the COMMIT parameter of the CRTSQLxxx command.

PRTSQLINF

The Print SQL Information (PRTSQLINF) command lets you print information about the embedded SQL statements in a program, SQL package, or service program. The information includes the SQL statements, the access plans used during the running of the statement, and a list of the command parameters used to precompile the source member for the object. For more information on printing information about SQL Statements, see the PRTSQLINF section in “Gather information about embedded SQL statements with the PRTSQLINF command” on page 67.

STRDBMON

You can use the Start Database Monitor (STRDBMON) command to capture to a file information about every SQL statement that runs. See “Gather statistics about your queries with the database monitor” on page 69 for more information.

CHGQRYA

You can use the Change Query Attribute (CHGQRYA) command to change the query attributes for the query optimizer. Among the attributes that can be changed by this command are the predictive query governor, parallelism, and the query options.

STRDBG

You can use the Start Debug (STRDBG) command to put a job into debug mode and, optionally, add as many as 20 programs and 20 class files and 20 service programs to debug mode. It also specifies certain attributes of the debugging session. For example, it can specify whether database files in production libraries can be updated while in debug mode.

Examine query optimizer debug messages in the job log

Query optimizer debug messages issue informational messages to the job log about the implementation of a query. These messages explain what happened during the query optimization process. For example, you can learn:

- Why an index was or was not used
- Why a temporary result was required
- Whether joins and blocking are used
- What type of index was advised by the optimizer
- Status of the job’s queries
- Indexes used
- Status of the cursor

The optimizer automatically logs messages for all queries it optimizes, including SQL, call level interface, ODBC, OPNQRYF, and SQL Query Manager.

Viewing debug messages:

To view the messages, put your job into debug mode using one of the following methods:

- Use the following command:

```
STRDBG PGM(Library/program) UPDPDPROD(*YES)
```

STRDBG places in the job log information about all SQL statements that run.

- Set the QRYOPTLIB parameter on the Change Query Attributes (CHGQRYA) command to a user library where the QAQQINI file exists. Set the parameter on the QAQQINI file to MESSAGES_DEBUG, and set the value to *YES. This option places query optimization information in the job log.

Pressing F10 from the command Entry panel displays the message text. To see the second-level text, press F1 (Help). The second-level text sometimes offers hints for improving query performance.

See “Query optimization performance information messages” and “Query optimization performance information messages and open data paths” on page 66 for the specific meanings of the debug messages.

See Viewing the Job Log in the SQL Programming Concepts book for information on viewing the job log with iSeries Navigator.

Query optimization performance information messages

You can evaluate the structure and performance of the given SQL statements in a program using informational messages put in the job log by the database manager. The messages are issued for an SQL program or interactive SQL when running in the debug mode. The database manager may send any of the following messages when appropriate. The ampersand variables (&1, &X) are replacement variables that contain either an object name or some other substitution value when the message appears in the job log. The messages are:

- “CPI4321 - Access path built for file &1.” on page 62
- “CPI4322 - Access path built from keyed file &1” on page 63
- “CPI4323 - The OS/400 query access plan has been rebuilt” on page 63
- “CPI4324 - Temporary file built for file &1” on page 63
- “CPI4325 - Temporary result file built for query” on page 64
- “CPI4326 - File &1 processed in join position &11” on page 64
- “CPI4327 - File &13 processed in join position &10” on page 64
- “CPI4328 - Access path of file &4 was used by query” on page 64
- “CPI4329 - Arrival sequence access was used for file &1” on page 64
- “CPI432A - Query optimizer timed out for file &1” on page 64
- “CPI432B - Subselects processed as join query” on page 65
- “CPI432C - All access paths were considered for file &1” on page 65
- “CPI432D - Additional access path reason codes were used” on page 65
- “CPI432E - Selection columns mapped to different attributes” on page 65
- CPI432F Access path suggestion for file &1.
- CPI4330 &6 tasks used for parallel &10 scan of file &1.
- CPI4331 &6 tasks used for parallel index created over file &1.
- CPI4332 &1 host variables used in query.
- CPI4333 Hashing algorithm used to process join.
- CPI4334 Query implemented as reusable ODP.
- CPI4335 Optimizer debug messages for hash join step &1 follow:
- CPI4336 Group processing generated.

- CPI4337 Temporary hash table built for hash join step &1.
- “CPI4338 - &1 Access path(s) used for bitmap processing of file &2” on page 65
- CPI4339 Query options retrieved from Library &1.
- CPI433A Unable to retrieve query options file.
- CPI433C Library &1 not found.
- CPI4341 Performing distributed query.
- CPI4342 Performing distributed join for query.
- CPI4345 Temporary distributed result file &4 built for query.
- CPI4346 Optimizer debug messages for query join step &1 of &2 follow:
- CPI4347 Query being processed in multiple steps.
- CPI4348 The ODP associated with the cursor was hard closed.
- CPI4349 Fast past refresh of the host variable values is not possible.
- CPI434A &1 Starting optimizer debug message for query &2.
- CPI434B &1 Ending debug message for query &2.
- CPI434C The OS/400 Query access plan was not rebuilt.
- “SQL7910 - All SQL cursors closed” on page 66
- “SQL7911 - ODP reused” on page 66
- “SQL7912 - ODP created” on page 66
- “SQL7913 - ODP deleted” on page 66
- “SQL7914 - ODP not deleted” on page 67
- “SQL7915 - Access plan for SQL statement has been built” on page 67
- “SQL7916 - Blocking used for query” on page 67
- “SQL7917 - Access plan not updated” on page 67
- “SQL7918 - Reusable ODP deleted” on page 67
- “SQL7919 - Data conversion required on FETCH or embedded SELECT” on page 67
- “SQL7939 - Data conversion required on INSERT or UPDATE” on page 67

These messages provide feedback on how a query was run and, in some cases, indicate the improvements that can be made to help the query run faster.

The messages contain message help that provides information about the cause for the message, object name references, and possible user responses.

The time at which the message is sent does not necessarily indicate when the associated function was performed. Some messages are sent altogether at the start of a query run.

The causes and user responses for the following messages are paraphrased. The actual message help is more complete and should be used when trying to determine the meaning and responses for each message.

The possible user action for each message are described in the following sections:

CPI4321 - Access path built for file &1.

This message indicates that a temporary index was created to process the query. The new index is created by reading all of the rows in the specified table.

The time required to create an index on each run of a query can be significant. Consider creating a logical file (CRTLF) or an SQL index (CREATE INDEX SQL statement):

- Over the table named in the message help.

- With key columns named in the message help.
- With the ascending or descending sequencing specified in the message help.
- With the sort sequence table specified in the message help.

Consider creating the logical file with select or omit criteria that either match or partially match the query's predicates involving constants. The database manager will consider using select or omit logical files even though they are not explicitly specified on the query.

For certain queries, the optimizer may decide to create an index even when an existing one can be used. This might occur when a query has an ordering column as a key column for an index, and the only row selection specified uses a different column. If the row selection results in roughly 20% of the rows or more to be returned, then the optimizer may create a new index to get faster performance when accessing the data. The new index minimizes the amount of data that needs to be read.

CPI4322 - Access path built from keyed file &1

This message indicates that a temporary index was created from the access path of an existing keyed table or index.

Generally, this action should not take a significant amount of time or resource because only a subset of the data in the table needs to be read. This is normally done to allow the optimizer to use an existing index for selection while creating one for ordering, grouping, or join criteria. Sometimes even faster performance can be achieved by creating a logical file or SQL index that satisfies the index requirement stated in the message help.

For more detail, see the previous message, CPI4321.

CPI4323 - The OS/400 query access plan has been rebuilt

This message can be sent for a variety of reasons. The specific reason is provided in the message help.

Most of the time, this message is sent when the queried table environment has changed, making the current access plan obsolete. An example of the table environment changing is when an index required by the query no longer exists on the server.

An access plan contains the instructions for how a query is to be run and lists the indexes for running the query. If a needed index is no longer available, the query is again optimized, and a new access plan is created, replacing the old one.

The process of again optimizing the query and building a new access plan at runtime is a function of DB2 UDB for iSeries. It allows a query to be run as efficiently as possible, using the most current state of the database without user intervention.

The infrequent appearance of this message is not a cause for action. For example, this message will be sent when an SQL package is run the first time after a restore, or anytime the optimizer detects that a change has occurred (such as a new index was created), that warrants an implicit rebuild. However, excessive rebuilds should be avoided because extra query processing will occur. Excessive rebuilds may indicate a possible application design problem or inefficient database management practices. See CPI434C.

CPI4324 - Temporary file built for file &1

Before the query processing could begin, the data in the specified table had to be copied into a temporary physical table to simplify running the query. The message help contains the reason why this message was sent.

If the specified table selects few rows, usually less than 1000 rows, then the row selection part of the query's implementation should not take a significant amount of resource and time. However if the query is taking more time and resources than can be allowed, consider changing the query so that a temporary table is not required.

One way to do this is by breaking the query into multiple steps. Consider using an INSERT statement with a subselect to select only the rows that are required into a table, and then use that table's rows for the rest of the query.

CPI4325 - Temporary result file built for query

A temporary result table was created to contain the intermediate results of the query. The results are stored in an internal temporary table (structure). This allows for more flexibility by the optimizer in how to process and store the results. The message help contains the reason why a temporary result table is required.

In some cases, creating a temporary result table provides the fastest way to run a query. Other queries that have many rows to be copied into the temporary result table can take a significant amount of time. However, if the query is taking more time and resources than can be allowed, consider changing the query so that a temporary result table is not required.

CPI4326 - File &1 processed in join position &11

This message provides the join position of the specified table when an index is used to access the table's data. **Join position** pertains to the order in which the tables are joined. See the Join optimization section for details.

CPI4327 - File &13 processed in join position &10

This message provides the name of the table and the join position when table access scan method is used to select rows from the table.

See the previous message, CPI4326, for information on join position and join performance tips.

CPI4328 - Access path of file &4 was used by query

This message names an existing index that was used by the query.

The reason the index was used is given in the message help.

CPI4329 - Arrival sequence access was used for file &1

No index was used to access the data in the specified table. The rows were scanned sequentially in arrival sequence.

If an index does not exist, you may want to create one whose key column matches one of the columns in the row selection. You should only create an index if the row selection (WHERE clause) selects 20% or fewer rows in the table. To force the use of an existing index, change the ORDER BY clause of the query to specify the first key column of the index, or ensure that the query is running under a first I/O environment.

CPI432A - Query optimizer timed out for file &1

The optimizer stops considering indexes when the time spent optimizing the query exceeds an internal value that corresponds to the estimated time to run the query and the number of rows in the queried tables. Generally, the more rows in the tables, the greater the number of indexes that will be considered.

When the estimated time to run the query is exceeded, the optimizer does not consider any more indexes and uses the current best method to implement the query. Either an index has been found to get the best performance, or an index will have to be created. If the actual time to execute the query exceeds the estimated run time this may indicate the optimizer did not consider the best index.

The message help contains a list of indexes that were considered before the optimizer timed out. By viewing this list of indexes, you may be able to determine if the optimizer timed out before the best index was considered.

To ensure that an index is considered for optimization, specify the logical file associated with the index as the table to be queried. The optimizer will consider the index of the table specified on the query or SQL statement first. Remember that SQL indexes cannot be queried.

You may want to delete any indexes that are no longer needed.

CPI432B - Subselects processed as join query

Two or more SQL subselects were combined by the query optimizer and processed as a join query. Generally, this method of processing is a good performing option.

CPI432C - All access paths were considered for file &1

The optimizer considered all indexes built over the specified table. Since the optimizer examined all indexes for the table, it determined the current best access to the table.

The message help contains a list of the indexes. With each index a reason code is added. The reason code explains why the index was or was not used.

CPI432D - Additional access path reason codes were used

Message CPI432A or CPI432C was issued immediately before this message. Because of message length restrictions, some of the reason codes used by messages CPI432A and CPI432C are explained in the message help of CPI432D. Use the message help from this message to interpret the information returned from message CPI432A or CPI432C.

CPI432E - Selection columns mapped to different attributes

This message indicates that the query optimizer was not able to consider the usage of an index to resolve one or more of the selection specifications of the query. If there was an index available which otherwise could have been used to limit the processing of the query to just a few rows, then the performance of this query will be affected.

The attributes of a comparison value and a comparison column must match otherwise a conversion will occur so that they do match. Generally, this conversion occurs such that the value with the smallest attributes is mapped to the attributes of the other value. When the attributes of the comparison column have to be mapped to be compatible with that of the comparison value, the optimizer can no longer use an index to implement this selection.

CPI4338 - &1 Access path(s) used for bitmap processing of file &2

The optimizer chooses to use one or more indexes, in conjunction with the query selection (WHERE clause), to build a bitmap. This resulting bitmap indicates which rows will actually be selected.

Conceptually, the bitmap contains one bit per row in the underlying table. Corresponding bits for selected rows are set to '1'. All other bits are set to '0'.

Once the bitmap is built, it is used, as appropriate, to avoid mapping in rows from the table not selected by the query. The use of the bitmap depends on whether the bitmap is used in combination with the arrival sequence or with a primary index.

When bitmap processing is used with arrival sequence, either message CPI4327 or CPI4329 will precede this message. In this case, the bitmap will help to selectively map only those rows from the table that the query selected.

When bitmap processing is used with a primary index, either message CPI4326 or CPI4328 will precede this message. Rows selected by the primary index will be checked against the bitmap before mapping the row from the table. See the Bitmap processing access method for details.

Query optimization performance information messages and open data paths

Several of the following SQL run-time messages refer to open data paths.

An open data path (ODP) definition is an internal object that is created when a cursor is opened or when other SQL statements are run. It provides a direct link to the data so that I/O operations can occur. ODPs are used on OPEN, INSERT, UPDATE, DELETE, and SELECT INTO statements to perform their respective operations on the data.

Even though SQL cursors are closed and SQL statements have already been run, the database manager in many cases will save the associated ODPs of the SQL operations to reuse them the next time the statement is run. So an SQL CLOSE statement may close the SQL cursor but leave the ODP available to be used again the next time the cursor is opened. This can significantly reduce the processing and response time in running SQL statements.

The ability to reuse ODPs when SQL statements are run repeatedly is an important consideration in achieving faster performance.

The following informational messages are issued at SQL run time:

SQL7910 - All SQL cursors closed

This message is sent when the job's call stack no longer contains a program that has run an SQL statement.

Unless CLOSQLCSR(*ENDJOB) or CLOSQLCSR(*ENDACTGRP) was specified, the SQL environment for reusing ODPs across program calls exists only until the active programs that ran the SQL statements complete.

Except for ODPs associated with *ENDJOB or *ENDACTGRP cursors, all ODPs are deleted when all the SQL programs on the call stack complete and the SQL environment is exited.

This completion process includes closing of cursors, the deletion of ODPs, the removal of prepared statements, and the release of locks.

Putting an SQL statement that can be run in the first program of an application keeps the SQL environment active for the duration of that application. This allows ODPs in other SQL programs to be reused when the programs are repeatedly called. CLOSQLCSR(*ENDJOB) or CLOSQLCSR(*ENDACTGRP) can also be specified.

SQL7911 - ODP reused

This message indicates that the last time the statement was run or when a CLOSE statement was run for this cursor, the ODP was not deleted. It will now be used again. This should be an indication of very efficient use of resources by eliminating unnecessary OPEN and CLOSE operations.

SQL7912 - ODP created

No ODP was found that could be used again. The first time that the statement is run or the cursor is opened for a process, an ODP will always have to be created. However, if this message appears on every run of the statement or open of the cursor, the tips recommended in "Database application design tips: Retaining cursor positions for non-ILE program calls" on page 120 should be applied to this application.

SQL7913 - ODP deleted

For a program that is run only once per job, this message could be normal. However, if this message appears on every run of the statement or open of the cursor, then the tips recommended in "Database application design tips: Retaining cursor positions for non-ILE program calls" on page 120 should be applied to this application.

SQL7914 - ODP not deleted

If the statement is rerun or the cursor is opened again, the ODP should be available again for use.

SQL7915 - Access plan for SQL statement has been built

The DB2 UDB for iSeries precompilers allow the creation of the program objects even when required tables are missing. In this case the binding of the access plan is done when the program is first run. This message indicates that an access plan was created and successfully stored in the program object.

SQL7916 - Blocking used for query

SQL will request multiple rows from the database manager when running this statement instead of requesting one row at a time.

SQL7917 - Access plan not updated

The database manager rebuilt the access plan for this statement, but the program could not be updated with the new access plan. Another job is currently running the program that has a shared lock on the access plan of the program.

The program cannot be updated with the new access plan until the job can obtain an exclusive lock on the access plan of the program. The exclusive lock cannot be obtained until the shared lock is released.

The statement will still run and the new access plan will be used; however, the access plan will continue to be rebuilt when the statement is run until the program is updated.

SQL7918 - Reusable ODP deleted

A reusable ODP exists for this statement, but either the job's library list or override specifications have changed the query.

The statement now refers to different tables or uses different override specifications than are in the existing ODP. The existing ODP cannot be reused, and a new ODP must be created. To make it possible to reuse the ODP, avoid changing the library list or the override specifications.

SQL7919 - Data conversion required on FETCH or embedded SELECT

When mapping data to host variables, data conversions were required. When these statements are run in the future, they will be slower than if no data conversions were required. The statement ran successfully, but performance could be improved by eliminating the data conversion. For example, a data conversion that would cause this message to occur would be the mapping of a character string of a certain length to a host variable character string with a different length. You could also cause this error by mapping a numeric value to a host variable that is a different type (decimal to integer). To prevent most conversions, use host variables that are of identical type and length as the columns that are being fetched.

SQL7939 - Data conversion required on INSERT or UPDATE

The attributes of the INSERT or UPDATE values are different than the attributes of the columns receiving the values. Since the values must be converted, they cannot be directly moved into the columns. Performance could be improved if the attributes of the INSERT or UPDATE values matched the attributes of the columns receiving the values.

Gather information about embedded SQL statements with the PRTSQLINF command

The PRTSQLINF command gathers information about the embedded SQL statements in a program, SQL package (the object normally used to store the access plan for a remote query), or service program. It then puts the information in a spooled file. PRTSQLINF provides information about:

- The SQL statements being executed
- The type of access plan used during execution. This includes information about how the query will be implemented, the indexes used, the join order, whether a sort is done, whether a database scan is used, and whether an index is created.

- A list of the command parameters used to precompile the source member for the object.

To gather this information, run PRTSQLINF against a saved access plan, or use the PRTSQLINF function in iSeries Navigator. This means you must execute or at least prepare the query (using SQL's PREPARE statement) before you use the command. It is best to execute the query because the index created as a result of PREPARE is relatively sparse and may well change after the first run. PRTSQLINF's requirement of a saved access plan means the command cannot be used with OPNQRYF.

PRTSQLINF gives output that is similar to the information you can get from debug messages, but PRTSQLINF must be run against a saved access plan. The query optimizer automatically logs information messages about the current query processing when your job is in debug mode. So, query debug messages work at runtime while PRTSQLINF works retroactively. You can also see this information in the second level text of the query governor inquiry message CPA4259. The messages are:

- SQL400A Temporary distributed result file &1 was created to contain join result. Result file was directed
- SQL400B Temporary distributed result file &1 was created to contain join result. Result file was broadcast.
- SQL400C Optimizer debug messages for distributed query step &1 and &2 follow.
- SQL400D GROUP BY processing generated.
- SQL400E Temporary distributed result file &1 was created while processing distributed subquery.
- SQL4001 Temporary result created.
- SQL4002 Reusable ODP sort used.
- SQL4003 UNION.
- SQL4004 SUBQUERY.
- SQL4005 Query optimizer timed out for table &1.
- SQL4006 All indexes considered for table &1.
- SQL4007 Query implementation for join position &1 table &2.
- SQL4008 Index &1 used for table &2.
- SQL4009 Index created for table &1.
- SQL401A Processing grouping criteria for query containing a distributed table.
- SQL401B Temporary distributed result table &1 was created while processing grouping criteria.
- SQL401C Performing distributed join for query.
- SQL401D Temporary distributed result table &1 was created because table &2 was directed.
- SQL401E Temporary distributed result table &1 was created because table &2 was broadcast.
- SQL401F Table &1 used in distributed join.
- SQL4010 Table scan access for table &1.
- SQL4011 Index scan-key row positioning used on table &1.
- SQL4012 Index created from index &1 for table &2.
- SQL4013 Access plan has not been built.
- SQL4014 &1 join column pair(s) are used for this join position.
- SQL4015 From-column &1.&2, to-column &3.&4, join operator &5, join predicate &6.
- SQL4016 Subselects processed as join query.
- SQL4017 Host variables implemented as reusable ODP.
- SQL4018 Host variables implemented as non-reusable ODP.
- SQL4019 Host variables implemented as file management row positioning reusable ODP.
- SQL402A Hashing algorithm used to process join.
- SQL402B Table &1 used in hash join step &2.
- SQL402C Temporary table created for hash join results.
- SQL402D Query attributes overridden from query options file &2 in library &1.

- SQL4020 Estimated query run time is &1 seconds.
- SQL4021 Access plan last saved on &1 at &2.
- SQL4022 Access plan was saved with SRVQRY attributes active.
- SQL4023 Parallel table prefetch used.
- SQL4024 Parallel index preload access method used.
- SQL4025 Parallel table preload access method used.
- SQL4026 Index only access used on table number &1.
- SQL4027 Access plan was saved with DB2 UDB Symmetric Multiprocessing installed on the system.
- SQL4028 The query contains a distributed table.
- SQL4029 Hashing algorithm used to process the grouping.
- SQL4030 &1 tasks specified for parallel scan on table &2.
- SQL4031 &1 tasks specified for parallel index create over table &2.
- SQL4032 Index &1 used for bitmap processing of table &2.
- SQL4033 &1 tasks specified for parallel bitmap create using &2.
- SQL4034 Multiple join classes used to process join.
- SQL4035 Table &1 used in join class &2.

Gather statistics about your queries with the database monitor

Database monitor statistics provide the most complete information about a query. You can gather performance statistics for a specific query or for every query on the server. There are several different ways to gather the statistics:

- Use the “Start Database Monitor (STRDBMON) command” on page 70 and the “End Database Monitor (ENDDDBMON) command” on page 71.
- Use the “Monitoring your database performance using SQL Performance monitors in iSeries Navigator” on page 80
- Use the Start Performance Monitor (STRPFRMON) command with the STRDBMON parameter.
- Use memory resident database monitor APIs. See “Gather statistics about your queries with memory-resident database monitor APIs” on page 78.

For examples on using the database monitor, see “Database monitor examples” on page 73.

Note: Database monitors can generate significant CPU and disk storage overhead when in use.

You can monitor a specific job or all jobs on the server. The statistics gathered are placed in the output database table specified on the command. Each job in the server can be monitored concurrently by two monitors:

- One started specifically on that job
- One started for all jobs in the server

When a job is monitored by two monitors, each monitor is logging rows to a different output table. You can identify rows in the output database table by each row’s unique identification number.

What kinds of statistics you can gather

The database monitor provides the same information that is provided with the query optimizer debug messages (STRDBG) and the Print SQL information (PRTSQLINF) command. The following is a sampling of the additional information that will be gathered by the database monitors:

- System and job name
- SQL statement and sub-select number

- Start and end timestamp
- Estimated processing time
- Total rows in table queried
- Number of rows selected
- Estimated number of rows selected
- Estimated number of joined rows
- Key columns for advised index
- Total optimization time
- Join type and method
- ODP implementation

How you can use performance statistics

You can use these performance statistics to generate various reports. For instance, you can include reports that show queries that:

- Use an abundance of the server resources.
- Take an extremely long time to execute.
- Did not run because of the query governor time limit.
- Create a temporary index during execution
- Use the query sort during execution
- Could perform faster with the creation of a keyed logical file containing keys suggested by the query optimizer.

Note: A query that is cancelled by an end request generally does not generate a full set of performance statistics. However, it does contain all the information about how a query was optimized, with the exception of runtime or multi-step query information.

Start Database Monitor (STRDBMON) command

The STRDBMON command starts the collection of database performance statistics for a specific job or all jobs on the server. The statistics are placed in an output database table and member specified on the command. If the output table and/or member does not exist, one is created based upon the table and format definition of model table QSYS/QAQQDBMN. If the output table and/or member exist, the row format of the output table must be named QQQDBMN.

You can specify a replace/append option that allows you to clear the member of information before writing rows or to just append new information to the end of the existing table.

You can also specify a force row write option that allows you to control how many rows are kept in the row buffer of each job being monitored before forcing the rows to be written to the output table. By specifying a force row write value of 1, FRCRCD(1), monitor rows will appear in the log as soon as they are created. FRCRCD(1) also ensures that the physical sequence of the rows are most likely, but not guaranteed, to be in time sequence. However, FRCRCD(1) will cause the most negative performance impact on the jobs being monitored. By specifying a larger number for the FRCRCD parameter, the performance impact of monitoring can be lessened.

Specifying *DETAIL on the TYPE parameter of the STRDBMON command indicates that detail rows, as well as summary rows, are to be collected. This is only useful for non-SQL queries, those queries which do not generate a QQQ1000 row. For non-SQL queries the only way to determine the number of rows returned and the total time to return those rows is to collect detail rows. Currently the only detail row is QQQ3019, in Appendix A, "Database monitor: DDS" on page 133. While the detail row contains valuable information, it creates a slight performance degradation for each block of rows returned. Therefore its use should be closely monitored.

If the monitor is started on all jobs, any jobs waiting on job queues or any jobs started during the monitoring period will have statistics gathered from them once they begin. If the monitor is started on a specific job, that job must be active in the server when the command is issued. Each job in the server can be monitored concurrently by only two monitors:

- One started specifically on that job.
- One started on all jobs in the server.

When a job is monitored by two monitors and each monitor is logging to a different output table, monitor rows will be written to both logs for this job. If both monitors have selected the same output table then the monitor rows are not duplicated in the output table.

End Database Monitor (ENDDBMON) command

The ENDDBMON command ends the Database Monitor for a specific job or all jobs on the server. If an attempt to end the monitor on all jobs is issued, there must have been a previous STRDBMON issued for all jobs. If a particular job is specified on this command, the job must have the monitor started explicitly and specifically on that job.

For example, consider the following sequence of events:

1. Monitoring was started for all jobs in the server.
2. Monitoring was started for a specific job.
3. Monitoring was ended for all jobs.

In this sequence, the specific job monitor continues to run because an explicit start of the monitor was done on it. It continues to run until an ENDDBMON on the specific job is issued.

Consider the following sequence:

1. Monitoring was started for all jobs in the server.
2. Monitoring was started for a specific job.
3. Monitoring was ended for the specific job.

In this sequence, monitoring continues to run for all jobs, even over the specific job, until an ENDDBMON for all jobs is issued.

In the following sequence:

1. Monitoring was started for a specific job.
2. Monitoring was started for all jobs in the server.
3. Monitoring was ended for all jobs.

In this sequence, monitoring continues to run for the specific job until you issue an ENDDBMON for that job.

In the following sequence:

1. Monitoring was started for a specific job.
2. Monitoring was started for all jobs in the server.
3. Monitoring was ended for the specific job.

In this sequence, monitoring continues to run for all jobs, including the specific job.

When monitoring is ended for all jobs, all of the jobs on the server will be triggered to close the output table, however, the ENDDBMON command can complete before all of the monitored jobs have written their final performance rows to the log. Use the Work with Object Locks (WRKOBJLCK) command to see that all of the monitored jobs no longer hold locks on the output table before assuming the monitoring is complete.

Database monitor performance rows

The rows in the database table are uniquely identified by their row identification number. The information within the file-based monitor (STRDBMON) is written out based upon a set of logical formats which are defined in Appendix A. These logical formats correlate closely to the debug messages and the PRSQLINF messages. The appendix also identifies which physical columns are used for each logical format and what information it contains. You can use the logical formats to identify the information that can be extracted from the monitor. These rows are defined in several different logical files which are not shipped with the server and must be created by the user, if desired. The logical files can be created with the DDS shown in “Optional database monitor logical file DDS” on page 140. The column descriptions are explained in the tables following each figure.

Note: The database monitor logical files are keyed logical files that contain some select/omit criteria. Therefore, there will be some maintenance overhead associated with these tables while the database monitor is active. The user may want to minimize this overhead while the database monitor is active, especially if monitoring all jobs. When monitoring all jobs, the number of rows generated could be quite large. The logicals are not required to process the results. They simply make the extraction of information for the table easier and more direct.

Minimizing maintenance overhead

Possible ways to minimize maintenance overhead associated with database monitor logical files:

- Do not create the database monitor logical files until the database monitor has completed.
- Create the database monitor logical files using dynamic select/omit criteria (DYNLSLT keyword on logical file's DDS).
- Create the database monitor logical files with rebuild index maintenance specified on the CRTLF command (*REBLD option on MAINT parameter).

By minimizing the maintenance overhead at run time, you are merely delaying the maintenance cost until the database monitor logical file is either created or opened. The choice is to either spend the time while the database monitor is active or spend the time after the database monitor has completed.

Query optimizer index advisor

The query optimizer analyzes the row selection in the query and determines, based on default values, if creation of a permanent index would improve performance. If the optimizer determines that a permanent index would be beneficial, it returns the key columns necessary to create the suggested index.

The index advisor information can be found in the Database Monitor logical files QQQ3000, QQQ3001 and QQQ3002. The advisor information is stored in columns QQIDXA, QQIDXK and QQIDX. When the QQIDXA column contains a value of 'Y' the optimizer is advising you to create an index using the key columns shown in column QQIDX. The intention of creating this index is to improve the performance of the query.

In the list of key columns contained in column QQIDX the optimizer has listed what it considers the suggested primary and secondary key columns. Primary key columns are columns that should significantly reduce the number of keys selected based on the corresponding query selection. Secondary key columns are columns that may or may not significantly reduce the number of keys selected.

The optimizer is able to perform index scan-key positioning over any combination of the primary key columns, plus one additional secondary key column. Therefore it is important that the first secondary key column be the most selective secondary key column. The optimizer will use index scan-key selection with any of the remaining secondary key columns. While index scan-key selection is not as fast as index scan-key positioning it can still reduce the number of keys selected. Hence, secondary key columns that are fairly selective should be included.

Column QQIDXK contains the number of suggested primary key columns that are listed in column QQIDXD. These are the left-most suggested key columns. The remaining key columns are considered secondary key columns and are listed in order of expected selectivity based on the query. For example, assuming QQIDXK contains the value of 4 and QQIDXD specifies 7 key columns, then the first 4 key columns specified in QQIDXK would be the primary key columns. The remaining 3 key columns would be the suggested secondary key columns.

It is up to the user to determine the true selectivity of any secondary key columns and to determine whether those key columns should be included when creating the index. When building the index the primary key columns should be the left-most key columns followed by any of the secondary key columns the user chooses and they should be prioritized by selectivity. The query optimizer index advisor should only be used to help analyze complex selection within a query that cannot be easily debugged manually.

Note: After creating the suggested index and executing the query again, it is possible that the query optimizer will choose not to use the suggested index. While the selection criteria is taken into consideration by the query optimizer, join, ordering, and grouping criteria are not.

Database monitor examples

Suppose you have an application program with SQL statements and you want to analyze and performance tune these queries. The first step in analyzing the performance is collection of data. The following examples show how you might collect and analyze data using STRDBMON and ENDDBMON.

Performance data is collected in LIB/PERFDATA for an application running in your current job. The following sequence collects performance data and prepares to analyze it.

1. STRDBMON FILE(LIB/PERFDATA). If this table does not already exist, the command will create one from the skeleton table in QSYS/QAQQDBMN.
2. Run your application
3. ENDDBMON
4. Create logical files over LIB/PERFDATA using the DDS shown in “Optional database monitor logical file DDS” on page 140. Creating the logical files is not mandatory. All of the information resides in the base table that was specified on the STRDBMON command. The logical files simply provide an easier way to view the data.

You are now ready to analyze the data. The following examples give you a few ideas on how to use this data. You should closely study the physical and logical file DDS to understand all the data being collected so you can create queries that give the best information for your applications.

Database monitor performance analysis example 1

Determine which queries in your SQL application are implemented with table scans. The complete information can be obtained by joining two logical files: QQQ1000, which contains information about the SQL statements, and QQQ3000, which contains data about queries performing table scans. The following SQL query could be used:

```
SELECT A.QQTLN, A.QQTFN, A.QQTOTR, A.QQIDXA, C.QQrcdr,
       (B.QQETIM - B.QQSTIM) AS TOT_TIME, B.QQSTTX
FROM   LIB/QQQ3000 A, LIB/QQQ1000 B, LIB/QQQ3019 C
WHERE  A.QQJFLD = B.QQJFLD
AND    A.QQCNT = B.QQCNT
AND    A.QQJFLD = C.QQJFLD AND A.QQCNT = C.QQCNT
```

Sample output of this query is shown in Table 3 on page 74. Key to this example are the join criteria:

```
WHERE A.QQJFLD = B.QQJFLD
AND   A.QQCNT = B.QQCNT
```

A lot of data about many queries is contained in multiple rows in table LIB/PERFDATA. It is not uncommon for data about a single query to be contained in 10 or more rows within the table. The combination of defining the logical files and then joining the tables together allows you to piece together all the data for a

query or set of queries. Column QQJFLD uniquely identifies all data common to a job; column QQUCNT is unique at the query level. The combination of the two, when referenced in the context of the logical files, connects the query implementation to the query statement information.

Table 3. Output for SQL Queries that Performed Table Scans

Lib Name	Table Name	Total Rows	Index Advised	Rows Returned	TOT_TIME	Statement Text
LIB1	TBL1	20000	Y	10	6.2	SELECT * FROM LIB1/TBL1 WHERE FLD1 = 'A'
LIB1	TBL2	100	N	100	0.9	SELECT * FROM LIB1/TBL2
LIB1	TBL1	20000	Y	32	7.1	SELECT * FROM LIB1/TBL1 WHERE FLD1 = 'B' AND FLD2 > 9000

If the query does not use SQL, the SQL information row (QQQ1000) is not created. This makes it more difficult to determine which rows in LIB/PERFDATA pertain to which query. When using SQL, row QQQ1000 contains the actual SQL statement text that matches the performance rows to the corresponding query. Only through SQL is the statement text captured. For queries executed using the OPNQRYF command, the OPNID parameter is captured and can be used to tie the rows to the query. The OPNID is contained in column QQOPID of row QQQ3014.

Database monitor performance analysis example 2

Similar to the preceding example that showed which SQL applications were implemented with table scans, the following example shows all queries that are implemented with table scans.

```

SELECT A.QQTLN, A.QQTFN, A.QQTOTR, A.QQIDXA,
       B.QQOPID, B.QQTTIM, C.QQCLKT, C.QQRCDR, D.QQRWR,
       (D.QQETIM - D.QQSTIM) AS TOT_TIME, D.QQSTTX
FROM LIB/QQQ3000 A INNER JOIN LIB/QQQ3014 B
   ON (A.QQJFLD = B.QQJFLD AND
       A.QQUCNT = B.QQUCNT)
   LEFT OUTER JOIN LIB/QQQ3019 C
   ON (A.QQJFLD = C.QQJFLD AND
       A.QQUCNT = C.QQUCNT)
   LEFT OUTER JOIN LIB/QQQ1000 D
   ON (A.QQJFLD = D.QQJFLD AND
       A.QQUCNT = D.QQUCNT)

```

In this example, the output for all queries that performed table scans are shown in Table 4.

Note: The columns selected from table QQQ1000 do return NULL default values if the query was not executed using SQL. For this example assume the default value for character data is blanks and the default value for numeric data is an asterisk (*).

Table 4. Output for All Queries that Performed Table Scans

Lib Name	Table Name	Total Rows	Index Advised	Query OPNID	ODP Open Time	Clock Time	Recs Rtned	Rows Rtned	TOT_TIME	Statement Text
LIB1	TBL1	20000	Y		1.1	4.7	10	10	6.2	SELECT * FROM LIB1/TBL1 WHERE FLD1 = 'A'
LIB1	TBL2	100	N		0.1	0.7	100	100	0.9	SELECT * FROM LIB1/TBL2
LIB1	TBL1	20000	Y		2.6	4.4	32	32	7.1	SELECT * FROM LIB1/TBL1 WHERE FLD1 = 'A' AND FLD2 > 9000

Table 4. Output for All Queries that Performed Table Scans (continued)

Lib Name	Table Name	Total Rows	Index Advised	Query OPNID	ODP Open Time	Clock Time	Recs Rtnd	Rows Rtnd	TOT_TIME	Statement Text
LIB1	TBL4	4000	N	QRY04	1.2	4.2	724	*	*	*

If the SQL statement text is not needed, joining to table QQQ1000 is not necessary. You can determine the total time and rows selected from data in the QQQ3014 and QQQ3019 rows.

Database monitor performance analysis example 3

Your next step may include further analysis of the table scan data. The previous examples contained a column titled Index Advised. A Y (yes) in this column is a hint from the query optimizer that the query may perform better with an index to access the data. For the queries where an index is advised, notice that the rows selected by the query are low in comparison to the total number of rows in the table. This is another indication that a table scan may not be optimal. Finally, a long execution time may highlight queries that may be improved by performance tuning.

The next logical step is to look into the index advised optimizer hint. The following query could be used for this:

```

SELECT A.QQTLN, A.QQTFN, A.QQIDXA, A.QQIDXD,
       A.QQIDXX, B.QQOPID, C.QQSTTX
FROM   LIB/QQQ3000 A INNER JOIN LIB/QQQ3014 B
       ON (A.QQJFLD = B.QQJFLD AND
          A.QQUCNT = B.QQUCNT)
       LEFT OUTER JOIN LIB/QQQ1000 C
       ON (A.QQJFLD = C.QQJFLD AND
          A.QQUCNT = C.QQUCNT)
WHERE  A.QQIDXA = 'Y'
    
```

There are two slight modifications from the first example. First, the selected columns have been changed. Most important is the selection of column QQIDXD that contains a list of possible key columns to use when creating the index suggested by the query optimizer. Second, the query selection limits the output to those table scan queries where the optimizer advises that an index be created (A.QQIDXA = 'Y'). Table 5 shows what the results might look like.

Table 5. Output with Recommended Key Columns

Lib Name	Table Name	Index Advised	Advised Key columns	Advised Primary Key	Query OPNID	Statement Text
LIB1	TBL1	Y	FLD1	1		SELECT * FROM LIB1/TBL1 WHERE FLD1 = 'A'
LIB1	TBL1	Y	FLD1, FLD2	1		SELECT * FROM LIB1/TBL1 WHERE FLD1 = 'B' AND FLD2 > 9000
LIB1	TBL4	Y	FLD1, FLD4	1	QRY04	

At this point you should determine whether it makes sense to create a permanent index as advised by the optimizer. In this example, creating one index over LIB1/TBL1 would satisfy all three queries since each use a primary or left-most key column of FLD1. By creating one index over LIB1/TBL1 with key columns FLD1, FLD2, there is potential to improve the performance of the second query even more. The frequency these queries are run and the overhead of maintaining an additional index over the table should be considered when deciding whether or not to create the suggested index.

If you create a permanent index over FLD1, FLD2 the next sequence of steps would be to:

1. Start the performance monitor again
2. Re-run the application
3. End the performance monitor
4. Re-evaluate the data.

It is likely that the three index-advised queries are no longer performing table scans.

Additional database monitor examples

The following are additional ideas or examples on how to extract information from the performance monitor statistics. All of the examples assume data has been collected in LIB/PERFDATA and the documented logical files have been created.

1. How many queries are performing dynamic replans?

```
SELECT COUNT(*)
FROM LIB/QQQ1000
WHERE QQDYNR <> 'NA'
```

2. What is the statement text and the reason for the dynamic replans?

```
SELECT QQDYNR, QQSTTX
FROM LIB/QQQ1000
WHERE QQDYNR <> 'NA'
```

Note: You have to refer to the description of column QQDYNR for definitions of the dynamic replan reason codes.

3. How many indexes have been created over LIB1/TBL1?

```
SELECT COUNT(*)
FROM LIB/QQQ3002
WHERE QQTLN = 'LIB1'
AND QQTFN = 'TBL1'
```

4. What key columns are used for all indexes created over LIB1/TBL1 and what is the associated SQL statement text?

```
SELECT A.QQTLN, A.QQTFN, A.QQIDX, B.QQSTTX
FROM LIB/QQQ3002 A, LIB/QQQ1000 B
WHERE A.QQJFLD = B.QQJFLD
AND A.QQUCNT = B.QQUCNT
AND A.QQTLN = 'LIB1'
AND A.QQTFN = 'TBL1'
```

Note: This query shows key columns only from queries executed using SQL.

5. What key columns are used for all indexes created over LIB1/TBL1 and what was the associated SQL statement text or query open ID?

```
SELECT A.QQTLN, A.QQTFN, A.QQIDX,
B.QQOPID, C.QQSTTX
FROM LIB/QQQ3002 A INNER JOIN LIB/QQQ3014 B
ON (A.QQJFLD = B.QQJFLD AND
A.QQUCNT = B.QQUCNT)
LEFT OUTER JOIN LIB/QQQ1000 C
ON (A.QQJFLD = C.QQJFLD AND
A.QQUCNT = C.QQUCNT)
WHERE A.QQTLN = 'LIB1'
AND A.QQTFN = 'TBL1'
```

Note: This query shows key columns from all queries on the server.

6. What types of SQL statements are being performed? Which are performed most frequently?

```
SELECT QQSTOP, COUNT(*)
FROM LIB/QQQ1000
GROUP BY QQSTOP
ORDER BY 2 DESC
```

7. Which SQL queries are the most time consuming? Which user is running these queries?


```

SELECT (QQETIM - QQSTIM), QQUSER, QQSTTX
FROM LIB/QQQ1000
ORDER BY 1 DESC

```

8. Which queries are the most time consuming?

```

SELECT (A.QQTTIM + B.QQCLKT), A.QQOPID, C.QQSTTX
FROM LIB/QQQ3014 A LEFT OUTER JOIN LIB/QQQ3019 B
ON (A.QQJFLD = B.QQJFLD AND
A.QQUCNT = B.QQUCNT)
LEFT OUTER JOIN LIB/QQQ1000 C
ON (A.QQJFLD = C.QQJFLD AND
A.QQUCNT = C.QQUCNT)
ORDER BY 1 DESC

```

Note: This example assumes detail data has been collected into row QQQ3019.

9. Show the data for all SQL queries with the data for each SQL query logically grouped together.

```

SELECT A.*
FROM LIB/PERFDATA A, LIB/QQQ1000 B
WHERE A.QQJFLD = B.QQJFLD
AND A.QQUCNT = B.QQUCNT

```

Note: This might be used within a report that will format the interesting data into a more readable format. For example, all reason code columns could be expanded by the report to print the definition of the reason code (that is, physical column QQRCD = 'T1' means a table scan was performed because no indexes exist over the queried table).

10. How many queries are being implemented with temporary tables because a key length of greater than 2000 bytes or more than 120 key columns was specified for ordering?

```

SELECT COUNT(*)
FROM LIB/QQQ3004
WHERE QQRCD = 'F6'

```

11. Which SQL queries were implemented with nonreusable ODPs?

```

SELECT B.QQSTTX
FROM LIB/QQQ3010 A, LIB/QQQ1000 B
WHERE A.QQJFLD = B.QQJFLD
AND A.QQUCNT = B.QQUCNT
AND A.QQODPI = 'N'

```

12. What is the estimated time for all queries stopped by the query governor?

```

SELECT QQEPT, QQOPID
FROM LIB/QQQ3014
WHERE QQGVNS = 'Y'

```

Note: This example assumes detail data has been collected into row QQQ3019.

13. Which queries estimated time exceeds actual time?

```

SELECT A.QQEPT, (A.QQTTIM + B.QQCLKT), A.QQOPID,
C.QQTTIM, C.QQSTTX
FROM LIB/QQQ3014 A LEFT OUTER JOIN LIB/QQQ3019 B
ON (A.QQJFLD = B.QQJFLD AND
A.QQUCNT = B.QQUCNT)
LEFT OUTER JOIN LIB/QQQ1000 C
ON (A.QQJFLD = C.QQJFLD AND
A.QQUCNT = C.QQUCNT)
WHERE A.QQEPT/1000 > (A.QQTTIM + B.QQCLKT)

```

Note: This example assumes detail data has been collected into row QQQ3019.

14. Should a PTF for queries that perform UNION exists be applied. It should be applied if any queries are performing UNION. Do any of the queries perform this function?

```

SELECT COUNT(*)
FROM QQQ3014
WHERE QQUNIN = 'Y'

```

Note: If result is greater than 0, the PTF should be applied.

15. You are a system administrator and an upgrade to the next release is planned. A comparison between the two releases would be interesting.
 - Collect data from your application on the current release and save this data in LIB/CUR_DATA
 - Move to the next release
 - Collect data from your application on the new release and save this data in a different table: LIB/NEW_DATA
 - Write a program to compare the results. You will need to compare the statement text between the rows in the two tables to correlate the data.

Gather statistics about your queries with memory-resident database monitor APIs

The Memory-Resident Database Monitor (DBMon) is a tool that provides another method for monitoring database performance. This tool is only intended for SQL performance monitoring and is useful for programmers and performance analysts. The DBMon monitor, with the help of a new set of APIs, takes database monitoring statistics and manages them for the user in memory. This memory-based monitor reduces CPU overhead as well as resulting table sizes.

The Start Database Monitor (STRDBMON) can constrain server resources when collecting performance information. This overhead is mainly attributed to the fact that performance information is written directly to a database table as the information is collected. The memory-based collection mode reduces the server resources consumed by collecting and managing performance results in memory. This allows the monitor to gather database performance statistics with a minimal impact to the performance of the server as whole (or to the performance of individual SQL statements).

The DBMon monitor collects much of the same information as the STRDBMON monitor, but the performance statistics are kept in memory. At the expense of some detail, information is summarized for identical SQL statements to reduce the amount of information collected. The objective is to get the statistics to memory as fast as possible while deferring any manipulation or conversion of the data until the performance data is dumped to a result table for analysis.

The DBMon monitor is not meant to replace the STRDBMON monitor. There are circumstances where the loss of detail in the DBMon monitor will not be sufficient to fully analyze an SQL statement. In these cases, the STRDBMON monitor should still be used.

The DBMon monitor manages the data in memory, combining and accumulating the information into a series of row formats. This means that for each unique SQL statement, information is accumulated from each run of the statement and the detail information is only collected for the most expensive statement execution.

Each SQL statement is identified by the monitor according to the:

- statement name
- package (or program)
- library that contains the prepared statement
- cursor name that is used

For pure dynamic statements, the statement text is kept in a separate space and the statement identification will be handled internally via a pointer.

While this system avoids the significant overhead of writing each SQL operation to a table, keeping statistics in memory comes at the expense of some detail. Your objective should be to get the statistics to memory as fast as possible, then reserve time for data manipulation or data conversion later when you dump data to a table.

The DBMon manages the data that is in memory by combining and accumulating the information into the new row formats. Therefore, for each unique SQL statement, information accumulates from each running of the statement, and the server only collects detail information for the most expensive statement execution.

Each SQL statement is identified by the monitor by the statement name, the package (or program) and library that contains the prepared statement and the cursor name that is used. For pure dynamic statements:

- Statement text is kept in a separate space and
- Statement identification is handled internally via a pointer.

API support for the DBMon monitor

A new set of APIs enable support for the DBMon monitor. An API supports each of the following activities:

- Start the new monitor
- Dump statistics to tables
- Clear the monitor data from memory
- Query the monitor status
- End the new monitor

When you start the new monitor, information is stored in the local address space of each job that the system monitors. As each statement completes, the system moves information from the local job space to a common system space. If more statements are executed than can fit in this amount of common system space, the system drops the statements that have not been executed recently.

| The following topics provide detailed information on the database monitor APIs:

- | • “Memory-resident database monitor external API description”
- | • “Memory-resident database monitor external table description”
- | • “Sample SQL queries” on page 80
- | • “Memory-resident database monitor row identification” on page 80

Memory-resident database monitor external API description

The memory-resident database monitor is controlled by a set of APIs. For additional information, see the OS/400 APIs information in the **Programming** category of the iSeries Information Center.

Table 6. External API Description

QQQSSDBM	API to start the SQL monitor
QQQCSDBM	API to clear SQL monitor memory
QQQDSDBM	API to dump the contents of the SQL monitor to table
QQQESDBM	API to end the SQL monitor
QQQQSDBM	API to query status of the database monitor

Memory-resident database monitor external table description

The memory resident database monitor uses its own set of tables instead of using the single table with logical files that the STRDBMON monitor uses. The memory resident database monitor tables closely match the suggested logical files of the STRDBMON monitor.

Note: Starting with Version 4 Release 5, newly captured information will not appear through the memory resident monitor, and although the file format for these files did not change, the file formats for the file based monitor did change.

Table 7. External table Description

QAQQRYI	Query (SQL) information
QAQQTEXT	SQL statement text
QAQQ3000	Table scan
QAQQ3001	Index used
QAQQ3002	Index created
QAQQ3003	Sort
QAQQ3004	Temporary table
QAQQ3007	Optimizer time out/ all indexes considered
QAQQ3008	Subquery
QAQQ3010	Host variable values

Sample SQL queries

As with the STRDBMON monitor, it is up to the user to extract the information from the tables in which all of the monitored data is stored. This can be done through any query interface that the user chooses.

If you are using iSeries Navigator with the support for the SQL Monitor, you have the ability to analyze the results direct through the graphical user interface. There are a number of shipped queries that can be used or modified to extract the information from any of the tables. For a list of these queries, go to [Common queries on analysis of DB Performance Monitor data the DB2 UDB for iSeries website](#) .

Memory-resident database monitor row identification

The join key column QQKEY simplifies the joining of multiple tables together. This column replaces the join field (QQJFLD) and unique query counters (QQCNT) that the database monitor used. The join key column contains a unique identifier that allows all of the information for this query to be received from each of the tables.

This join key column does not replace all of the detail columns that are still required to identify the specific information about the individual steps of a query. The Query Definition Template (QDT) Number or the Subselect Number identifies information about each detailed step. Use these columns to identify which rows belong to each step of the query process:

- QQQDTN - Query Definition Template Number
- QQQDTL - Query Definition Template Subselect Number (Subquery)
- QQMATN - Materialized Query Definition Template Number (View)
- QQMATL - Materialized Query Definition Template Subselect Number (View w/ Subquery)
- QQMATULVL - Materialized Query Definition Template Union Number (View w/Union)

Use these columns when the monitored query contains a subquery, union, or a view operation. All query types can generate multiple QDT's to satisfy the original query request. The server uses these columns to separate the information for each QDT while still allowing each QDT to be identified as belonging to this original query (QQKEY).

Monitoring your database performance using SQL Performance monitors in iSeries Navigator

The SQL performance Monitor allows you to keep track of the resources that your SQL statements use. You can monitor specific resources or many resources. The information on resource use can help you determine whether your system and your SQL statements are performing as they should, or whether they need fine tuning. There are two types of monitors that you can choose to monitor your resources:

Summary SQL performance monitor

The summary SQL performance monitor is the iSeries Navigator version of the Memory Resident Database monitor, found on the system interface. As the name implies, this monitor resides in memory and only retains a summary of the data collected. When the monitor is paused or ended, this data is written to a hard disk and can be analyzed. Because the monitor stores its information in memory, the performance impact to your system is minimized. However, you do lose some of the detail. For more details, see “Gather statistics about your queries with memory-resident database monitor APIs” on page 78.

Detailed SQL performance monitor

The detailed SQL performance monitor is the iSeries Navigator version of the database monitor, found on the system interface. This monitor save detailed data in real time to a hard disk and does not need to be paused or ended in order to analyze the results. You can also choose to run a Visual Explain based on the data gathered by the monitor. Since this monitor does save data in real time, it may have a performance impact on your system. For more details, see “Start Database Monitor (STRDBMON) command” on page 70.

For more information about using SQL performance monitors, see the following topics:

- “Creating an SQL performance monitor”
- “Saving SQL performance monitor data (pausing a monitor)” on page 82
- “Analyzing SQL performance monitor data” on page 82

Creating an SQL performance monitor

Creating a new SQL performance monitor creates a new instance of a monitor on your system. You can have multiple instances of monitors running on you system at one time, however, there can only be one monitor instance monitoring all jobs. When collecting information for all jobs, the monitor will collect on previously started jobs or new jobs started after the monitor is created. However, when you end a performance monitor, the instance of the monitor is terminated and cannot be continued whereas a paused monitor can be restarted. To create an SQL performance monitor:

1. In the **iSeries Navigator** window, expand your server → **Database**.
2. Right-click **SQL Performance Monitor** and select **New**.
3. Select **Summary** or **Detailed**.
4. Specify the name you want to give the monitor in the **Name** field.
5. Specify the library in which you want to save the information that the monitor gathers in the **Library** field.
6. If you want to specify the maximum amount of system memory that the monitor is allowed to consume, specify the size in MB in the **Storage (MB)** field.
7. Click the **Monitored Jobs** tab.
8. If you want to monitor all of the available jobs and another monitor is not monitoring all jobs, select **All**. Only one monitor can monitor all jobs at a time.
9. If you want to monitor only certain jobs, select a job that you want to monitor in the **Available jobs** list and click **Select**. Repeat this step for each job that you want to monitor. Individual jobs can be monitored by only one active monitor at a time.
10. If you have selected a job that you do not want to monitor or if a job you have selected is already being monitored, select that job in the **Selected jobs** list and click **Remove**.
11. If you selected Summary monitor, click the **Data to Collect** tab.
12. On the **Data to Collect** tab, select the types of data that you want to collect. If you want to collect all types of data, click **Select All**.
13. Click **OK**. The performance monitor starts and runs until it is ended or paused.

Saving SQL performance monitor data (pausing a monitor)

Unlike the detailed SQL performance monitor that saves data in real time as it runs, the data that the summary SQL monitor collected is stored in memory and must be saved in order to use it. To save a summary SQL performance monitor:

1. In the **iSeries Navigator** window, expand your server → **Database** → **SQL Performance Monitors**.
2. Select a performance monitor in the right pane.
3. If you may want to start this monitor again, right-click the monitor that you want to save and select **Pause**.
4. If you do not want to start this monitor again, right-click the monitor that you want to save and select **End**.

The data that the monitor collected is saved.

Analyzing SQL performance monitor data

Once you have gathered resources using your performance monitor, you will want to analyze the data that the monitor has collected. If you are using a summary SQL performance monitor, you will first need to pause or end the monitor. A detailed SQL performance monitor can still be running.

1. In the **iSeries Navigator** window, expand your server → **Database** → **SQL Performance Monitor**.
2. Right-click on a performance monitor in the right pane, and select **Analyze Results**.
3. Select the collection period for which you want to view data from the **Collection period** list. Select the data you want to view. The **Collection period** list applies to all monitors, but only Summary monitors can have multiple periods. Imported monitors display **Information Not Available**.
4. Select the data that you want to view.
5. If you want to view the data in different ways, click the tab whose views you want to use and select the queries that you want. For information about each type of view, select the view and press F1.
6. If you want to modify the query, click **Modify Query Selected**. You must run the query before exiting the **Run SQL Scripts** window. To save this query, you must save it from the **Run SQL Scripts** window.
7. After you have selected each view that you want, click **OK** or **View Results**.

View the effectiveness of your queries with Visual Explain

You can use the **Visual Explain** tool with iSeries Navigator to create a query graph that graphically displays the implementation of an SQL statement. You can use this tool to see information about both static and dynamic SQL statements. Visual Explain supports the following types of SQL statements: SELECT, INSERT, UPDATE, and DELETE.

Visual Explain can be used to find the most expensive or most time consuming operations in your query. You can improve query performance by:

- Rewriting your SQL statement
- Changing query attributes and environment settings
- Creating indexes recommended by the query optimizer

You can use Visual Explain to:

- View the statistics that were used during optimization
- Determine whether or not an index was used to access the data. If an index was not used, Visual Explain can help you determine which columns might benefit from being indexed.
- View the effects of performing tuning techniques by comparing the before and after pictures of the implementation.

- Obtain information about each operation (icon) in the query graph, including the total estimated cost and estimated number of rows returned.

Visual Explain works against the data stored in the database monitor table. That is, a table that contains the results from executing the STRDBMON command. It does not work with tables resulting from the memory-resident monitor. There are two ways to invoke the Visual Explain tool. The first, and most common, is through iSeries Navigator. The second is through the Visual Explain API. See the OS/400 APIs information in the Programming category of the iSeries Information Center. You can launch Visual Explain either of the following windows in iSeries Navigator:

- Expand the list of available SQL Performance Monitors. Right-click on an SQL Performance Monitor and choose the **List explainable statements** option. This opens the **Explainable statements for SQL performance monitor** window.
- Highlight (left click) on the SQL statement that you wish to explain and click the Run Visual Explain button.
- Enter an SQL statement in the **Run SQL Scripts** window. Select the statement and choose **Explain...** from the context menu, or select **Run and Explain...** from the **Visual Explain** pull-down menu.

The database monitor table (results of running the STRDBMON command on the server) can be explained through iSeries Navigator. First you must import the database monitor table into iSeries Navigator. To do this, right-click on the SQL Performance Monitors and choose the **Import** option. Specify a name for the performance monitor (name it will be known by within iSeries Navigator) and the qualified name of the database monitor table. Be sure to select Detailed as the type of monitor. Detailed implies the file-based (STRDBMON) monitor while Summary implies the memory-resident monitor (which is not supported by Visual Explain). Once the monitor has been imported, follow the steps to launch Visual Explain from within iSeries Navigator.

All the information in the query optimizer debug messages (e.g., table name, estimated number of rows, index advised information) is shown through Visual Explain in the Attributes section. In fact, Visual Explain and the file-based monitor (STRDBMON) contain more information than what is contained in the optimizer debug messages:

- You can see the long and short name of tables being queried.
- You can see which columns were used in the selection to create the temporary index (remember, most temporary indexes are sparse or select/omit indexes).
- You can see attributes about the environment when the query is executed. For example, it will show what the ALWCPYDTA setting was.
- It will show the name and the size of the memory pool
- It will show which query INI file was used.

Another benefit of Visual Explain is its ability to differentiate the subselects within the query. If you execute a query which contains a subquery it is sometimes difficult to determine which optimizer debug messages belong to which subselect, the outer query or the subquery. Visual Explain handles all this.

Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command

You can modify different types of attributes of the queries that you will execute during a certain job with the CHGQRYA command, or by using the iSeries Navigator interface. The types of attributes that you can modify include:

- Predictive Query Governor
- Query Parallelism
- Asynchronous Job
- Apply CHGQRYA to remote

- Query options file parameter

Before the server starts a query, the server checks the query time limit against the estimated elapsed query time. The server also uses a time limit of zero to optimize performance on queries without having to run through several iterations.

You can check the inquiry message CPA4259 for the predicted runtime and for what operations the query will perform. If the query is cancelled, debug messages will still be written to the job log.

The DB2 Universal Database for iSeries Predictive Query Governor can stop the initiation of a query if the query's estimated or predicted runtime (elapsed execution time) is excessive. The governor acts *before* a query is run instead of while a query is running. You can use it in any interactive or batch job on iSeries. You can also use it with all DB2 Universal Database for iSeries query interfaces; it is not limited to use with SQL queries. See "Control long-running queries with the DB2 UDB for iSeries Predictive Query Governor" on page 93 for details.

Control queries dynamically with the query options file QAQQINI

The query options file QAQQINI support provides the ability to dynamically modify or override the environment in which queries are executed through the CHGQRYA command and the QAQQINI file.

The query options file QAQQINI is used to set some attributes used by the Query Optimizer. For each query that is run the query option values are retrieved from the QAQQINI file in the library specified on the QRYOPTLIB parameter of the CHGQRYA CL command and used to optimize or implement the query.

Environmental attributes that you can modify through the QAQQINI file include:

- | • APPLY_REMOTE
- | • ASYNC_JOB_USAGE
- | • COMMITMENT_CONTROL_LOCK_LIMIT
- | • FORCE_JOIN_ORDER
- | • IGNORE_LIKE_REDUNDANT_SHIFTS
- | • MESSAGES_DEBUG
- | • OPEN_CURSOR_CLOSE_COUNT
- | • OPEN_CURSOR_THRESHOLD
- | • OPTIMIZE_STATISTIC_LIMITATION
- | • OPTIMIZATION_GOAL
- | • PARALLEL_DEGREE
- | • PARAMETER_MARKER_CONVERSION
- | • QUERY_TIME_LIMIT
- | • REOPTIMIZE_ACCESS_PLAN
- | • SQLSTANDARDS_MIXED_CONSTANT
- | • SQL_SUPPRESS_WARNINGS
- | • SQL_TRANSLATE_ASCII_TO_JOB
- | • STAR_JOIN
- | • SYSTEM_SQL_STATEMENT_CACHE
- | • UDF_TIME_OUT
- | • VISUAL_EXPLAIN_DIAGRAM

To specify the library that currently holds or will contain the query options file QAQQINI, see "Specifying the QAQQINI file" on page 85.

To create your own QAQQINI file, see “Creating the QAQQINI query options file”

Specifying the QAQQINI file

Use the CHGQRYA command with the QRYOPLIB (query options library) parameter to specify which library currently contains or will contain the query options file QAQQINI. The query options file will be retrieved from the library specified on the QRYOPLIB parameter for each query and remains in effect for the duration of the job or user session, or until the QRYOPLIB parameter is changed by the CHGQRYA command.

If the CHGQRYA command is not issued or is issued but the QRYOPLIB parameter is not specified, the library QUSRSYS is searched for the existence of the QAQQINI file. If a query options file is not found for a query, no attributes will be modified. Since the server is shipped with no INI file in QUSRSYS, you may receive a message indicating that there is no INI file. This message is not an error but simply an indication that a QAQQINI file that contains all default values is being used. The initial value of the QRYOPLIB parameter for a job is QUSRSYS.

Creating the QAQQINI query options file

Each server is shipped with a QAQQINI template file in library QSYS. The QAQQINI file in QSYS is to be used as a template when creating all user specified QAQQINI files. To create your own QAQQINI file, use the CRTDUPOBJ command to create a copy of the QAQQINI file in the library that will be specified on the CHGQRYA QRYOPLIB parameter. The file name must remain QAQQINI, for example:

```
CRTDUPOBJ OBJ(QAQQINI)
          FROMLIB(QSYS)
          OBJTYPE(*FILE)
          TOLIB(MYLIB)
          DATA(*YES)
```

System-supplied triggers are attached to the QAQQINI file in QSYS therefore it is imperative that the only means of copying the QAQQINI file is through the CRTDUPOBJ CL command. If another means is used, such as CPYF, then the triggers may be corrupted and an error will be signaled that the options file cannot be retrieved or that the options file cannot be updated.

Because of the trigger programs attached to the QAQQINI file, the following CPI321A informational message will be displayed six times in the job log when the CRTDUPOBJ CL is used to create the file. This is not an error. It is only an informational message.

CPI321A Information Message: Trigger QSYS_TRIG_&1__QAQQINI__00000&N in library &1 was added to file QAQQINI in library &1. The ampersand variables (&1, &N) are replacement variables that contain either the library name or a numeric value.

Note: It is recommended that the file QAQQINI, in QSYS, not be modified. This is the original template that is to be duplicated into QUSRSYS or a user specified library for use.

QAQQINI query options file format

Query Options File:

A			UNIQUE
A	R QAQQINI		TEXT('Query options + file')
A	QQPARM	256A	VARLEN(10) + TEXT('Query+ option parameter') + COLHDG('Parameter')
A	QQVAL	256A	VARLEN(10) + TEXT('Query option + parameter value') + COLHDG('Parameter Value')
A	QQTEXT	1000G	VARLEN(100) + TEXT('Query + option text') +

ALWNULL +
 COLHDG('Query Option' +
 'Text') +
 CCSID(13488) +
 DFT(*NULL)

A K QQPARM

The QAQQINI file shipped in the library QSYS has been pre-populated with the following rows:

Table 8. QAQQINI File Records. Description

QPARM	QVAL
APPLY_REMOTE	*DEFAULT
ASYNC_JOB_USAGE	*DEFAULT
COMMITMENT_CONTROL_LOCK_LIMIT	*DEFAULT
FORCE_JOIN_ORDER	*DEFAULT
IGNORE_LIKE_REDUNDANT_SHIFTS	*DEFAULT
MESSAGES_DEBUG	*DEFAULT
OPEN_CURSOR_CLOSE_COUNT	*DEFAULT
OPEN_CURSOR_THRESHOLD	*DEFAULT
OPTIMIZATION_GOAL	*DEFAULT
OPTIMIZE_STATISTIC_LIMITATION	*DEFAULT
PARALLEL_DEGREE	*DEFAULT
PARAMETER_MARKER_CONVERSION	*DEFAULT
QUERY_TIME_LIMIT	*DEFAULT
REOPTIMIZE_ACCESS_PLAN	*DEFAULT
SQLSTANDARDS_MIXED_CONSTANT	*DEFAULT
SQL_SUPPRESS_WARNINGS	*DEFAULT
SQL_TRANSLATE_ASCII_TO_JOB	*DEFAULT
STAR_JOIN	*DEFAULT
SYSTEM_SQL_STATEMENT_CACHE	*DEFAULT
UDF_TIME_OUT	*DEFAULT
VISUAL_EXPLAIN_DIAGRAM	*DEFAULT

Setting the options within the query options file

The QAQQINI file query options can be modified with the INSERT, UPDATE, or DELETE SQL statements.

For the following examples, a QAQQINI file has already been created in library MyLib. To update an existing row in MyLib/QAQQINI use the UPDATE SQL statement. This example sets MESSAGES_DEBUG = *YES so that the query optimizer will print out the optimizer debug messages:

```
UPDATE MyLib/QAQQINI SET QVAL='*YES'
WHERE QPARM='MESSAGES_DEBUG'
```

To delete an existing row in MyLib/QAQQINI use the DELETE SQL statement. This example removes the QUERY_TIME_LIMIT row from the QAQQINI file:

```
DELETE FROM MyLib/QAQQINI
WHERE QPARM='QUERY_TIME_LIMIT'
```

To insert a new row into MyLib/QAQQINI use the INSERT SQL statement. This example adds the QUERY_TIME_LIMIT row with a value of *NOMAX to the QAQQINI file:

```
INSERT INTO MyLib/QAQQINI
VALUES('QUERY_TIME_LIMIT','*NOMAX','New time limit set by DBAdmin')
```

QAQQINI query options

The following table summarizes the query options that can be specified on the QAQQINI command:

Table 9. Query Options Specified on QAQQINI Command

Parameter	Value	Description
APPLY_REMOTE	*DEFAULT	The default value is set to *NO.
	*NO	The CHGQRYA attributes for the job are not applied to the remote jobs. The remote jobs will use the attributes associated to them on their servers.
	*YES	The query attributes for the job are applied to the remote jobs used in processing database queries involving distributed tables. For attributes where *SYSVAL is specified, the system value on the remote server is used for the remote job. This option requires that, if CHGQRYA was used for this job, the remote jobs must have authority to use the CHGQRYA command.
ASYNC_JOB_USAGE	*DEFAULT	The default value is set to *LOCAL.
	*LOCAL	Asynchronous jobs may be used for database queries that involve only tables local to the server where the database queries are being run. In addition, for queries involving distributed tables, this option allows the communications required to be asynchronous. This allows each server involved in the query of the distributed tables to run its portion of the query at the same time (in parallel) as the other servers.
	*DIST	Asynchronous jobs may be used for database queries that involve distributed tables.
	*ANY	Asynchronous jobs may be used for any database query.
	*NONE	No asynchronous jobs are allowed to be used for database query processing. In addition, all processing for queries involving distributed tables occurs synchronously. Therefore, no inter-system parallel processing will occur.
COMMIT_CONTROL_LOCK_LIMIT	*DEFAULT	*DEFAULT is equivalent to 500,000,000.
	Integer Value	The maximum number of records that can be locked to a commit transaction initiated after setting the new value. The valid integer value is 1–500,000,000.

Table 9. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
FORCE_JOIN_ORDER	*DEFAULT	The default is set to *NO.
	*NO	Allow the optimizer to re-order join tables.
	*SQL	Only force the join order for those queries that use the SQL JOIN syntax. This mimics the behavior for the optimizer prior to V4R4M0.
	*PRIMARY nnn	Only force the join position for the file listed by the numeric value nnn (nnn is optional and will default to 1) into the primary position (or dial) for the join. The optimizer will then determine the join order for all of the remaining files based upon cost.
	*YES	Do not allow the query optimizer to re-order join tables as part of its optimization process. The join will occur in the order in which the tables were specified in the query.
IGNORE_LIKE_REDUNDANT_SHIFTS	*DEFAULT	The default value is set to *ALWAYS.
	*ALWAYS	When processing the SQL LIKE predicate or OPNQRYF command %WLDCRD built-in function, redundant shift characters are ignored for DBCS-Open operands. Note that this option restricts the query optimizer from using an index to perform key row positioning for SQL LIKE or OPNQRYF %WLDCRD predicates involving DBCS-Open, DBCS-Either, or DBCS-Only operands.
	*OPTIMIZE	When processing the SQL LIKE predicate or the OPNQRYF command %WLDCRD built-in function, redundant shift characters may or may not be ignored for DBCS-Open operands depending on whether an index is used to perform key row positioning for these predicates. Note that this option will enable the query optimizer to consider key row positioning for SQL LIKE or OPNQRYF %WLDCRD predicates involving DBCS-Open, DBCS-Either, or DBCS-Only operands.
MESSAGE_DEBUG	*DEFAULT	The default is set to *NO.
	*NO	No debug messages are to be displayed.
	*YES	Issue all Query Optimizer debug messages.
OPEN_CURSOR_CLOSE_COUNT	*DEFAULT	*DEFAULT is equivalent to 0. See Integer Value for details.
	Integer Value	OPEN_CURSOR_CLOSE_COUNT is used in conjunction with OPEN_CURSOR_THRESHOLD to manage the number of open cursors within a job. If the number of open cursors, which includes open cursors and pseudo-closed cursors, reaches the value specified by the OPEN_CURSOR_THRESHOLD, pseudo-closed cursors are hard (fully) closed with the least recently used cursors being closed first. This value determines the number of cursors to be closed. The valid values for this parameter are 1 - 65536. The value for this parameter should be less than or equal to the number in the OPEN_CURSOR_THRESHOLD parameter. This value is ignored if OPEN_CURSOR_THRESHOLD is *DEFAULT. If OPEN_CURSOR_THRESHOLD is specified and this value is *DEFAULT, the number of cursors closed is equal to OPEN_CURSOR_THRESHOLD multiplied by 10 percent and rounded up to the next integer value.

Table 9. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
OPEN_CURSOR_THRESHOLD	*DEFAULT	*DEFAULT is equivalent to 0. See Integer Value for details.
	Integer Value	OPEN_CURSOR_THRESHOLD is used in conjunction with OPEN_CURSOR_CLOSE_COUNT to manage the number of open cursors within a job. If the number of open cursors, which includes open cursors and pseudo-closed cursors, reaches this threshold value, pseudo-closed cursors are hard (fully) closed with the least recently used cursors being closed first. The number of cursors to be closed is determined by OPEN_CURSOR_CLOSE_COUNT. The valid user-entered values for this parameter are 1 - 65536. Having a value of 0 (default value) would indicate that there is no threshold and hard closes will not be forced on the basis of the number of open cursors within a job.
OPTIMIZATION_GOAL	*DEFAULT	Optimization goal is determined by the interface (ODBC, SQL precompiler options, OPTIMIZE FOR nnn ROWS clause).
	*FIRSTIO	All queries will be optimized with the goal of returning the first page of output as fast as possible. This goal works well when the control of the output is controlled by a user who is most likely to abort the query after viewing the first page of output data. Queries coded with an OPTIMIZE FOR nnn ROWS clause will honor the goal specified by the clause.
	*ALLIO	All queries will be optimized with the goal of running the entire query to completion in the shortest amount of elapsed time. This is a good option for when the output of a query is being written to a file or report, or the interface is queuing the output data. Queries coded with an OPTIMIZE FOR nnn ROWS clause will honor the goal specified by the clause.
OPTIMIZE_STATISTIC_LIMITATION	*DEFAULT	The amount of time spent in gathering index statistics is determined by the query optimizer.
	*NO	No index statistics will be gathered by the query optimizer. Default statistics will be used for optimization. (Use this option sparingly.)
	*PERCENTAGE integer value	Specifies the maximum percentage of the index that will be searched while gathering statistics. Valid values for are 1 to 99.
	*MAX_NUMBER_OF_RECORDS_ALLOWED integer value	Specifies the largest table size, in number of rows, for which gathering statistics is allowed. For tables with more rows than the specified value, the optimizer will not gather statistics and will use default values.

Table 9. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
PARALLEL_DEGREE	*DEFAULT	The default value is set to *SYSVAL.
	*SYSVAL	The processing option used is set to the current value of the system value, QQRYDEGREE.
	*IO	Any number of tasks can be used when the database query optimizer chooses to use I/O parallel processing for queries. SMP parallel processing is not allowed.
	*OPTIMIZE	The query optimizer can choose to use any number of tasks for either I/O or SMP parallel processing to process the query or database file keyed access path build, rebuild, or maintenance. SMP parallel processing is used only if the system feature, DB2 Symmetric Multiprocessing for OS/400, is installed. Use of parallel processing and the number of tasks used is determined with respect to the number of processors available in the server, this job has a share of the amount of active memory available in the pool in which the job is run, and whether the expected elapsed time for the query or database file keyed access path build or rebuild is limited by CPU processing or I/O resources. The query optimizer chooses an implementation that minimizes elapsed time based on the job has a share of the memory in the pool.
	*MAX	The query optimizer chooses to use either I/O or SMP parallel processing to process the query. SMP parallel processing will only be used if the system feature, DB2 Symmetric Multiprocessing for OS/400, is installed. The choices made by the query optimizer are similar to those made for parameter value *OPTIMIZE except the optimizer assumes that all active memory in the pool can be used to process the query or database file keyed access path build, rebuild, or maintenance.
	*NONE	No parallel processing is allowed for database query processing or database table index build, rebuild, or maintenance.
	*NUMBER_OF_TASKS nn	Indicates the maximum number of tasks that can be used for a single query. The number of tasks will be capped off at either this value or the number of disk arms associated with the table.
PARAMETER_MARKER_CONVERSION	*DEFAULT	The default value is set to *YES.
	*NO	Constants cannot be implemented as parameter markers.
	*YES	Constants can be implemented as parameter markers.
QUERY_TIME_LIMIT	*DEFAULT	The default value is set to *SYSVAL.
	*SYSVAL	The query time limit for this job will be obtained from the system value, QQRYTIMLMT.
	*NOMAX	There is no maximum number of estimated elapsed seconds.
	integer value	Specifies the maximum value that is checked against the estimated number of elapsed seconds required to run a query. If the estimated elapsed seconds is greater than this value, the query is not started. Valid values range from 0 through 2147352578.

Table 9. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
REOPTIMIZE_ACCESS_PLAN	*DEFAULT	Do not force the existing query to be reoptimized. However, if the optimizer determines that optimization is necessary, the query will be reoptimized.
	*NO	Do not force the existing query to be reoptimized. However, if the optimizer determines that optimization is necessary, the query will be reoptimized.
	*YES	Force the existing query to be reoptimized.
	*FORCE	Force the existing query to be reoptimized.
	*ONLY_REQUIRED	Do not allow the plan to be reoptimized for any subjective reasons. For these cases, continue to use the existing plan since it is still a valid workable plan. This may mean that you may not get all of the performance benefits that a reoptimization plan may derive. Subjective reasons include, file size changes, new indexes, etc. Non-subjective reasons include, deletion of an index used by existing access plan, query file being deleted and recreated, etc.
SQLSTANDARDS_MIXED_CONSTANT	*DEFAULT	The default value is set to *YES.
	*YES	SQL IGC constants will be treated as IGC-OPEN constants.
	*NO	If the data in the IGC constant only contains shift-out DBCS-data shift-in, then the constant will be treated as IGC-ONLY, otherwise it will be treated as IGC-OPEN.
SQL_SUPPRESS_WARNINGS	*DEFAULT	The default value is set to *NO.
	*YES	Examine the SQLCODE in the SQLCA after execution of a statement. If the SQLCODE > 0, then alter the SQLCA so that no warning is returned to the caller. Set the SQLCODE to 0, the SQLSTATE to '00000' and SQLWARN to ' '.
	*NO	Specifies that SQL warnings will be returned to the caller.
SQL_TRANSLATE_ASCII_TO_JOB	*DEFAULT	The default value is set to *NO.
	*YES	Translate ASCII SQL statement text to the CCSID of the iSeries job.
	*NO	Translate ASCII SQL statement text to the EBCDIC CCSID associated with the ASCII CCSID.

Table 9. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
STAR_JOIN	*DEFAULT	The default value is set to *NO
	*NO	The EVI Star Join optimization support is not enabled.
	*FORCE Integer Value	The EVI Star Join optimization algorithm will be attempted for all hash join queries. For those hash join steps where Distinct List selection exists over a column with an EVI created over it, the optimizer will allow those EVIs to be added to the plan without regard to their cost up to the Nth (i.e., Integer Value) index. The Integer Value indicates how many indexes will be allowed into the plan chosen by the optimizer. The optimizer will keep allowing EVIs built over Distinct List selection until either no more indexes exist or the Integer Value is reached. The acceptable range of values for Integer Value are between 1 and 65534. If no value is specified for the Integer Value, then a value of 65535 will be used.
	*COST	Allow query optimization to consider (cost) the usage of EVI Star Join support. The determination of whether or not the Distinct List selection is used will be determined by the optimizer based on how much benefit can be derived from using that selection.
SYSTEM_SQL_STATEMENT_CACHE	*DEFAULT	The default value is set to *YES.
	*YES	Examine the SQL system-wide statement cache when an SQL prepare request is processed. If a matching statement already exists in the cache, use the results of that prepare. This allows the application to potentially have better performing prepares.
	*NO	Specifies that the SQL system-wide statement cache should not be examined when processing an SQL prepare request. Access plans and QDTs will be built from scratch.
UDF_TIME_OUT	*DEFAULT	The amount of time to wait is determined by the database. The default is 30 seconds.
	*MAX	The maximum amount of time that the database will wait for the UDF to finish.
	integer value	Specify the number of seconds that the database should wait for a UDF to finish. If the value given exceeds the database maximum wait time, the maximum wait time will be used by the database. Minimum value is 1 and maximum value is system defined.

Table 9. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
VISUAL_EXPLAIN_ DIAGRAM	*DEFAULT	The default is set to *BASIC.
	*BASIC	If multiple access methods are performed through one index, this option will show only one data access method and one icon to represent that data access method. It will not show the data access method performed when creating any temporary indexes.
	*DETAIL	If multiple access methods are performed through one index, this option will show an icon for each data access method performed. Similarly, this option will show the data access method that was performed when creating a temporary index.

QAQQINI query options file authority requirements

QAQQINI is shipped with a *PUBLIC *USE authority. This allows users to view the query options file, but not change it. It is recommended that only the system or database administrator have *CHANGE authority to the QAQQINI query options file.

The query options file, which resides in the library specified on the CHGQRYA CL command QRYOPTLIB parameter, is always used by the query optimizer. This is true even if the user has no authority to the query options library and file. This provides the system administrator with an additional security mechanism.

When the QAQQINI file resides in the library QUSRSYS the query options will effect all of the query users on the server. To prevent anyone from inserting, deleting, or updating the query options, the system administrator should remove update authority from *PUBLIC to the file. This will prevent users from changing the data in the file.

When the QAQQINI file resides in a user library and that library is specified on the QRYOPTLIB parameter of the CHGQRYA command, the query options will effect all of the queries run for that user's job. To prevent the query options from being retrieved from a particular library the system administrator can revoke authority to the CHGQRYA CL command.

QAQQINI file system supplied triggers

The query options file QAQQINI file uses a system-supplied trigger program in order to process any changes made to the file. A trigger cannot be removed from or added to the file QAQQINI.

If an error occurs on the update of the QAQQINI file (an INSERT, DELETE, or UPDATE operation), the following SQL0443 diagnostic message will be issued:

Trigger program or external routine detected an error.

Control long-running queries with the DB2 UDB for iSeries Predictive Query Governor

The DB2 Universal Database for iSeries Predictive Query Governor can stop the initiation of a query if the estimated or predicted run time (elapsed execution time) for the query is excessive. The governor acts *before* a query is run instead of while a query is run. The governor can be used in any interactive or batch job on the iSeries. It can be used with all DB2 Universal Database for iSeries query interfaces and is not limited to use with SQL queries.

The ability of the governor to predict and stop queries before they are started is important because:

- Operating a long-running query and abnormally ending the query before obtaining any results wastes server resources.

- Some operations within a query cannot be interrupted by the End Request (ENDRQS) CL command. The creation of a temporary index or a query using a column function without a GROUP BY clause are two examples of these types of queries. It is important to not start these operations if they will take longer than the user wants to wait.

The governor in DB2 Universal Database for iSeries is based on the estimated runtime for a query. If the query's estimated runtime exceeds the user defined time limit, the initiation of the query can be stopped.

To define a time limit for the governor to use, do one of the following:

- Use the Query Time Limit (QRYTIMLMT) parameter on the Change Query Attributes (CHGQRYA) CL command. This is the first place where the query optimizer attempts to find the time limit.
- Set the Query Time Limit option in the query options file. This is the second place where the query optimizer attempts to find the time limit.
- Set the QQRYTIMLMT system value. Allow each job to use the value *SYSVAL on the CHGQRYA CL command, and set the query options file to *DEFAULT. This is the third place where the query optimizer attempts to find the time limit.

| See "How the query governor works" for details on how the query governor works in conjunction with query optimizer.

| Before using the predictive query governor, you should see "Query governor implementation considerations" on page 95, "Query governor considerations for user applications: Setting the time limit" on page 95, and "Controlling the default reply to the query governor inquiry message" on page 95 for information on effectively using the predictive query governor.

| You can also test the performance of your queries using the predictive query governor. See "Testing performance with the query governor" on page 96.

| And finally, see "Cancelling a query" on page 95 to see how to cancel a query that is predicted to run beyond its time limit.

How the query governor works

The governor works in conjunction with the query optimizer. When a user requests DB2 Universal Database for iSeries to run a query, the following occurs:

1. The query access plan is evaluated by the optimizer.
As part of the evaluation, the optimizer predicts or estimates the runtime for the query. This helps determine the best way to access and retrieve the data for the query.
2. The estimated runtime is compared against the user-defined query time limit currently in effect for the job or user session.
3. If the predicted runtime for the query is less than or equal to the query time limit, the query governor lets the query run without interruption and no message is sent to the user.
4. If the query time limit is exceeded, inquiry message CPA4259 is sent to the user. The message states that the estimated query processing time of XX seconds exceeds the time limit of YY seconds.

Note: A default reply can be established for this message so that the user does not have the option to reply to the message, and the query request is *always* ended.

5. If a default message reply is not used, the user chooses to do one of the following:
 - End the query request before it is actually run.
 - Continue and run the query even though the predicted runtime exceeds the governor time limit.

Canceling a query

When a query is expected to run longer than the set time limit, the governor issues inquiry message CPA4259. You can respond to the message in one of the following ways:

- Enter a C to cancel the query. Escape message CPF427F is issued to the SQL runtime code. SQL returns SQLCODE -666.
- Enter an I to ignore the time limit and let the query run to completion.

Query governor implementation considerations

It is important to remember that the time limit generated by the optimizer is *only* an estimate. The actual query runtime could be more or less than the estimate, but the value of the two should be about the same. When setting the time limit for the entire server, it is usually best to set the limit to the maximum allowable time that any query should be allowed to run. By setting the limit too low you will run the risk of preventing some queries from completing and thus preventing the application from successfully finishing. There are many functions that use the query component to internally perform query requests. These requests will also be compared to the user-defined time limit.

Query governor considerations for user applications: Setting the time limit

Setting the time limit for jobs other than the current job

You can set the time limit for a job other than the current job. You do this by using the JOB parameter on the CHGQRYA command to specify either a query options file library to search (QRYOPLIB) or a specific QRYTIMLMT for that job.

Using the time limit to balance system resources

After the source job runs the CHGQRYA command, effects of the governor on the target job is not dependent upon the source job. The query time limit remains in effect for the duration of the job or user session, or until the time limit is changed by a CHGQRYA command. Under program control, a user could be given different query time limits depending on the application function being performed, the time of day, or the amount of system resources available. This provides a significant amount of flexibility when trying to balance system resources with temporary query requirements.

Controlling the default reply to the query governor inquiry message

The system administrator can control whether the interactive user has the option of ignoring the database query inquiry message by using the CHGJOB CL command as follows:

- If a value of *DFT is specified for the INQMSGRPY parameter of the CHGJOB CL command, the interactive user does not see the inquiry messages and the query is canceled immediately.
- If a value of *RQD is specified for the INQMSGRPY parameter of the CHGJOB CL command, the interactive user sees the inquiry and must reply to the inquiry.
- If a value of *SYSRPLY is specified for the INQMSGRPY parameter of the CHGJOB CL command, a system reply list is used to determine whether the interactive user sees the inquiry and whether a reply is necessary. For more information on the *SYSRPLY parameter, see the CL command information in the **Programming** category of the iSeries Information Center. The system reply list entries can be used to customize different default replies based on user profile name, user id, or process names. The fully qualified job name is available in the message data for inquiry message CPA4259. This will allow the keyword CMPDTA to be used to select the system reply list entry that applies to the process or user profile. The user profile name is 10 characters long and starts at position 51. The process name is 10 character long and starts at position 27.
- The following example will add a reply list element that will cause the default reply of C to cancel any requests for jobs whose user profile is 'QPGMR'.

```
ADDRPYLE SEQNBR(56) MSGID(CPA4259) CMPDTA(QPGMR) 51) RPY(C)
```

The following example will add a reply list element that will cause the default reply of C to cancel any requests for jobs whose process name is 'QPADEV0011'.

```
ADDRPYLE SEQNBR(57) MSGID(CPA4259) CMPDTA(QPADEV0011 27) RPY(C)
```

Testing performance with the query governor

You can use the query governor to test the performance of your queries:

1. Set the query time limit to zero (QRYTIMLMT(0)) using the CHGQRYA command or in the INI file. This forces an inquiry message from the governor stating that the estimated time to run the query exceeds the query time limit.
2. Prompt for message help on the inquiry message and find the same information that you would find by running the PRTSQLINF (Print SQL Information) command.

The query governor lets you optimize performance without having to run through several iterations of the query.

Additionally, if the query is canceled, the query optimizer evaluates the access plan and sends the optimizer debug messages to the job log. This occurs even if the job is *not* in debug mode. You can then review the optimizer tuning messages in the job log to see if additional tuning is needed to obtain optimal query performance. This allows you to try several permutations of the query with different attributes, indexes, and/or syntax to determine what performs better through the optimizer without actually running the query to completion. This saves on system resources because the actual query of the data is never actually done. If the tables to be queried contain a large number of rows, this represents a significant savings in system resources.

Be careful when you use this technique for performance testing, because all query requests will be stopped before they are run. This is especially important for a query that cannot be implemented in a single query step. For these types of queries, separate multiple query requests are issued, and then their results are accumulated before returning the final results. Stopping the query in one of these intermediate steps gives you only the performance information that relates to that intermediate step, and not for the entire query.

Examples of setting query time limits

To set the query time limit for the current job or user session using query options file QAQQINI, specify QRYOPTLIB parameter on the CHGQRYA command to a user library where the QAQQINI file exists with the parameter set to QUERY_TIME_LIMIT, and the value set to a valid query time limit. For more information on setting the query options file, see "Control queries dynamically with the query options file QAQQINI" on page 84.

To set the query time limit for 45 seconds you would use the following CHGQRYA command:

```
CHGQRYA JOB(*) QRYTIMLMT(45)
```

This sets the query time limit at 45 seconds. If the user runs a query with an estimated runtime equal to or less than 45 seconds, the query runs without interruption. The time limit remains in effect for the duration of the job or user session, or until the time limit is changed by the CHGQRYA command.

Assume that the query optimizer estimated the runtime for a query as 135 seconds. A message would be sent to the user that stated that the estimated runtime of 135 seconds exceeds the query time limit of 45 seconds.

To set or change the query time limit for a job other than your current job, the CHGQRYA command is run using the JOB parameter. To set the query time limit to 45 seconds for job 123456/USERNAME/JOBNAME you would use the following CHGQRYA command:

```
CHGQRYA JOB(123456/USERNAME/JOBNAME) QRYTIMLMT(45)
```

This sets the query time limit at 45 seconds for job 123456/USERNAME/JOBNAME. If job 123456/USERNAME/JOBNAME tries to run a query with an estimated runtime equal to or less than 45 seconds the query runs without interruption. If the estimated runtime for the query is greater than 45 seconds, for example 50 seconds, a message would be sent to the user stating that the estimated runtime of 50 seconds exceeds the query time limit of 45 seconds. The time limit remains in effect for the duration of job 123456/USERNAME/JOBNAME, or until the time limit for job 123456/USERNAME/JOBNAME is changed by the CHGQRYA command.

To set or change the query time limit to the QQRYTIMLMT system value, use the following CHGQRYA command:

```
CHGQRYA QRYTIMLMT(*SYSVAL)
```

The QQRYTIMLMT system value is used for duration of the job or user session, or until the time limit is changed by the CHGQRYA command. This is the default behavior for the CHGQRYA command.

Note: The query time limit can also be set in the INI file, or by using the SYSVAL command.

Control parallel processing for queries

You can turn parallel processing on and off. If the DB2 UDB Symmetric Multiprocessing feature is installed, then you can also turn symmetric multiprocessing (SMP) on and off.

- For system wide control, use the system value QQRYDEGREE.
- For job level control, use the DEGREE parameter on the CHGQRYA command, or the PARALLEL_DEGREE option of the query options file QAQQINI.

Even though parallelism has been enabled for a server or given job, the individual queries that run in a job might not actually use a parallel method. This might be because of functional restrictions, or the optimizer might choose a non-parallel method because it runs faster. See the previous sections that describe the performance characteristics and restrictions of each of the parallel access methods. The parallel methods that are available are:

- Parallel table scan method
- Parallel index scan-key selection method
- Parallel index scan-key positioning method
- Parallel index only access method (also for non-parallel)
- Parallel hashing method (also for non-parallel)
- Parallel bitmap processing method

Because queries being processed with parallel access methods aggressively use main storage, CPU, and disk resources, the number of queries that use parallel processing should be limited and controlled.

Controlling system wide parallel processing for queries

You can use the QQRYDEGREE system value to control parallel processing for a server. The current value of the system value can be displayed or modified using the following CL commands:

- WRKSYSVAL - Work with System Value
- CHGSYSVAL - Change System Value
- DSPSYSVAL - Display System Value
- RTVSYSVAL - Retrieve System Value

The special values for QQRYDEGREE control whether parallel processing is allowed by default for all jobs on the server. The possible values are:

***NONE**

No parallel processing is allowed for database query processing.

***IO**

I/O parallel processing is allowed for queries.

***OPTIMIZE**

The query optimizer can choose to use any number of tasks for either I/O or SMP parallel processing to process the queries. SMP parallel processing is used only if the DB2 UDB Symmetric Multiprocessing feature is installed. The query optimizer chooses to use parallel processing to minimize elapsed time based on the job's share of the memory in the pool.

***MAX**

The query optimizer can choose to use either I/O or SMP parallel processing to process the query. SMP parallel processing can be used only if the DB2 UDB Symmetric Multiprocessing feature is installed. The choices made by the query optimizer are similar to those made for parameter value *OPTIMIZE, except the optimizer assumes that all active memory in the pool can be used to process the query.

The default value of the QQRVDEGREE system value is *NONE, so the value must be changed if parallel query processing is desired as the default for jobs run on the server.

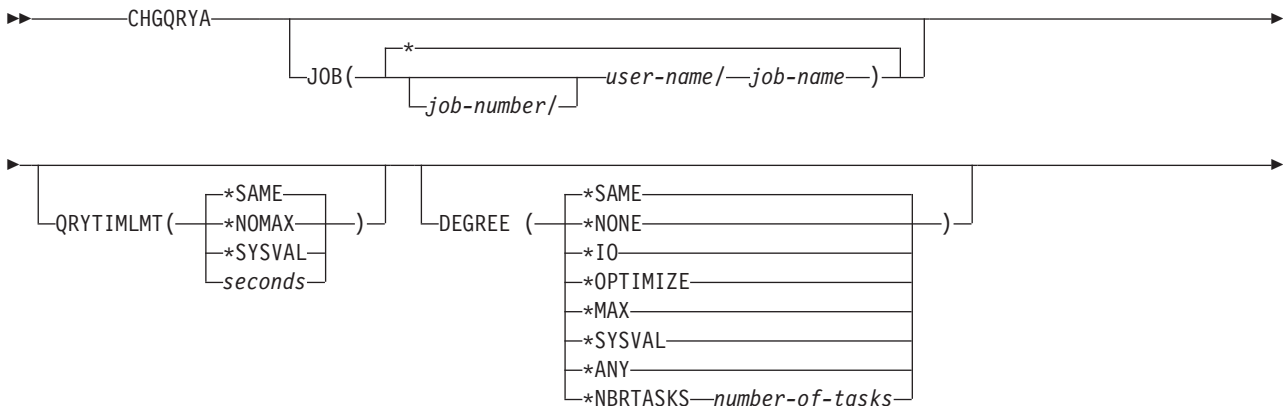
Changing this system value affects all jobs that will be run or are currently running on the server whose DEGREE query attribute is *SYSVAL. However, queries that have already been started or queries using reusable ODPs are not affected.

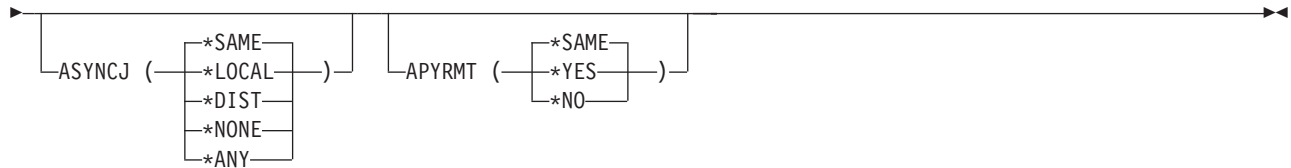
Controlling job level parallel processing for queries

You can also control query parallel processing at the job level using the DEGREE parameter of the Change Query Attributes (CHGQRYA) command or in the INI file. The parallel processing option allowed and, optionally, the number of tasks that can be used when running database queries in the job can be specified. You can prompt on the CHGQRYA command in an interactive job to display the current values of the DEGREE query attribute.

Changing the DEGREE query attribute does not affect queries that have already been started or queries using reusable ODPs.

Job: B,I Pgm: B,I REXX: B,I Exec
(1) (2)





Notes:

- 1 Value *ANY is equivalent to value *IO.
- 2 All parameters preceding this point can be specified in positional form.

The parameter values for the DEGREE keyword are:

***SAME**

The parallel degree query attribute does not change.

***NONE**

No parallel processing is allowed for database query processing.

***IO**

Any number of tasks can be used when the database query optimizer chooses to use I/O parallel processing for queries. SMP parallel processing is not allowed.

***OPTIMIZE**

The query optimizer can choose to use any number of tasks for either I/O or SMP parallel processing to process the query. SMP parallel processing can be used only if the DB2 UDB Symmetric Multiprocessing feature is installed. Use of parallel processing and the number of tasks used is determined with respect to the number of processors available in the server, the job's share of the amount of active memory available in the pool in which the job is run, and whether the expected elapsed time for the query is limited by CPU processing or I/O resources. The query optimizer chooses an implementation that minimizes elapsed time based on the job's share of the memory in the pool.

***MAX**

The query optimizer can choose to use either I/O or SMP parallel processing to process the query. SMP parallel processing can be used only if the DB2 UDB Symmetric Multiprocessing feature is installed. The choices made by the query optimizer are similar to those made for parameter value *OPTIMIZE except the optimizer assumes that all active memory in the pool can be used to process the query.

***NBRTASKS** *number-of-tasks*

Specifies the number of tasks to be used when the query optimizer chooses to use SMP parallel processing to process a query. I/O parallelism is also allowed. SMP parallel processing can be used only if the DB2 UDB Symmetric Multiprocessing feature is installed.

Using a number of tasks less than the number of processors available on the server restricts the number of processors used simultaneously for running a given query. A larger number of tasks ensures that the query is allowed to use all of the processors available on the server to run the query. Too many tasks can degrade performance because of the over commitment of active memory and the overhead cost of managing all of the tasks.

***SYSVAL**

Specifies that the processing option used should be set to the current value of the QQRVDEGREE system value.

The initial value of the DEGREE attribute for a job is *SYSVAL.

Analyzing queries with the Statistics Manager

Statistical information and other factors can be used by the query optimizer to determine the best access plan for a query. To be of value, this statistical information must be accurate and complete. Since the query optimizer bases its choice of access plan on the statistical information found in the table, it is important that this information be current. On many platforms, statistics collection is a manual process that is the responsibility of the database administrator. With iSeries servers, the database statistics collection process is handled automatically, and it is rarely necessary for the statistics to be manually updated, even though it is possible to manage statistics manually. In this release, the database statistics function of iSeries Navigator gives you the ability to manage statistical information for a table.

Note: If you decide to collect statistics manually, and you set the statistics in iSeries Navigator to be maintained manually, not allowing the system to perform automatic updates, or if you want to speed up the automatic update process, then statistics should be updated when:

- a table is loaded or reorganized
- a significant number of rows have been inserted, updated, or deleted
- a new column has been added to the table
- the Statistics Advisor in Visual Explain recommends that statistics should be created or updated

Statistics Manager APIs

The following APIs are used to implement the statistics function of iSeries Navigator.

- Cancel Requested Statistics Collections(QDBSTCRS, QdbstCancelRequestedStatistics) immediately cancels statistics collections that have been requested, but are not yet completed or not successfully completed.
- Delete Statistics Collections (QDBSTDS, QdbstDeleteStatistics) immediately deletes existing completed statistics collections.
- List Requested Statistics Collections(QDBSTLRS, QdbstListRequestedStatistics) lists all of the columns and combination of columns and file members that have background statistic collections requested, but not yet completed.
- List Statistics Collection Details(QDBSTLDS,) lists additional statistics data for a single statistics collection.
- List Statistics Collections(QDBSTLS, QdbstListStatistics) lists all of the columns and combination of columns for a given file member that have statistics available.
- Request Statistics Collections(QDBSTRS, QdbstRequestStatistics) allows you to request one or more statistics collections for a given set of columns of a specific file member.
- Update Statistics Collection(QDBSTUS, QdbstUpdateStatistics) allows you to update the attributes and to refresh the data of an existing single statistics collection

Managing statistical information with iSeries Navigator

With the iSeries Navigator statistics function, you can manage statistical information for a table. Once you have the statistical information that you need, you can use it to evaluate why queries against a particular table are performing poorly. Select one of the following topics to help you use iSeries Navigator to manage statistics.

- “Creating a statistics with iSeries Navigator”
- “Viewing statistics data for a table or alias with iSeries Navigator” on page 101
- “Updating statistics with iSeries Navigator” on page 101

Creating a statistics with iSeries Navigator

To create new statistics with iSeries Navigator, follow these steps.

1. Open iSeries Navigator.
2. In the iSeries Navigator window, expand the server you want to use.

3. Expand **Databases**.
4. Expand the database that contains the library in which the table or alias is stored.
5. Right-click on the table or alias and select **Statistics Data**.
6. On the **Statistics Data** dialog, click **New**.
7. On the **New Statistics** dialog, in the Columns available list, select the column or columns for which you want to collect statistics. Click **Add**.

Viewing statistics data for a table or alias with iSeries Navigator

To view statistics for a table or alias with iSeries Navigator, follow these steps.

1. Open iSeries Navigator.
 2. In the iSeries Navigator window, expand the server you want to use.
 3. Expand **Databases**.
 4. Expand the database that contains the library in which the table or alias is stored.
 5. Right-click on the table or alias and select **Statistics Data**.
- From this dialog, you can view statistic data details and update the statistics.

Updating statistics with iSeries Navigator

To update statistics for a table or alias with iSeries Navigator, follow these steps.

1. Open iSeries Navigator.
2. In the iSeries Navigator window, expand the server you want to use.
3. Expand **Databases**.
4. Expand the database that contains the library in which the table or alias is stored.
5. Right-click on the table or alias and select **Statistics Data**.
6. On the **Statistics Data** dialog, click **Update**.

Query optimization tools: Comparison table

PRTSQLINF	STRDBG or CHGQRYA	File-based monitor	Memory -Based Monitor	Visual Explain
Available without running query (after access plan has been created)	Only available when the query is run	Only available when the query is run	Only available when the query is run	Only available when the query is explained
Displayed for all queries in SQL program, whether executed or not	Displayed only for those queries which are executed	Displayed only for those queries which are executed	Displayed only for those queries which are executed	Displayed only for those queries that are explained
Information on host variable implementation	Limited information on the implementation of host variables	All information on host variables, implementation, and values	All information on host variables, implementation, and values	All information on host variables, implementation, and values
Available only to SQL users with programs, packages, or service programs	Available to all query users (OPNQRYF, SQL, QUERY/400)	Available to all query users (OPNQRYF, SQL, QUERY/400)	Available only to SQL interfaces	Available through iSeries Navigator Database and API interface
Messages are printed to spool file	Messages is displayed in job log	Performance rows are written to database table	Performance information is collected in memory and then written to database table	Information is displayed visually through iSeries Navigator

PRTSQLINF	STRDBG or CHGQRYA	File-based monitor	Memory -Based Monitor	Visual Explain
Easier to tie messages to query with subqueries or unions	Difficult to tie messages to query with subqueries or unions	Uniquely identifies every query, subquery and materialized view	Repeated query requests are summarized	Easy to view implementation of the query and associated information

Chapter 5. Using indexes to speed access to large tables

DB2 Universal Database for iSeries provides two basic means for accessing tables: a table scan (sequential) and an index-based (direct) retrieval. Index-based retrieval is usually more efficient than table scan. However, when a very large percentage of pages are retrieved, table scan is more efficient than index-based retrieval.

If DB2 Universal Database for iSeries cannot use an index to access the data in a table, it will have to read all the data in the table. Very large tables present a special performance problem: the high cost of retrieving all the data in the table. The following topics provide suggestions that will help you to design code which allows DB2 Universal Database for iSeries to take advantage of available indexes:

- “Coding for effective indexes: Avoid numeric conversions”
- “Coding for effective indexes: Avoid arithmetic expressions” on page 104
- “Coding for effective indexes: Avoid character string padding” on page 104
- “Coding for effective indexes: Avoid the use of like patterns beginning with % or _” on page 104
- “Coding for effective indexes: Be aware of the instances where DB2 UDB for iSeries does not use an index” on page 105

Additional information about using indexes:

See “Coding for effective indexes: Using indexes with sort sequence” on page 106 for information about how indexes work with sort sequence tables.

See “Examples of indexes” on page 107 for coding examples of effective indexes.

For information on the various ways to create an index, see the create an index topic in the iSeries Information Center.

Coding for effective indexes: Avoid numeric conversions

When a column value and a host variable (or constant value) are being compared, try to specify the same data types and attributes. DB2 Universal Database for iSeries does not use an index for the named column if the host variable or constant value has a greater precision than the precision of the column. If the two items being compared have different data types, DB2 Universal Database for iSeries will have to convert one or the other of the values, which can result in inaccuracies (because of limited machine precision). To avoid problems for columns and constants being compared, use the following:

- same data type
- same scale, if applicable
- same precision, if applicable

For example, EDUCLVL is a halfword integer value (SMALLINT). When using SQL, specify:

```
... WHERE EDUCLVL < 11 AND  
        EDUCLVL >= 2
```

instead of

```
... WHERE EDUCLVL < 1.1E1 AND  
        EDUCLVL > 1.3
```

When using the OPNQRYF command, specify:

```
... QRYSLT('EDUCLVL *LT 11 *AND ENUCLVL *GE 2')
```

instead of

```
... QRYSLT('EDUCLVL *LT 1.1E1 *AND EDUCLVL *GT 1.3')
```

If an index was created over the EDUCLVL column, then the optimizer does not use the index in the second example because the precision of the constant is greater than the precision of the column. In the first example, the optimizer considers using the index, because the precisions are equal.

Coding for effective indexes: Avoid arithmetic expressions

Do not use an arithmetic expression as an operand to be compared to a column in a row selection predicate. The optimizer does not use an index on a column that is being compared to an arithmetic expression. While this may not cause an index over the column to become unusable, it will prevent any estimates and possibly the use of index scan-key positioning on the index. The primary thing that is lost is the ability to use and extract any statistics that might be useful in the optimization of the query. For example, when using SQL, specify the following:

```
... WHERE SALARY > 16500
```

instead of

```
... WHERE SALARY > 15000*1.1
```

Coding for effective indexes: Avoid character string padding

Try to use the same data length when comparing a fixed-length character string column value to a host variable or constant value. DB2 Universal Database for iSeries does not use an index if the constant value or host variable is longer than the column length. For example, EMPNO is CHAR(6) and DEPTNO is CHAR(3). For example, when using SQL, specify the following:

```
... WHERE EMPNO > '000300' AND
      DEPTNO < 'E20'
```

instead of

```
... WHERE EMPNO > '000300 ' AND
      DEPTNO < 'E20 '
```

When using the OPNQRYF command, specify:

```
... QRYSLT('EMPNO *GT "000300" *AND DEPTNO *LT "E20"')
```

instead of

```
... QRYSLT('EMPNO *GT "000300" *AND DEPTNO *LT "E20"')
```

Coding for effective indexes: Avoid the use of like patterns beginning with % or _

The percent sign (%), and the underline (_), when used in the pattern of a LIKE (OPNQRYF %WLDCRD) predicate, specify a character string that is similar to the column value of rows you want to select. They can take advantage of indexes when used to denote characters in the middle or at the end of a character string, as in the following. For example, when using SQL, specify the following:

```
... WHERE LASTNAME LIKE 'J%SON%'
```

When using the OPNQRYF command, specify the following:

```
... QRYSLT('LASTNAME *EQ %WLDCRD(''J*SON*''')
```

However, when used at the beginning of a character string, they can prevent DB2 Universal Database for iSeries from using any indexes that might be defined on the LASTNAME column to limit the number of rows scanned using index scan-key positioning. Index scan-key selection, however, is allowed. For example, in the following queries index scan-key selection could be used, but index scan-key positioning could not be.

In SQL:

```
... WHERE LASTNAME LIKE '%SON'
```

In OPNQRYF:

```
... QRYSLT('LASTNAME *EQ %WLDCRD(' '*SON''')
```

Ideally, you should avoid patterns with a % so that you can get the best performance when you perform key processing on the predicate. If you can exercise control over the queries or application, you should try to get a partial string to search so that index scan-key positioning can be used.

For example, if you were looking for the name "Smithers", but you only type "S%," this query will return all names starting with "S." You would probably then adjust the query to return all names with "Smi%", so by forcing the use of partial strings, better performance would be realized in the long term.

Coding for effective indexes: Be aware of the instances where DB2 UDB for iSeries does not use an index

DB2 Universal Database for iSeries does not use indexes in the following instances:

- For a column that is expected to be updated; for example, when using SQL, your program might include the following:

```
EXEC SQL
  DECLARE DEPTEMP CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  WHERE (WORKDEPT = 'D11' OR
        WORKDEPT = 'D21') AND
        EMPNO = '000190'
  FOR UPDATE OF EMPNO, WORKDEPT
END-EXEC.
```

When using the OPNQRYF command, for example:

```
OPNQRYF FILE((CORPDATA/EMPLOYEE)) OPTION(*ALL)
  QRYSLT('(WORKDEPT *EQ ''D11'' *OR WORKDEPT *EQ ''D21''
  *AND EMPNO *EQ ''000190''')
```

Even if you do not intend to update the employee's department, DB2 Universal Database for iSeries cannot use an index with a key of WORKDEPT.

DB2 Universal Database for iSeries can use an index if all of the updateable columns used within the index are also used within the query as an isolatable selection predicate with an equal operator. In the previous example, DB2 Universal Database for iSeries would use an index with a key of EMPNO.

DB2 Universal Database for iSeries can operate more efficiently if the FOR UPDATE OF column list only names the column you intend to update: *WORKDEPT*. Therefore, do not specify a column in the FOR UPDATE OF column list unless you intend to update the column.

If you have an updateable cursor because of dynamic SQL or the FOR UPDATE clause was not specified and the program contains an UPDATE statement then all columns can be updated.

- For a column being compared with another column from the same row. For example, when using SQL, your program might include the following:

```
EXEC SQL
  DECLARE DEPTDATA CURSOR FOR
  SELECT WORKDEPT, DEPTNAME
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = ADMRDEPT
END-EXEC.
```

When using the OPNQRYP command, for example:

```
OPNQRYP FILE (EMPLOYEE) FORMAT(FORMAT1)
        QRYSLT('WORKDEPT *EQ ADMRDEPT')
```

Even though there is an index for *WORKDEPT* and another index for *ADMRDEPT*, DB2 Universal Database for iSeries will not use either index. The index has no added benefit because every row of the table needs to be looked at.

Coding for effective indexes: Using indexes with sort sequence

The following sections provide useful information about how indexes work with sort sequence tables.

- “Coding for effective indexes: Using indexes and sort sequence with selection, joins, or grouping”
- “Coding for effective indexes: Ordering”

For more information on how sort sequence tables work, see the topic “Sort Sequence” in the SQL Reference book.

Coding for effective indexes: Using indexes and sort sequence with selection, joins, or grouping

Before using an existing index, DB2 Universal Database for iSeries ensures the attributes of the columns (selection, join, or grouping columns) match the attributes of the key columns in the existing index. The sort sequence table is an additional attribute that must be compared.

The sort sequence table associated with the query (specified by the SRTSEQ and LANGID parameters) must match the sort sequence table with which the existing index was built. DB2 Universal Database for iSeries compares the sort sequence tables. If they do not match, the existing index cannot be used.

There is an exception to this, however. If the sort sequence table associated with the query is a unique-weight sequence table (including *HEX), DB2 Universal Database for iSeries acts as though no sort sequence table is specified for selection, join, or grouping columns that use the following operators and predicates:

- equal (=) operator
- not equal (^= or <>) operator
- LIKE predicate (OPNQRYP %WLDICRD and *CT)
- IN predicate (OPNQRYP %VALUES)

When these conditions are true, DB2 Universal Database for iSeries is free to use any existing index where the key columns match the columns and either:

- The index does not contain a sort sequence table or
- The index contains a unique-weight sort sequence table

Note: The table does not have to match the unique-weight sort sequence table associated with the query.

Note: Bitmap processing has a special consideration when multiple indexes are used for a table. If two or more indexes have a common key column between them that is also referenced in the query selection, then those indexes must either use the same sort sequence table or use no sort sequence table.

Coding for effective indexes: Ordering

Unless the optimizer chooses to do a sort to satisfy the ordering request, the sort sequence table associated with the index must match the sort sequence table associated with the query.

When a sort is used, the translation is done during the sort. Since the sort is handling the sort sequence requirement, this allows DB2 Universal Database for iSeries to use any existing index that meets the selection criteria.

Examples of indexes

| The following index examples are provided to help you create effective indexes.

For the purposes of the examples, assume that three indexes are created.

Assume that an index HEXIX was created with *HEX as the sort sequence.

```
CREATE INDEX HEXIX ON STAFF (JOB)
```

Assume that an index UNQIX was created with a unique-weight sort sequence.

```
CREATE INDEX UNQIX ON STAFF (JOB)
```

Assume that an index SHRIX was created with a shared-weight sort sequence.

```
CREATE INDEX SHRIX ON STAFF (JOB)
```

- | • Equals selection with no sort sequence table
- | • Equals selection with a unique-weight sort sequence table
- | • Equals selection with a shared-weight sort sequence table
- | • Greater than selection with a unique-weight sort sequence table
- | • Join selection with a unique-weight sort sequence table
- | • Join selection with a shared-weight sort sequence table
- | • Ordering with no sort sequence table
- | • Ordering with a unique-weight sort sequence table
- | • Ordering with a shared-weight sort sequence table
- | • Ordering with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table
- | • Grouping with no sort sequence table
- | • Grouping with a unique-weight sort sequence table
- | • Grouping with a shared-weight sort sequence table
- | • Ordering and grouping on the same columns with a unique-weight sort sequence table
- | • Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table
- | • Ordering and grouping on the same columns with a shared-weight sort sequence table
- | • Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table
- | • Ordering and grouping on different columns with a unique-weight sort sequence table
- | • Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table
- | • Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table

Index example: Equals selection with no sort sequence table

Equals selection with no sort sequence table (SRTSEQ(*HEX)).

```
SELECT * FROM STAFF  
WHERE JOB = 'MGR'
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
  QRYSLT('JOB *EQ 'MGR''')
  SRTSEQ(*HEX)
```

DB2 Universal Database for iSeries could use either index HEXIX or index UNQIX.

Index example: Equals selection with a unique-weight sort sequence table

Equals selection with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF
WHERE JOB = 'MGR'
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
  QRYSLT('JOB *EQ 'MGR''')
  SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

DB2 Universal Database for iSeries could use either index HEXIX or index UNQIX.

Index example: Equal selection with a shared-weight sort sequence table

Equal selection with a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT * FROM STAFF
WHERE JOB = 'MGR'
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
  QRYSLT('JOB *EQ 'MGR''')
  SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

DB2 Universal Database for iSeries could only use index SHRIX.

Index example: Greater than selection with a unique-weight sort sequence table

Greater than selection with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF
WHERE JOB > 'MGR'
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
  QRYSLT('JOB *GT 'MGR''')
  SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

DB2 Universal Database for iSeries could only use index UNQIX.

Index example: Join selection with a unique-weight sort sequence table

Join selection with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF S1, STAFF S2
WHERE S1.JOB = S2.JOB
```

or the same query using the JOIN syntax.

```

SELECT *
FROM STAFF S1 INNER JOIN STAFF S2
ON S1.JOB = S2.JOB

```

When using the OPNQRYF command, specify:

```

OPNQRYF FILE(STAFF STAFF)
FORMAT(FORMAT1)
JFLD((1/JOB 2/JOB *EQ))
SRTSEQ(*LANGIDUNQ) LANGID(ENU)

```

DB2 Universal Database for iSeries could use either index HEXIX or index UNQIX for either query.

Index example: Join selection with a shared-weight sort sequence table

Join selection with a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```

SELECT * FROM STAFF S1, STAFF S2
WHERE S1.JOB = S2.JOB

```

or the same query using the JOIN syntax.

```

SELECT *
FROM STAFF S1 INNER JOIN STAFF S2
ON S1.JOB = S2.JOB

```

When using the OPNQRYF command, specify:

```

OPNQRYF FILE(STAFF STAFF) FORMAT(FORMAT1)
JFLD((1/JOB 2/JOB *EQ))
SRTSEQ(*LANGIDSHR) LANGID(ENU)

```

DB2 Universal Database for iSeries could only use index SHRIX for either query.

Index example: Ordering with no sort sequence table

Ordering with no sort sequence table (SRTSEQ(*HEX)).

```

SELECT * FROM STAFF
WHERE JOB = 'MGR'
ORDER BY JOB

```

When using the OPNQRYF command, specify:

```

OPNQRYF FILE((STAFF))
QRYSLT('JOB *EQ ''MGR''')
KEYFLD(JOB)
SRTSEQ(*HEX)

```

DB2 Universal Database for iSeries could only use index HEXIX.

Index example: Ordering with a unique-weight sort sequence table

Ordering with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```

SELECT * FROM STAFF
WHERE JOB = 'MGR'
ORDER BY JOB

```

When using the OPNQRYF command, specify:

```

OPNQRYF FILE((STAFF))
QRYSLT('JOB *EQ ''MGR''')
KEYFLD(JOB) SRTSEQ(*LANGIDUNQ) LANGID(ENU)

```

DB2 Universal Database for iSeries could only use index UNQIX.

Index example: Ordering with a shared-weight sort sequence table

Ordering with a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT * FROM STAFF
WHERE JOB = 'MGR'
ORDER BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
  QRYSLT('JOB *EQ ''MGR''')
  KEYFLD(JOB) SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

DB2 Universal Database for iSeries could only use index SHRIX.

Index example: Ordering with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table

Ordering with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF
WHERE JOB = 'MGR'
ORDER BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
  QRYSLT('JOB *EQ ''MGR''')
  KEYFLD(JOB)
  SRTSEQ(*LANGIDUNQ) LANGID(ENU)
  ALWCPYDTA(*OPTIMIZE)
```

DB2 Universal Database for iSeries could use either index HEXIX or index UNQIX for selection. Ordering would be done during the sort using the *LANGIDUNQ sort sequence table.

Index example: Grouping with no sort sequence table

Grouping with no sort sequence table (SRTSEQ(*HEX)).

```
SELECT JOB FROM STAFF
GROUP BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT2)
  GRPFLD((JOB))
  SRTSEQ(*HEX)
```

DB2 Universal Database for iSeries could use either index HEXIX or index UNQIX.

Index example: Grouping with a unique-weight sort sequence table

Grouping with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT JOB FROM STAFF
GROUP BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT2)
  GRPFLD((JOB))
  SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

DB2 Universal Database for iSeries could use either index HEXIX or index UNQIX.

Index example: Grouping with a shared-weight sort sequence table

Grouping with a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT JOB FROM STAFF
GROUP BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT2)
  GRPFLD((JOB))
  SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

DB2 Universal Database for iSeries could only use index SHRIX.

The following examples assume that 3 more indexes are created over columns JOB and SALARY. The CREATE INDEX statements precede the examples.

Assume an index HEXIX2 was created with *HEX as the sort sequence.

```
CREATE INDEX HEXIX2 ON STAFF (JOB, SALARY)
```

Assume that an index UNQIX2 was created and the sort sequence is a unique-weight sort sequence.

```
CREATE INDEX UNQIX2 ON STAFF (JOB, SALARY)
```

Assume an index SHRIX2 was created with a shared-weight sort sequence.

```
CREATE INDEX SHRIX2 ON STAFF (JOB, SALARY)
```

Index example: Ordering and grouping on the same columns with a unique-weight sort sequence table

Ordering and grouping on the same columns with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT JOB, SALARY FROM STAFF
GROUP BY JOB, SALARY
ORDER BY JOB, SALARY
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(JOB SALARY)
  SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

DB2 Universal Database for iSeries could use UNQIX2 to satisfy both the grouping and ordering requirements. If index UNQIX2 did not exist, DB2 Universal Database for iSeries would create an index using a sort sequence table of *LANGIDUNQ.

Index example: Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table

Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT JOB, SALARY FROM STAFF
GROUP BY JOB, SALARY
ORDER BY JOB, SALARY
```

When using the OPNQRYF command, specify:

```
OPNQRYP FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(JOB SALARY)
  SRTSEQ(*LANGIDUNQ) LANGID(ENU)
  ALWCPYDTA(*OPTIMIZE)
```

DB2 Universal Database for iSeries could use UNQIX2 to satisfy both the grouping and ordering requirements. If index UNQIX2 did not exist, DB2 Universal Database for iSeries would either:

- Create an index using a sort sequence table of *LANGIDUNQ or
- Use index HEXIX2 to satisfy the grouping and to perform a sort to satisfy the ordering

Index example: Ordering and grouping on the same columns with a shared-weight sort sequence table

Ordering and grouping on the same columns with a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT JOB, SALARY FROM STAFF
  GROUP BY JOB, SALARY
  ORDER BY JOB, SALARY
```

When using the OPNQRYP command, specify:

```
OPNQRYP FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(JOB SALARY)
  SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

DB2 Universal Database for iSeries could use SHRIX2 to satisfy both the grouping and ordering requirements. If index SHRIX2 did not exist, DB2 Universal Database for iSeries would create an index using a sort sequence table of *LANGIDSHR.

Index example: Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table

Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT JOB, SALARY FROM STAFF
  GROUP BY JOB, SALARY
  ORDER BY JOB, SALARY
```

When using the OPNQRYP command, specify:

```
OPNQRYP FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(JOB SALARY)
  SRTSEQ(*LANGIDSHR) LANGID(ENU)
  ALWCPYDTA(*OPTIMIZE)
```

DB2 Universal Database for iSeries could use SHRIX2 to satisfy both the grouping and ordering requirements. If index SHRIX2 did not exist, DB2 Universal Database for iSeries would create an index using a sort sequence table of *LANGIDSHR.

Index example: Ordering and grouping on different columns with a unique-weight sort sequence table

Ordering and grouping on different columns with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT JOB, SALARY FROM STAFF
  GROUP BY JOB, SALARY
  ORDER BY SALARY, JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(SALARY JOB)
  SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

DB2 Universal Database for iSeries could use index HEXIX2 or index UNQIX2 to satisfy the grouping requirements. A temporary result would be created containing the grouping results. A temporary index would then be built over the temporary result using a *LANGIDUNQ sort sequence table to satisfy the ordering requirements.

Index example: Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table

Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT JOB, SALARY FROM STAFF
  GROUP BY JOB, SALARY
  ORDER BY SALARY, JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(SALARY JOB)
  SRTSEQ(*LANGIDUNQ) LANGID(ENU)
  ALWCPYDTA(*OPTIMIZE)
```

DB2 Universal Database for iSeries could use index HEXIX2 or index UNQIX2 to satisfy the grouping requirements. A sort would be performed to satisfy the ordering requirements.

Index example: Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table

Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT JOB, SALARY FROM STAFF
  GROUP BY JOB, SALARY
  ORDER BY SALARY, JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(SALARY JOB)
  SRTSEQ(*LANGIDSHR) LANGID(ENU)
  ALWCPYDTA(*OPTIMIZE)
```

DB2 Universal Database for iSeries could use index SHRIX2 to satisfy the grouping requirements. A sort would be performed to satisfy the ordering requirements.

What are encoded vector indexes?

An encoded vector index (EVI) is an index object that is used by the query optimizer and database engine to provide fast data access in decision support and query reporting environments. EVIs are a complementary alternative to existing index objects (binary radix tree structure - logical file or SQL index) and are a variation on bitmap indexing. Because of their compact size and relative simplicity, EVIs provide for faster scans of a table that can also be processed in parallel.

An EVI is a data structure that is stored as two components:

- The symbol table contains statistical and descriptive information about each distinct key value represented in the table. Each distinct key is assigned a unique code, either 1, 2 or 4 bytes in size.
- The vector is an array of codes listed in the same ordinal position as the rows in the table. The vector does not contain any pointers to the actual rows in the table.

Advantages of EVIs

- Require less storage
- May have better build times
- Provide more accurate statistics to the query optimizer

Disadvantages of EVIs

- Cannot be used in ordering and grouping
- Have limited use in joins
- Some additional maintenance idiosyncrasies

How the EVI works

The optimizer uses the symbol table to collect the costing information about the query. If the optimizer decides to use an EVI to process the query, the database engine uses the vector to build the dynamic bitmap that contains one bit for each row in the table. If the row satisfies the query selection, the bit is set on. If the row does not satisfy the query selection, the bit is set off. Like a bitmap index, intermediate dynamic bitmaps can be AND'ed and OR'ed together to satisfy an ad hoc query. For example, if a user wants to see sales data for a certain region during a certain time period, you can define an EVI over the region column and the Quarter column of the database. When the query runs, the database engine builds dynamic bitmaps using the two EVIs and then ANDs the bitmaps together to produce a bitmap that contains only the relevant rows for both selection criteria. This AND'ing capability drastically reduces the number of rows that the server must read and test. The dynamic bitmap(s) exists only as long as the query is executing. Once the query is completed, the dynamic bitmap(s) are eliminated.

When should EVIs be used?

Encoded vector indexes should be considered when you want to gather statistics, when full table scan is selected, selectivity of the query is 20%-70% and using skip sequential access with dynamic bitmaps will speed up the scan, or when a star schema join is expected to be used for star schema join queries.

Encoded vector indexes should be created with:

- Single key columns with a low number of distinct values expected
- Keys columns with a low volatility (they don't change often)
- Maximum number of distinct values expected using the WITH n DISTINCT VALUES clause
- Single key over foreign key columns for a star schema model

General index maintenance EVI maintenance When using EVIs, there are a unique challenges to index maintenance. The following table shows a progression of how EVIs are maintained and the conditions under which EVIs are most effective and to the conditions where EVIs are least effective based on the EVI maintenance .idiosyncrasies.

General index maintenance

Whenever indexes are created and used, there is a potential for a decrease in I/O velocity due to maintenance, therefore, it is essential that you consider the maintenance cost of creating and using additional indexes. For radix indexes with MAINT(*IMMED) and EVIs, maintenance occurs when inserting, updating or deleting rows.

To reduce the maintenance of your indexes consider:

- Minimizing the number of indexes over a given table
- Dropping indexes during batch inserts, updates, and deletes
- Creating indexes, one at a time, in parallel using SMP


- Creating multiple indexes simultaneously with multiple batch jobs using multiple CPUs
- Maintaining indexes in parallel using SMP

The goal of creating indexes for performance is to balance the maximum number of indexes for statistics and implementation while minimizing the number of indexes to maintain.

EVI maintenance

When using EVIs, there are a unique challenges to index maintenance. The following table shows a progression of how EVIs are maintained and the conditions under which EVIs are most effective and where EVIs are least effective based on the EVI maintenance idiosyncrasies.

Table 10. EVI Maintenance Considerations

	Condition	Characteristics
<div style="display: flex; flex-direction: column; align-items: center;"> <div style="margin-bottom: 20px;">Most Effective</div>  <div style="margin-top: 20px;">Least Effective</div> </div>	When inserting an existing distinct key value	<ul style="list-style-type: none"> • Minimum overhead • Symbol table key value looked up and statistics updated • Vector element added for new row, with existing byte code
	When inserting a <i>new</i> distinct key value - <u>in order</u> , within byte code range	<ul style="list-style-type: none"> • Minimum overhead • Symbol table key value added, byte code assigned, statistics assigned • Vector element added for new row, with new byte code
	When inserting a new distinct key value - <u>out of order</u> , within byte code range	<ul style="list-style-type: none"> • Minimum overhead if contained within overflow area threshold • Symbol table key value added to overflow area, byte code assigned, statistics assigned • Vector element added for new row, with new byte code • Considerable overhead if overflow area threshold reached • Access path validated - not available • EVI refreshed, overflow area keys incorporated, new byte codes assigned (symbol table and vector elements updated)
	When inserting a new distinct key value - <u>out of byte code range</u>	<ul style="list-style-type: none"> • Considerable overhead • Access plan invalidated - not available • EVI refreshed, next byte code size used, new byte codes assigned (symbol table and vector elements updated)

Recommendations for EVI use

Encoded vector indexes are a powerful tool for providing fast data access in decision support and query reporting environments, but to ensure the effective use of EVIs, you should implement EVIs with the following guidelines:

Create EVIs on:

- Read only tables or tables with a minimum of INSERT, UPDATE, DELETE activity.
- Key columns that are used in the WHERE clause - local selection predicates of SQL requests.
- Single key columns that have a relatively small set of distinct values.
- Multiple key columns that result in a relatively small set of distinct values.
- Key columns that have a static or relatively static set of distinct values.
- Non-unique key columns, with many duplicates.

Create EVIs with the maximum byte code size expected:

- Use the "WITH n DISTINCT VALUES" clause on the CREATE ENCODED VECTOR INDEX statement.
- If unsure, use a number greater than 65,535 to create a 4 byte code, thus avoiding the EVI maintenance overhead of switching byte code sizes.

When loading data:

- Drop EVIs, load data, create EVIs.
- EVI byte code size will be assigned automatically based on the number of actual distinct key values found in the table.
- Symbol table will contain all key values, in order, no keys in overflow area.

Consider SMP and parallel index creation and maintenance:

Symmetrical Multiprocessing (SMP) is a valuable tool for building and maintaining indexes in parallel. The results of using the optional SMP feature of OS/400 are faster index build times, and faster I/O velocities while maintaining indexes in parallel. Using an SMP degree value of either *OPTIMIZE or *MAX, additional multiple tasks and additional server resources are used to build or maintain the indexes. With a degree value of *MAX, expect linear scalability on index creation. For example, creating indexes on a 4 processor server can be 4 times as fast as a 1 processor server.

Checking values in the overflow area:

You can also use the Display Field Description (DSPFD) command (or V4R5M0 Operations Navigator - Database) to check how many values are in the overflow area. Once the DSPFD command is issued, check the overflow area parameter for details on the initial and actual number of distinct key values in the overflow area.

Using CHGLF to rebuild an index's access path:

Use the Change Logical File (CHGLF) command with the attribute Force Rebuild Access Path set to YES (FRCRBDAP(*YES)). This command accomplishes the same thing as dropping and recreating the index, but it does not require that you know about how the index was built. This command is especially effective for applications where the original index definitions are not available, or for refreshing the access path.

Chapter 6. Application design tips for database performance

This section contains the following design tips that you can apply when designing SQL applications to maximize your database performance:

- “Database application design tips: Use live data”
- “Database application design tips: Reduce the number of open operations” on page 118
- “Database application design tips: Retain cursor positions” on page 120

Database application design tips: Use live data

The term **live data** refers to the type of access that the database manager uses when it retrieves data without making a copy of the data. Using this type of access, the data, which is returned to the program, always reflects the current values of the data in the database. The programmer can control whether the database manager uses a copy of the data or retrieves the data directly. This is done by specifying the allow copy data (ALWCOPYDATA) parameter on the precompiler commands or on the Start SQL (STRSQL) command.

Specifying ALWCOPYDATA(*NO) instructs the database manager to always use live data. Live data access can be used as a performance advantage because the cursor does not have to be closed and opened again to refresh the data being retrieved. An example application demonstrating this advantage is one that produces a list on a display. If the display screen can only show 20 elements of the list at a time, then, after the initial 20 elements are displayed, the application programmer can request that the next 20 rows be displayed. A typical SQL application designed for an operating system other than the OS/400 operating system, might be structured as follows:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  ORDER BY EMPNO
END-EXEC.
```

```
EXEC SQL
  OPEN C1
END-EXEC.
```

```
*   PERFORM FETCH-C1-PARA 20 TIMES.

      MOVE EMPNO to LAST-EMPNO.
```

```
EXEC SQL
  CLOSE C1
END-EXEC.
```

```
*   Show the display and wait for the user to indicate that
*   the next 20 rows should be displayed.
```

```
EXEC SQL
  DECLARE C2 CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  WHERE EMPNO > :LAST-EMPNO
  ORDER BY EMPNO
END-EXEC.
```

```
EXEC SQL
  OPEN C2
END-EXEC.
```

```
*   PERFORM FETCH-C21-PARA 20 TIMES.
```

* Show the display with these 20 rows of data.

```
EXEC SQL
  CLOSE C2
END-EXEC.
```

In the above example, notice that an additional cursor had to be opened to continue the list and to get current data. This could result in creating an additional ODP that would increase the processing time on the iSeries server. In place of the above example, the programmer could design the application specifying ALWCOPYDATA(*NO) with the following SQL statements:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  ORDER BY EMPNO
END-EXEC.
```

```
EXEC SQL
  OPEN C1
END-EXEC.
```

* Display the screen with these 20 rows of data.

* PERFORM FETCH-C1-PARA 20 TIMES.

* Show the display and wait for the user to indicate that
* the next 20 rows should be displayed.

* PERFORM FETCH-C1-PARA 20 TIMES.

```
EXEC SQL
  CLOSE C1
END-EXEC.
```

In the above example, the query could perform better if the FOR 20 ROWS clause was used on the multiple-row FETCH statement. Then, the 20 rows would be retrieved in one operation.

Database application design tips: Reduce the number of open operations

The SQL data manipulation language statements must do database open operations in order to create an open data path (ODP) to the data. An open data path is the path through which all input/output operations for the table are performed. In a sense, it connects the SQL application to a table. The number of open operations in a program can significantly affect performance. A database open operation occurs on:

- An OPEN statement
- SELECT INTO statement
- An INSERT statement with a VALUES clause
- An UPDATE statement with a WHERE condition
- An UPDATE statement with a WHERE CURRENT OF cursor and SET™ clauses that refer to operators or functions
- SET statement that contains an expression
- VALUES INTO statement that contains an expression
- A DELETE statement with a WHERE condition

An INSERT statement with a select-statement requires two open operations. Certain forms of subqueries may also require one open per subselect.

To minimize the number of opens, DB2 Universal Database for iSeries leaves the open data path (ODP) open and reuses the ODP if the statement is run again, unless:

- The ODP used a host variable to build a subset temporary index. The OS/400 database support may choose to build a temporary index with entries for only the rows that match the row selection specified in the SQL statement. If a host variable was used in the row selection, the temporary index will not have the entries required for a different value contained in the host variable.
- Ordering was specified on a host variable value.
- An Override Database File (OVRDBF) or Delete Override (DLTOVR) CL command has been issued since the ODP was opened, which would affect the SQL statement execution.

Note: Only overrides that affect the name of the table being referred to will cause the ODP to be closed within a given program invocation.

- The join is a complex join that requires temporaries to contain the intermediate steps of the join.
- Some cases involve a complex sort, where a temporary file is required, may not be reusable.
- A change to the library list since the last open has occurred, which would change the table selected by an unqualified referral in system naming mode.
- The join was implemented using hash join.

| For embedded static SQL, DB2 Universal Database for iSeries only reuses ODPs opened by the same
| statement. An identical statement coded later in the program does not reuse an ODP from any other
| statement. If the identical statement must be run in the program many times, code it once in a subroutine
| and call the subroutine to run the statement.

The ODPs opened by DB2 Universal Database for iSeries are closed when any of the following occurs:

- A CLOSE, INSERT, UPDATE, DELETE, or SELECT INTO statement completes and the ODP required a temporary result or a subset temporary index.
- The Reclaim Resources (RCLRSC) command is issued. A RCLRSC is issued when the first COBOL program on the call stack ends or when a COBOL program issues the STOP RUN COBOL statement. RCLRSC will not close ODPs created for programs precompiled using CLOSQLCSR(*ENDJOB). For interaction of RCLRSC with non-default activation groups, see the following books:
 - *WebSphere Development Studio: ILE C/C++ Programmer's Guide*
 - *WebSphere Development Studio: ILE COBOL Programmer's Guide*
 - *WebSphere Development Studio: ILE RPG Programmer's Guide*
- When the last program that contains SQL statements on the call stack exits, except for ODPs created for programs precompiled using CLOSQLCSR(*ENDJOB) or modules precompiled using CLOSQLCSR(*ENDACTGRP).
- When a CONNECT (Type 1) statement changes the application server for an activation group, all ODPs created for the activation group are closed.
- When a DISCONNECT statement ends a connection to the application server, all ODPs for that application server are closed.
- When a released connection is ended by a successful COMMIT, all ODPs for that application server are closed.
- | • When the threshold for open cursors specified by the query options file (QAQQINI) parameter
| OPEN_CURSOR_THRESHOLD is reached.

You can control whether DB2 Universal Database for iSeries keeps the ODPs open in the following ways:

- Design the application so a program that issues an SQL statement is always on the call stack
- Use the CLOSQLCSR(*ENDJOB) or CLOSQLCSR(*ENDACTGRP) parameter
- By specifying the OPEN_CURSOR_THRESHOLD and OPEN_CURSOR_CLOSE_COUNT parameters of the query options file (QAQQINI)

DB2 Universal Database for iSeries does an open operation for the first execution of each UPDATE WHERE CURRENT OF when any expression in the SET clause contains an operator or function. The open can be avoided by coding the function or operation in the host language code.

For example, the following UPDATE causes DB2 Universal Database for iSeries to do an open operation:

```
EXEC SQL
  FETCH EMPT INTO :SALARY
END-EXEC.

EXEC SQL
  UPDATE CORPDATA.EMPLOYEE
  SET SALARY = :SALARY + 1000
  WHERE CURRENT OF EMPT
END-EXEC.
```

Instead, use the following coding technique to avoid opens:

```
EXEC SQL
  FETCH EMPT INTO :SALARY
END EXEC.

ADD 1000 TO SALARY.

EXEC SQL
  UPDATE CORPDATA.EMPLOYEE
  SET SALARY = :SALARY
  WHERE CURRENT OF EMPT
END-EXEC.
```

You can determine whether or not SQL statements result in full opens in several ways. The preferred methods are to use the Database Monitor or by looking at the messages issued while debug is active. You can also use the CL commands Trace Job (TRCJOB) or Display Journal (DSPJRN).

Database application design tips: Retain cursor positions

- | You can improve performance by retaining cursor positions. Cursor positions can be retained for Non-ILE program calls and for ILE program calls. Also, there are some general rules for retaining cursor positions for all program calls.

Database application design tips: Retaining cursor positions for non-ILE program calls

For non-ILE program calls, the close SQL cursor (CLOSQLCSR) parameter allows you to specify the scope of the following:

- The cursors
- The prepared statements
- The locks

When used properly, the CLOSQLCSR parameter can reduce the number of SQL OPEN, PREPARE, and LOCK statements needed. It can also simplify applications by allowing you to retain cursor positions across program calls.

***ENDPGM**

This is the default for all non-ILE precompilers. With this option, a cursor remains open and accessible only while the program that opened it is on the call stack. When the program ends, the SQL cursor can no longer be used. Prepared statements are also lost when the program ends. Locks, however, remain until the last SQL program on the call stack has completed.

***ENDSQL**

With this option, SQL cursors and prepared statements that are created by a program remain

open until the last SQL program on the call stack has completed. They cannot be used by other programs, only by a different call to the same program. Locks remain until the last SQL program in the call stack completes.

***ENDJOB**

This option allows you to keep SQL cursors, prepared statements, and locks active for the duration of the job. When the last SQL program on the stack has completed, any SQL resources created by *ENDJOB programs are still active. The locks remain in effect. The SQL cursors that were not explicitly closed by the CLOSE, COMMIT, or ROLLBACK statements remain open. The prepared statements are still usable on subsequent calls to the same program.

Database application design tips: Retaining cursor positions across ILE program calls

For ILE program calls, the close SQL cursor (CLOSQLCSR) parameter allows you to specify the scope of the following:

- The cursors
- The prepared statements
- The locks

When used properly, the CLOSQLCSR parameter can reduce the number of SQL OPEN, PREPARE, and LOCK statements needed. It can also simplify applications by allowing you to retain cursor positions across program calls.

***ENDACTGRP**

This is the default for the ILE precompilers. With this option, SQL cursors and prepared statements remain open until the activation group that the program is running under ends. They cannot be used by other programs, only by a different call to the same program. Locks remain until the activation group ends.

***ENDMOD**

With this option, a cursor remains open and accessible only while the module that opened it is active. When the module ends, the SQL cursor can no longer be used. Prepared statements will also be lost when the module ends. Locks, however, remain until the last SQL program in the call stack completes.

Database application design tips: General rules for retaining cursor positions for all program calls

When using programs compiled with either CLOSQLCSR(*ENDPGM) or CLOSQLCSR(*ENDMOD), a cursor must be opened every time the program or module is called, in order to access the data. If the SQL program or module is going to be called several times, and you want to take advantage of a reusable ODP, then the cursor must be explicitly closed before the program or module exits.

Using the CLOSQLCSR parameter and specifying *ENDSQL, *ENDJOB, or *ENDACTGRP, you may not need to run an OPEN and a CLOSE statement on every call. In addition to having fewer statements to run, you can maintain the cursor position between calls to the program or module.

The following examples of SQL statements help demonstrate the advantage of using the CLOSQLCSR parameter:

```
EXEC SQL
  DECLARE DEPTDATA CURSOR FOR
  SELECT EMPNO, LASTNAME
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = :DEPTNUM
END-EXEC.
```

```
EXEC SQL
  OPEN DEPTDATA
```

```

END-EXEC.

EXEC SQL
  FETCH DEPTDATA INTO :EMPNUM, :LNAME
END-EXEC.

EXEC SQL
  CLOSE DEPTDATA
END-EXEC.

```

If this program is called several times from another SQL program, it will be able to use a reusable ODP. This means that, as long as SQL remains active between the calls to this program, the OPEN statement will not require a database open operation. However, the cursor is still positioned to the first result row after each OPEN statement, and the FETCH statement will always return the first row.

In the following example, the CLOSE statement has been removed:

```

EXEC SQL
  DECLARE DEPTDATA CURSOR FOR
  SELECT EMPNO, LASTNAME
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = :DEPTNUM
END-EXEC.

  IF CURSOR-CLOSED IS = TRUE THEN
EXEC SQL
  OPEN DEPTDATA
END-EXEC.

EXEC SQL
  FETCH DEPTDATA INTO :EMPNUM, :LNAME
END-EXEC.

```

If this program is precompiled with the *ENDJOB option or the *ENDACTGRP option and the activation group remains active, the cursor position is maintained. The cursor position is also maintained when the following occurs:

- The program is precompiled with the *ENDSQL option.
- SQL remains active between program calls.

The result of this strategy is that each call to the program retrieves the next row in the cursor. On subsequent data requests, the OPEN statement is unnecessary and, in fact, fails with a -502 SQLCODE. You can ignore the error, or add code to skip the OPEN. You can do this by using a FETCH statement first, and then running the OPEN statement only if the FETCH operation failed.

This technique also applies to prepared statements. A program could first try the EXECUTE, and if it fails, perform the PREPARE. The result is that the PREPARE would only be needed on the first call to the program, assuming the correct CLOSQLCSR option was chosen. Of course, if the statement can change between calls to the program, it should perform the PREPARE in all cases.

The main program could also control this by sending a special parameter on the first call only. This special parameter value would indicate that because it is the first call, the subprogram should perform the OPENS, PREPAREs, and LOCKs.

Note: If you are using COBOL programs, do not use the STOP RUN statement. When the first COBOL program on the call stack ends or a STOP RUN statement runs, a reclaim resource (RCLRSC) operation is done. This operation closes the SQL cursor. The *ENDSQL option does not work as desired.

Chapter 7. Programming techniques for database performance

The following coding tips can help you improve the performance of your SQL queries:

- “Programming techniques for database performance: Use the OPTIMIZE clause”
- “Programming techniques for database performance: Use FETCH FOR n ROWS” on page 124
- “Programming techniques for database performance: Use INSERT n ROWS” on page 125
- “Programming techniques for database performance: Control database manager blocking” on page 125
- “Programming techniques for database performance: Optimize the number of columns that are selected with SELECT statements” on page 126
- “Programming techniques for database performance: Eliminate redundant validation with SQL PREPARE statements” on page 127
- “Programming techniques for database performance: Page interactively displayed data with REFRESH(*FORWARD)” on page 127

Programming techniques for database performance: Use the OPTIMIZE clause

If an application is not going to retrieve the entire result table for a cursor, using the OPTIMIZE clause can improve performance. The query optimizer modifies the cost estimates to retrieve the subset of rows using the value specified on the OPTIMIZE clause.

Assume that the following query returns 1000 rows:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'A00'
  ORDER BY LASTNAME
  OPTIMIZE FOR 100 ROWS
END EXEC.
```

Note: The values that can be used for the OPTIMIZE clause above are 1–9999999 or ALL.

The optimizer calculates the following costs.

The optimize ratio = optimize for n rows value / estimated number of rows in answer set.

Cost using a temporarily created index:

```
Cost to retrieve answer set rows
+ Cost to create the index
+ Cost to retrieve the rows again
  with a temporary index      * optimize ratio
```

Cost using a SORT:

```
Cost to retrieve answer set rows
+ Cost for SORT input processing
+ Cost for SORT output processing * optimize ratio
```

Cost using an existing index:

```
Cost to retrieve answer set rows
using an existing index      * optimize ratio
```

In the previous examples, the estimated cost to sort or to create an index is not adjusted by the optimize ratio. This enables the optimizer to balance the optimization and preprocessing requirements. If the optimize number is larger than the number of rows in the result table, no adjustments are made to the cost estimates. If the OPTIMIZE clause is not specified for a query, a default value is used based on the statement type, value of ALWCPYDTA specified, or output device.

Statement Type	ALWCPYDTA(*OPTIMIZE)	ALWCPYDTA(*YES or *NO)
DECLARE CURSOR	The number or rows in the result table.	3% or the number of rows in the result table.
Embedded Select	2	2
INTERACTIVE Select output to display	3% or the number of rows in the result table.	3% or the number of rows in the result table.
INTERACTIVE Select output to printer or database table	The number of rows in the result table.	The number of rows in the result table.

The OPTIMIZE clause influences the optimization of a query:

- To use an existing index (by specifying a small number).
- To enable the creation of an index or to run a sort or a hash by specifying a large number of possible rows in the answer set.

Programming techniques for database performance: Use FETCH FOR n ROWS

Applications that perform many FETCH statements in succession may be improved by using FETCH FOR n ROWS. With this clause, you can retrieve multiple rows of data from a table and put them into a host structure array or row storage area with a single FETCH. For more information on declaring arrays of host structures or row storage areas, see the SQL Reference book or the individual programming chapters in the SQL Programming with Host Languages book.

An SQL application that uses a FETCH statement without the FOR n ROWS clause can be improved by using the multiple-row FETCH statement to retrieve multiple rows. After the host structure array or row storage area has been filled by the FETCH, the application can loop through the data in the array or storage area to process each of the individual rows. The statement runs faster because the SQL run-time was called only once and all the data was simultaneously returned to the application program.

You can change the application program to allow the database manager to block the rows that the SQL run-time retrieves from the tables. For more information, see “Programming techniques for database performance: Control database manager blocking” on page 125.

You can also use a few techniques to Improve SQL blocking performance when using FETCH FOR n ROWS.

In the following table, the program attempted to FETCH 100 rows into the application. Note the differences in the table for the number of calls to SQL run-time and the database manager when blocking can be performed.

Table 11. Number of Calls Using a FETCH Statement

	Database Manager Not Using Blocking	Database Manager Using Blocking
Single-Row FETCH Statement	100 SQL calls 100 database calls	100 SQL calls 1 database call
Multiple-Row FETCH Statement	1 SQL run-time call 100 database calls	1 SQL run-time call 1 database call

Programming techniques for database performance: Improve SQL blocking performance when using FETCH FOR n ROWS

Special performance considerations should be made for the following points when using FETCH FOR n ROWS. You can improve SQL blocking performance with the following:

- The attribute information in the host structure array or the descriptor associated with the row storage area should match the attributes of the columns retrieved.
- The application should retrieve as many rows as possible with a single multiple-row FETCH call. The blocking factor for a multiple-row FETCH request is not controlled by the system page sizes or the SEQONLY parameter on the OVRDBF command. It is controlled by the number of rows that are requested on the multiple-row FETCH request.
- Single- and multiple-row FETCH requests against the same cursor should not be mixed within a program. If one FETCH against a cursor is treated as a multiple-row FETCH, all fetches against that cursor are treated as multiple-row fetches. In that case, each of the single-row FETCH requests would be treated as a multiple-row FETCH of one row.
- The PRIOR, CURRENT, and RELATIVE scroll options should not be used with multiple-row FETCH statements. To allow random movement of the cursor by the application, the database manager must maintain the same cursor position as the application. Therefore, the SQL run-time treats all FETCH requests against a scrollable cursor with these options specified as multiple-row FETCH requests.

Programming techniques for database performance: Use INSERT n ROWS

Applications that perform many INSERT statements in succession may be improved by using INSERT n ROWS. With this clause, you can insert one or more rows of data from a host structure array into a target table. This array must be an array of structures where the elements of the structure correspond to columns in the target table.

An SQL application that loops over an INSERT...VALUES statement (without the n ROWS clause) can be improved by using the INSERT n ROWS statement to insert multiple rows into the table. After the application has looped to fill the host array with rows, a single INSERT n ROWS statement can be run to insert the entire array into the table. The statement runs faster because the SQL run-time was only called once and all the data was simultaneously inserted into the target table.

In the following table, the program attempted to INSERT 100 rows into a table. Note the differences in the number of calls to SQL run-time and to the database manager when blocking can be performed.

Table 12. Number of Calls Using an INSERT Statement

	Database Manager Not Using Blocking	Database Manager Using Blocking
Single-Row INSERT Statement	100 SQL run-time calls 100 database calls	100 SQL run-time calls 1 database call
Multiple-Row INSERT Statement	1 SQL run-time call 100 database calls	1 SQL run-time call 1 database call

Programming techniques for database performance: Control database manager blocking

To improve performance, the SQL run-time attempts to retrieve and insert rows from the database manager a block at a time whenever possible.

You can control blocking, if desired. Use the SEQONLY parameter on the CL command Override Database File (OVRDBF) before calling the application program that contains the SQL statements. You can also specify the ALWBLK parameter on the CRTSQLxxx commands.

The database manager does not allow blocking in the following situations:

- The cursor is update or delete capable.
- The length of the row plus the feedback information is greater than 32767. The minimum size for the feedback information is 11 bytes. The feedback size is increased by the number of bytes in the key columns for the index used by the cursor and by the number of key columns, if any, that are null capable.
- COMMIT(*CS) is specified, and ALWBLK(*ALLREAD) is not specified.
- COMMIT(*ALL) is specified, and the following are true:
 - A SELECT INTO statement or a blocked FETCH statement is not used
 - The query does not use column functions or specify group by columns.
 - A temporary result table does not have to be created.
- COMMIT(*CHG) is specified, and ALWBLK(*ALLREAD) is not specified.
- The cursor contains at least one subquery and the outermost subselect provided a correlated reference for a subquery or the outermost subselect processed a subquery with an IN, = ANY, or < > ALL subquery predicate operator, which is treated as a correlated reference, and that subquery is not isolatable.

The SQL run-time automatically blocks rows with the database manager in the following cases:

- INSERT

If an INSERT statement contains a select-statement, inserted rows are blocked and not actually inserted into the target table until the block is full. The SQL run-time automatically does blocking for blocked inserts.

Note: If an INSERT with a VALUES clause is specified, the SQL run-time might not actually close the internal cursor that is used to perform the inserts until the program ends. If the same INSERT statement is run again, a full open is not necessary and the application runs much faster.

- OPEN

Blocking is done under the OPEN statement when the rows are retrieved if all of the following conditions are true:

- The cursor is only used for FETCH statements.
- No EXECUTE or EXECUTE IMMEDIATE statements are in the program, or ALWBLK(*ALLREAD) was specified, or the cursor is declared with the FOR FETCH ONLY clause.
- COMMIT(*CHG) and ALWBLK(*ALLREAD) are specified, COMMIT(*CS) and ALWBLK(*ALLREAD) are specified, or COMMIT(*NONE) is specified.

Programming techniques for database performance: Optimize the number of columns that are selected with SELECT statements

The number of columns that you specify in the select list of a SELECT statement causes the database manager to retrieve the data from the underlying tables and map the data into host variables in the application programs. By minimizing the number of columns that are specified, processing unit resource usage can be conserved. Even though it is convenient to code SELECT *, it is far better to explicitly code the columns that are actually required for the application. This is especially important if index-only access is desired or if all of the columns will participate in a sort operation (as happens for SELECT DISTINCT and for SELECT UNION).

This is also important when considering index only access, since you minimize the number of columns in a query and thereby increase the odds that an index can be used to completely satisfy the request for all the data.

Programming techniques for database performance: Eliminate redundant validation with SQL PREPARE statements

The processing which occurs when an SQL PREPARE statement is run is similar to the processing which occurs during precompile processing. The following processing occurs for the statement that is being prepared:

- The syntax is checked.
- The statement is validated to ensure that the usage of objects are valid.
- An access plan is built.

Again when the statement is executed or opened, the database manager will revalidate that the access plan is still valid. Much of this open processing validation is redundant with the validation which occurred during the PREPARE processing. The DLYPRP(*YES) parameter specifies whether PREPARE statements in this program will completely validate the dynamic statement. The validation will be completed when the dynamic statement is opened or executed. This parameter can provide a significant performance enhancement for programs which use the PREPARE SQL statement because it eliminates redundant validation. Programs that specify this precompile option should check the SQLCODE and SQLSTATE after running the OPEN or EXECUTE statement to ensure that the statement is valid. DLYPRP(*YES) will not provide any performance improvement if the INTO clause is used on the PREPARE statement or if a DESCRIBE statement uses the dynamic statement before an OPEN is issued for the statement.

Programming techniques for database performance: Page interactively displayed data with REFRESH(*FORWARD)

In large tables, paging performance is usually degraded because of the REFRESH(*ALWAYS) parameter on the STRSQL command which dynamically retrieves the latest data directly from the table. Paging performance can be improved by specifying REFRESH(*FORWARD).

When interactively displaying data using REFRESH(*FORWARD), the results of a select-statement are copied to a temporary table as you page forward through the display. Other users sharing the table can make changes to the rows while you are displaying the select-statement results. If you page backward or forward to rows that have already been displayed, the rows shown are those in the temporary table instead of those in the updated table.

The refresh option can be changed on the Session Services display.

Chapter 8. General DB2 UDB for iSeries performance considerations

As you code your applications, the following general tips can help you optimize performance:

- “Effects on database performance when using long object names”
- “Effects of precompile options on database performance”
- “Effects of the ALWCPYDTA parameter on database performance” on page 130
- “Tips for using VARCHAR and VARGRAPHIC data types in databases” on page 131

Effects on database performance when using long object names

Long object names are converted internally to system object names when used in SQL statements. This conversion can have some performance impacts.

Qualify the long object name with a library name, and the conversion to the short name happens at precompile time. In this case, there is no performance impact when the statement is executed. Otherwise, the conversion is done at execution time, and has a small performance impact.

Effects of precompile options on database performance

Several precompile options are available for creating SQL programs with improved performance. They are only options because using them may impact the function of the application. For this reason, the default value for these parameters is the value that will ensure successful migration of applications from prior releases. However, you can improve performance by specifying other options. The following table shows these precompile options and their performance impacts.

Some of these options may be suitable for most of your applications. Use the command CRTDUPOBJ to create a copy of the SQL CRTSQLxxx command, and the CHGCMDDFLT command to customize the optimal values for the precompile parameters. The DSPPGM, DSPSRVPGM, DSPMOD, or PRSQLINF commands can be used to show the precompile options that are used for an existing program object.

Precompile Option	Optimal Value	Improvements	Considerations	Related Topics
ALWCPYDTA	*OPTIMIZE (the default)	Queries where the ordering or grouping criteria conflicts with the selection criteria.	A copy of the data may be made when the query is opened.	See “Effects of the ALWCPYDTA parameter on database performance” on page 130.
ALWBLK	*ALLREAD (the default)	Additional read-only cursors use blocking.	ROLLBACK HOLD may not change the position of a read-only cursor. Dynamic processing of positioned updates or deletes might fail.	See “Programming techniques for database performance: Control database manager blocking” on page 125.
CLOSQLCSR	*ENDJOB, *ENDSQL, or *ENDACTGRP	Cursor position can be retained across program invocations.	Implicit closing of SQL cursor is not done when the program invocation ends.	See “Database application design tips: Retaining cursor positions for non-ILE program calls” on page 120.

Precompile Option	Optimal Value	Improvements	Considerations	Related Topics
DLYPRP	*YES	Programs using SQL PREPARE statements may run faster.	Complete validation of the prepared statement is delayed until the statement is run or opened.	See "Programming techniques for database performance: Eliminate redundant validation with SQL PREPARE statements" on page 127.
TGTRLS	*CURRENT (the default)	The precompiler can generate code that will take advantage of performance enhancements available in the current release.	The program object cannot be used on a server from a previous release.	

Effects of the ALWCPYDTA parameter on database performance

Some complex queries can perform better by using a sort or hashing method to evaluate the query instead of using or creating an index. By using the sort or hash, the database manager is able to separate the row selection from the ordering and grouping process. Bitmap processing can also be partially controlled through this parameter. This separation allows the use of the most efficient index for the selection. For example, consider the following SQL statement:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'A00'
  ORDER BY LASTNAME
END-EXEC.
```

The above SQL statement would be written in the following way by using the OPNQRYF command:

```
OPNQRYF FILE(CORPDATA/EMPLOYEE)
  FORMAT(FORMAT1)
  QRYSLT(WORKDEPT *EQ 'A00')
  KEYFLD(LASTNAME)
```

In the above example when ALWCPYDTA(*NO) or ALWCPYDTA(*YES) is specified, the database manager may try to create an index from the first index with a column named LASTNAME, if such an index exists. The rows in the table are scanned, using the index, to select only the rows matching the WHERE condition.

If ALWCPYDTA(*OPTIMIZE) is specified, the database manager uses an index with the first index column of WORKDEPT. It then makes a copy of all of the rows that match the WHERE condition. Finally, it may sort the copied rows by the values in LASTNAME. This row selection processing is significantly more efficient, because the index used immediately locates the rows to be selected.

ALWCPYDTA(*OPTIMIZE) optimizes the total time that is required to process the query. However, the time required to receive the first row may be increased because a copy of the data must be made prior to returning the first row of the result table. This initial change in response time may be important for applications that are presenting interactive displays or that retrieve only the first few rows of the query. The DB2 Universal Database for iSeries query optimizer can be influenced to avoid sorting by using the OPTIMIZE clause. Refer to "Programming techniques for database performance: Use the OPTIMIZE clause" on page 123 for more information.

Queries that involve a join operation may also benefit from ALWCOPYDTA(*OPTIMIZE) because the join order can be optimized regardless of the ORDER BY specification.

Tips for using VARCHAR and VARGRAPHIC data types in databases

Variable-length column (VARCHAR or VARGRAPHIC) support allows you to define any number of columns in a table as variable length. If you use VARCHAR or VARGRAPHIC support, the size of a table can usually be reduced.

Data in a variable-length column is stored internally in two areas: a fixed-length or ALLOCATE area and an overflow area. If a default value is specified, the allocated length is at least as large as the value. The following points help you determine the best way to use your storage area.

When you define a table with variable-length data, you must decide the width of the ALLOCATE area. If the primary goal is:

- **Space saving:** use ALLOCATE(0).
- **Performance:** the ALLOCATE area should be wide enough to incorporate at least 90% to 95% of the values for the column.

It is possible to balance space savings and performance. In the following example of an electronic phone book, the following data is used:

- 8600 names that are identified by: last, first, and middle name
- The Last, First, and Middle columns are variable length.
- The shortest last name is 2 characters; the longest is 22 characters.

This example shows how space can be saved by using variable-length columns. The fixed-length column table uses the most space. The table with the carefully calculated allocate sizes uses less disk space. The table that was defined with no allocate size (with all of the data stored in the overflow area) uses the least disk space.

Variety of Support	Last Name Max/Alloc	First Name Max/Alloc	Middle Name Max/Alloc	Total Physical File Size	Number of Rows in Overflow Space
Fixed Length	22	22	22	567 K	0
Variable Length	40/10	40/10	40/7	408 K	73
Variable-Length Default	40/0	40/0	40/0	373 K	8600

In many applications, performance must be considered. If you use the default ALLOCATE(0), it will double the disk unit traffic. ALLOCATE(0) requires two reads; one to read the fixed-length portion of the row and one to read the overflow space. The variable-length implementation, with the carefully chosen ALLOCATE, minimizes overflow and space and maximizes performance. The size of the table is 28% smaller than the fixed-length implementation. Because 1% of rows are in the overflow area, the access requiring two reads is minimized. The variable-length implementation performs about the same as the fixed-length implementation.

To create the table using the ALLOCATE keyword:

```
CREATE TABLE PHONEDIR
  (LAST   VARCHAR(40) ALLOCATE(10),
   FIRST  VARCHAR(40) ALLOCATE(10),
   MIDDLE VARCHAR(40) ALLOCATE(7))
```

If you are using host variables to insert or update variable-length columns, the host variables should be variable length. Because blanks are not truncated from fixed-length host variables, using fixed-length host variables would cause more rows to spill into the overflow space. This would increase the size of the table.

In this example, fixed-length host variables are used to insert a row into a table:

```
01 LAST-NAME PIC X(40).  
...  
MOVE "SMITH" TO LAST-NAME.  
EXEC SQL  
  INSERT INTO PHONEDIR  
  VALUES(:LAST-NAME, :FIRST-NAME, :MIDDLE-NAME, :PHONE)  
END-EXEC.
```

The host-variable LAST-NAME is not variable length. The string "SMITH", followed by 35 blanks, is inserted into the VARCHAR column LAST. The value is longer than the allocated size of 10. Thirty of thirty-five trailing blanks are in the overflow area.

In this example, variable-length host variables are used to insert a row into a table:

```
01 VLAST-NAME.  
49 LAST-NAME-LEN PIC S9(4) BINARY.  
49 LAST-NAME-DATA PIC X(40).  
...  
MOVE "SMITH" TO LAST-NAME-DATA.  
MOVE 5 TO LAST-NAME-LEN.  
EXEC SQL  
  INSERT INTO PHONEDIR  
  VALUES(:VLAST-NAME, :VFIRST-NAME, :VMIDDLE-NAME, :PHONE)  
END-EXEC.
```

The host variable VLAST-NAME is variable length. The actual length of the data is set to 5. The value is shorter than the allocated length. It can be placed in the fixed portion of the column.

For more information about using variable-length host variables, see the SQL Programming with Host Languages book.

Running the RGZPFM command against tables that contain variable-length columns can improve performance. The fragments in the overflow area that are not in use are compacted by the RGZPFM command. This reduces the read time for rows that overflow, increases the locality of reference, and produces optimal order for serial batch processing.

Choose the appropriate maximum length for variable-length columns. Selecting lengths that are too long increases the process access group (PAG). A large PAG slows performance. A large maximum length makes SEQONLY(*YES) less effective. Variable-length columns longer than 2000 bytes are not eligible as key columns.

Appendix A. Database monitor: DDS

This appendix contains the DDS that is used to create the database monitor physical and logical files:

- “Database monitor physical file DDS”
- “Optional database monitor logical file DDS” on page 140

Database monitor physical file DDS

The following figure shows the DDS that is used to create the QSYS/QAQQDBMN performance statistics physical file.

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

```
A*
A* Database Monitor physical file row format
A*
A      R QQQDBMN          TEXT('Database Monitor +
                        Base Table')
A      QQRID             15P      TEXT('Row +
                        ID') +
                        EDTCDE(4) +
                        COLHDG('Row' 'ID')
A      QQTIME            Z        TEXT('Time row was +
                        created') +
                        COLHDG('Time' +
                        'Row' +
                        'Created')
A      QQJFLD            46H      TEXT('Join Column') +
                        COLHDG('Join' 'Column')
A      QQRDBN            18A      TEXT('Relational +
                        Database Name') +
                        COLHDG('Relational' +
                        'Database' 'Name')
A      QSYS              8A      TEXT('System Name') +
                        COLHDG('System' 'Name')
A      QQJOB             10A      TEXT('Job Name') +
                        COLHDG('Job' 'Name')
A      QQUSER            10A      TEXT('Job User') +
                        COLHDG('Job' 'User')
A      QQJNUM            6A      TEXT('Job Number') +
                        COLHDG('Job' 'Number')
A      QQUCNT            15P      TEXT('Unique Counter') +
                        ALWNULL +
                        COLHDG('Unique' 'Counter')
A      QQUDEF            100A     VARLEN TEXT('User Defined +
                        Column') +
                        ALWNULL +
                        COLHDG('User' 'Defined' +
                        'Column')
A      QQSTN             15P      TEXT('Statement Number') +
                        ALWNULL +
                        COLHDG('Statement' +
                        'Number')
A      QQQDTN            15P      TEXT('Subselect Number') +
                        ALWNULL +
                        COLHDG('Subselect' +
                        'Number')
A      QQQDTL            15P      TEXT('Nested level of +
                        subselect') +
                        ALWNULL +
                        COLHDG('Nested' +
                        'Level of' +
                        'Subselect')
A      QQMATN            15P      TEXT('Subselect of +
```

			materialized view') + ALWNULL + COLHDG('Subselect' + 'Number of' + 'Materialized View')
A	QQMATL	15P	TEXT('Nested level of + Views subselect') + ALWNULL + COLHDG('Nested Level' + 'of View''s' + 'Subselect')
A	QQTLN	10A	TEXT('Library of + Table Queried') + ALWNULL + COLHDG('Library of' + 'Table' + 'Queried')
A	QQTFN	10A	TEXT('Name of + Table Queried') + ALWNULL + COLHDG('Name of' + 'Table' + 'Queried')
A	QQTMN	10A	TEXT('Member of + Table Queried') + ALWNULL + COLHDG('Member of' + 'Table' + 'Queried')
A	QQPTLN	10A	TEXT('Base Library') + ALWNULL + COLHDG('Library of' + 'Base' + 'Table')
A	QQPTFN	10A	TEXT('Base Table') + ALWNULL + COLHDG('Name of' + 'Base' + 'Table')
A	QQPTMN	10A	TEXT('Base Member') + ALWNULL + COLHDG('Member of' + 'Base' + 'Table')
A	QQILNM	10A	TEXT('Library of + Index Used') + ALWNULL + COLHDG('Library of' + 'Index' + 'Used')
A	QQIFNM	10A	TEXT('Name of + Index Used') + ALWNULL + COLHDG('Name of' + 'Index' + 'Used')
A	QQIMNM	10A	TEXT('Member of + Index Used') + ALWNULL + COLHDG('Member of' + 'Index' + 'Used')
A	QQNTNM	10A	TEXT('NLSS Table') + ALWNULL + COLHDG('NLSS' 'Table')
A	QQNLNM	10A	TEXT('NLSS Library') + ALWNULL +

A	QQSTIM	Z	COLHDG('NLSS' 'Library') TEXT('Start timestamp') + ALWNULL + COLHDG('Start' 'Time')
A	QQETIM	Z	TEXT('End timestamp') + ALWNULL + COLHDG('End' 'Time')
A	QQKP	1A	TEXT('Index scan-key positioning') + ALWNULL + COLHDG('Key' 'Positioning')
A	QQKS	1A	TEXT('Key selection') + ALWNULL + COLHDG('Key' 'Selection')
A	QQTOTR	15P	TEXT('Total rows in table') + ALWNULL + COLHDG('Total' + 'Rows in' + 'Table')
A	QQTMPR	15P	TEXT('Number of rows in + temporary') + ALWNULL + COLHDG('Number' + 'of Rows' + 'in Temporary')
A	QQJNP	15P	TEXT('Join Position') + ALWNULL + COLHDG('Join' 'Position')
A	QQEPT	15P	TEXT('Estimated processing + time') + ALWNULL + COLHDG('Estimated' + 'Processing' + 'Time')
A	QQDSS	1A	TEXT('Data space + Selection')
A	QQIDXA	1A	ALWNULL + COLHDG('Data' 'Space' + 'Selection')
A	QQORDG	1A	TEXT('Index advised') + ALWNULL + COLHDG('Index' 'Advised')
A	QQORNG	1A	TEXT('Ordering') + ALWNULL + COLHDG('Ordering')
A	QQGRPG	1A	TEXT('Grouping') + ALWNULL + COLHDG('Grouping')
A	QQJNG	1A	TEXT('Join') + ALWNULL + COLHDG('Join')
A	QQUNIN	1A	TEXT('Union') + ALWNULL + COLHDG('Union')
A	QQSUBQ	1A	TEXT('Subquery') + ALWNULL + COLHDG('Subquery')
A	QQHSTV	1A	TEXT('Host Variables') + ALWNULL + COLHDG('Host' 'Variables')
A	QQRCDS	1A	TEXT('Row Selection') + ALWNULL + COLHDG('Row' 'Selection')
A	QQRCOD	2A	TEXT('Reason Code') + ALWNULL + COLHDG('Reason' 'Code')
A	QQRSS	15P	TEXT('Number of rows + selected or sorted') +

			ALWNULL + COLHDG('Number of' + 'Rows' + 'Selected')
A	QREST	15P	TEXT('Estimated number + of rows selected') + ALWNULL + COLHDG('Estimated' + 'Rows' + 'Selected')
A	QQRIDX	15P	TEXT('Number of entries + in index created') + ALWNULL + COLHDG('Entries in' + 'Index' + 'Created')
A	QQFKEY	15P	TEXT('Estimated keys for + index scan-key positioning') + ALWNULL + COLHDG('Estimated' + 'Entries for' + 'index scan-key positioning')
A	QQKSEL	15P	TEXT('Estimated keys for + key selection') + ALWNULL + COLHDG('Estimated' + 'Entries for' + 'Key Selection')
A	QQAJN	15P	TEXT('Estimated number + of joined rows') + ALWNULL + COLHDG('Estimated' + 'Joined' + 'Rows')
A	QQIDX	1000A	VARLEN(48) + TEXT('Columns + for the index advised') + ALWNULL + COLHDG('Advised' 'Key' + 'Columns')
A	QQC11	1A	ALWNULL
A	QQC12	1A	ALWNULL
A	QQC13	1A	ALWNULL
A	QQC14	1A	ALWNULL
A	QQC15	1A	ALWNULL
A	QQC16	1A	ALWNULL
A	QQC18	1A	ALWNULL
A	QQC21	2A	ALWNULL
A	QQC22	2A	ALWNULL
A	QQC23	2A	ALWNULL
A	QQI1	15P	ALWNULL
A	QQI2	15P	ALWNULL
A	QQI3	15P	ALWNULL
A	QQI4	15P	ALWNULL
A	QQI5	15P	ALWNULL
A	QQI6	15P	ALWNULL
A	QQI7	15P	ALWNULL
A	QQI8	15P	ALWNULL
A	QQI9	15P	TEXT('Thread + Identifier') + ALWNULL + COLHDG('Thread' + 'Identifier')
A	QQIA	15P	ALWNULL
A	QQF1	15P	ALWNULL
A	QQF2	15P	ALWNULL
A	QQF3	15P	ALWNULL

A	QQC61	6A	ALWNULL
A	QQC81	8A	ALWNULL
A	QQC82	8A	ALWNULL
A	QQC83	8A	ALWNULL
A	QQC84	8A	ALWNULL
A	QQC101	10A	ALWNULL
A	QQC102	10A	ALWNULL
A	QQC103	10A	ALWNULL
A	QQC104	10A	ALWNULL
A	QQC105	10A	ALWNULL
A	QQC106	10A	ALWNULL
A	QQC181	18A	ALWNULL
A	QQC182	18A	ALWNULL
A	QQC183	18A	ALWNULL
A	QQC301	30A	VARLEN(10) ALWNULL
A	QQC302	30A	VARLEN(10) ALWNULL
A	QQC303	30A	VARLEN(10) ALWNULL
A	QQ1000	1000A	VARLEN(48) ALWNULL
A	QQTIM1	Z	ALWNULL
A	QQTIM2	Z	ALWNULL
A*			
A*	New columns added for Visual Explain		
A*			
A	QVQTBL	128A	VARLEN(10) + TEXT('Queried Table, + Long Name') + ALWNULL + COLHDG('Queried' + 'Table' + 'Long Name')
A	QVQLIB	128A	VARLEN(10) + TEXT('Queried Library, + Long Name') + ALWNULL + COLHDG('Queried' + 'Library' + 'Long Name')
A	QVPTBL	128A	VARLEN(10) + TEXT('Base Table, + Long Name') + ALWNULL + COLHDG('Base' + 'Table' + 'Long Name')
A	QVPLIB	128A	VARLEN(10) + TEXT('Base Library, + Long Name') + ALWNULL + COLHDG('Base' + 'Library' + 'Long Name')
A	QVINAM	128A	VARLEN(10) + TEXT('Index Used, + Long Name') + ALWNULL + COLHDG('Index' + 'Used' + 'Long Name')
A	QVILIB	128A	VARLEN(10) + TEXT('Index Used, + Library Name') + ALWNULL + COLHDG('Index' + 'Used' + 'Library' + 'Name')
A	QVQTBLI	1A	TEXT('Table Long +

			Required')
			ALWNULL +
			COLHDG('Table' +
			'Long' +
			'Required')
A	QVPTBLI	1A	TEXT('Base Long +
			Required')
			ALWNULL +
			COLHDG('Base' +
			'Long' +
			'Required')
A	QVINAMI	1A	TEXT('Index Long +
			Required')
			ALWNULL +
			COLHDG('Index' +
			'Long' +
			'Required')
A	QVBNDY	1A	TEXT('I/O or CPU +
			Bound') +
			ALWNULL +
			COLHDG('I/O or CPU' +
			'Bound')
A	QVJFANO	1A	TEXT('Join +
			Fan out') +
			ALWNULL +
			COLHDG('Join' +
			'Fan' +
			'Out')
A	QVPARPF	1A	TEXT('Parallel +
			Pre-Fetch') +
			ALWNULL +
			COLHDG('Parallel' +
			'Pre-Fetch')
A	QVPARPL	1A	TEXT('Parallel +
			Preload') +
			ALWNULL +
			COLHDG('Parallel' +
			'Preload')
A	QVC11	1A	ALWNULL
A	QVC12	1A	ALWNULL
A	QVC13	1A	ALWNULL
A	QVC14	1A	ALWNULL
A	QVC15	1A	ALWNULL
A	QVC16	1A	ALWNULL
A	QVC17	1A	ALWNULL
A	QVC18	1A	ALWNULL
A	QVC19	1A	ALWNULL
A	QVC1A	1A	ALWNULL
A	QVC1B	1A	ALWNULL
A	QVC1C	1A	ALWNULL
A	QVC1D	1A	ALWNULL
A	QVC1E	1A	ALWNULL
A	QVC1F	1A	ALWNULL
A	QWC11	1A	ALWNULL
A	QWC12	1A	ALWNULL
A	QWC13	1A	ALWNULL
A	QWC14	1A	ALWNULL
A	QWC15	1A	ALWNULL
A	QWC16	1A	ALWNULL
A	QWC17	1A	ALWNULL
A	QWC18	1A	ALWNULL
A	QWC19	1A	ALWNULL
A	QWC1A	1A	ALWNULL
A	QWC1B	1A	ALWNULL
A	QWC1C	1A	ALWNULL
A	QWC1D	1A	ALWNULL
A	QWC1E	1A	ALWNULL

A	QWC1F	1A	ALWNULL
A	QVC21	2A	ALWNULL
A	QVC22	2A	ALWNULL
A	QVC23	2A	ALWNULL
A	QVC24	2A	ALWNULL
A	QVCTIM	15P	TEXT('Cumulative + Time') + ALWNULL + COLHDG('Estimated' + 'Cumulative' + 'Time')
A	QVPARD	15P	TEXT('Parallel Degree, + Requested') + ALWNULL + COLHDG('Parallel' + 'Degree' + 'Requested')
A	QVPARU	15P	TEXT('Parallel Degree, + Used') + ALWNULL + COLHDG('Parallel' + 'Degree' + 'Used')
A	QVPARRC	15P	TEXT('Parallel Limited, + Reason Code') + ALWNULL + COLHDG('Parallel' + 'Limited' + 'Reason Code')
A	QVRCNT	15P	TEXT('Refresh Count') + ALWNULL + COLHDG('Refresh' + 'Count')
A	QVFILES	15P	TEXT('Number of, + Tables Joined') + ALWNULL + COLHDG('Number of' + 'Tables' + 'Joined')
A	QVP151	15P	ALWNULL
A	QVP152	15P	ALWNULL
A	QVP153	15P	ALWNULL
A	QVP154	15P	ALWNULL
A	QVP155	15P	ALWNULL
A	QVP156	15P	ALWNULL
A	QVP157	15P	ALWNULL
A	QVP158	15P	ALWNULL
A	QVP159	15P	ALWNULL
A	QVP15A	15P	TEXT('Decomposed' + 'Subselect Number') + ALWNULL + COLHDG('Decomposed' 'Subselect' + 'Number')
A	QVP15B	15P	TEXT('Number of' + 'Decomposed + Subselects') + ALWNULL + COLHDG('Number of' + 'Decomposed' + Subselects')
A	QVP15C	15P	TEXT('Decomposed' + 'Subselect + Reason code') + ALWNULL + COLHDG('Decomposed' + 'Reason' + 'Code')
A	QVP15D	15P	TEXT('Number of first' + 'Decomposed + Subselect') +

			ALWNULL + COLHDG('Starting' + 'Decomposed' + 'Subselect'
A	QVP15E	15P	TEXT('Materialized Union' + 'Level')
			ALWNULL + COLHDG('Materialized' + 'Union' + 'Level')
A	QVP15F	15P	ALWNULL
A	QVC41	4A	ALWNULL
A	QVC42	4A	ALWNULL
A	QVC43	4A	ALWNULL
A	QVC44	4A	ALWNULL
A	QVC81	8A	ALWNULL
A	QVC82	8A	ALWNULL
A	QVC83	8A	ALWNULL
A	QVC84	8A	ALWNULL
A	QVC85	8A	ALWNULL
A	QVC86	8A	ALWNULL
A	QVC87	8A	ALWNULL
A	QVC88	8A	ALWNULL
A	QVC101	10A	ALWNULL
A	QVC102	10A	ALWNULL
A	QVC103	10A	ALWNULL
A	QVC104	10A	ALWNULL
A	QVC105	10A	ALWNULL
A	QVC106	10A	ALWNULL
A	QVC107	10A	ALWNULL
A	QVC108	10A	ALWNULL
A	QVC1281	128A	VARLEN(10) ALWNULL
A	QVC1282	128A	VARLEN(10) ALWNULL
A	QVC1283	128A	VARLEN(10) ALWNULL
A	QVC1284	128A	VARLEN(10) ALWNULL
A	QVC3001	300A	VARLEN(32) ALWNULL
A	QVC3002	300A	VARLEN(32) ALWNULL
A	QVC3003	300A	VARLEN(32) ALWNULL
A	QVC3004	300A	VARLEN(32) ALWNULL
A	QVC3005	300A	VARLEN(32) ALWNULL
A	QVC3006	300A	VARLEN(32) ALWNULL
A	QVC3007	300A	VARLEN(32) ALWNULL
A	QVC3008	300A	VARLEN(32) ALWNULL
A	QVC5001	500A	VARLEN(32) ALWNULL
A	QVC5002	500A	VARLEN(32) ALWNULL
A	QVC1000	1000A	VARLEN(48) ALWNULL
A	QWC1000	1000A	VARLEN(48) ALWNULL

Optional database monitor logical file DDS

The following examples show the different optional logical files that you can create with the DDS shown. The column descriptions are explained in the tables following each example. These tables are not shipped with the server, and you must create them, if you choose to do so. These files are optional and are not required for analyzing monitor data.

- “Database monitor logical table 1000 - Summary Row for SQL Information” on page 141
- “Database monitor logical table 3000 - Summary Row for Table Scan” on page 152
- “Database monitor logical table 3001 - Summary Row for Index Used” on page 157
- “Database monitor logical table 3002 - Summary Row for Index Created” on page 163
- “Database monitor logical table 3003 - Summary Row for Query Sort” on page 170
- “Database monitor logical table 3004 - Summary Row for Temp Table” on page 174
- “Database monitor logical table 3005 - Summary Row for Table Locked” on page 179
- “Database monitor logical table 3006 - Summary Row for Access Plan Rebuilt” on page 182

- “Database monitor logical table 3007 - Summary Row for Optimizer Timed Out” on page 185
- “Database monitor logical table 3008 - Summary Row for Subquery Processing” on page 188
- “Database monitor logical table 3010 - Summary for HostVar & ODP Implementation” on page 189
- “Database monitor logical table 3014 - Summary Row for Generic QQ Information” on page 190
- “Database monitor logical table 3015 - Summary Row for Statistics Information” on page 197
- “Database monitor logical table 3018 - Summary Row for STRDBMON/ENDDDBMON” on page 200
- “Database monitor logical table 3019 - Detail Row for Rows Retrieved” on page 201
- “Database monitor logical table 3021 - Summary Row for Bitmap Created” on page 202
- “Database monitor logical table 3022 - Summary Row for Bitmap Merge” on page 205
- “Database monitor logical table 3023 - Summary for Temp Hash Table Created” on page 208
- “Database monitor logical table 3025 - Summary Row for Distinct Processing” on page 212
- “Database monitor logical table 3027 - Summary Row for Subquery Merge” on page 213
- “Database monitor logical table 3028 - Summary Row for Grouping” on page 217

Database monitor logical table 1000 - Summary Row for SQL Information

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
| |
| |A*
| |A*
| |A* DB Monitor logical table 1000 - Summary Row for SQL Information
| |A*
| |A          R QQQ1000          PTABLE(*CURLIB/QAQQDBMN)
| |A          QQRID
| |A          QQTIME
| |A          QQJFLD
| |A          QQRDBN
| |A          QQSYS
| |A          QQJOB
| |A          QQUSER
| |A          QQJNUM
| |A          QQTHRD          RENAME(QQI9) +
| |          COLHDG('Thread' +
| |          'Identifier')
| |
| |A          QQUCNT
| |A          QQRcnt          RENAME(QQI5) +
| |          COLHDG('Refresh' +
| |          'Counter')
| |
| |A          QQUDEF
| |A*
| |A* Information about the SQL statement executed
| |A*
| |A          QQSTN
| |A          QQSTF          RENAME(QQC11) +
| |          COLHDG('Statement' +
| |          'Function')
| |
| |A          QQSTOP          RENAME(QQC21) +
| |          COLHDG('Statement' +
| |          'Operation')
| |
| |A          QQSTTY          RENAME(QQC12) +
| |          COLHDG('Statement' 'Type')
| |
| |A          QQPARS          RENAME(QQC13) +
| |          COLHDG('Parse' 'Required')
| |
| |A          QQPNAM          RENAME(QQC103) +
| |          COLHDG('Package' 'Name')
| |
| |A          QQPLIB          RENAME(QQC104) +
| |          COLHDG('Package' 'Library')
| |
| |A          QQCNAM          RENAME(QQC181) +
| |          COLHDG('Cursor' 'Name')
| |
| |A          QQSNAME          RENAME(QQC182) +

```


		COLHDG('Statement' 'Name')
A	QQSTIM	
A	QQSTTX	RENAME(QQ1000) + COLHDG('Statement' 'Text')
A	QQSTOC	RENAME(QQC14) + COLHDG('Statement' + 'Outcome')
A	QQROWR	RENAME(QQI2) + COLHDG('Rows' 'Returned')
A	QQDYNR	RENAME(QQC22) + COLHDG('Dynamic' 'Replan')
A	QQDACV	RENAME(QQC16) + COLHDG('Data' 'Conversion')
A	QQTTIM	RENAME(QQI4) + COLHDG('Total' 'Time' + 'Milliseconds')
A	QQROWF	RENAME(QQI3) + COLHDG('Rows' 'Fetched')
A	QQETIM	
A	QQTTIMM	RENAME(QQI6) + COLHDG('Total' 'Time') 'Microseconds')
A	QQSTMTLN	RENAME(QQI7) + COLHDG('Total' + 'Statement' + 'Length')
A	QQIUCNT	RENAME(QQI1) + COLHDG('Insert' 'Unique') 'Count') A*
A	QQADDTXT	RENAME(QWC14) + COLHDG('Additional' 'SQL') 'Text')
A*		
A*		
A*	Additional information about the SQL statement executed	
A*		
A	QVSQCOD	RENAME(QQI8) + COLHDG('SQL' + 'Return' + 'Code')
A	QVSQST	RENAME(QQC81) + COLHDG('SQLSTATE')
A	QVCLSCR	RENAME(QVC101) + COLHDG('CLOSQCSR' + 'Setting')
A	QVALWCY	RENAME(QVC11) + COLHDG('ALWCPYDTA' + 'Setting')
A	QVPSUDO	RENAME(QVC12) + COLHDG('Pseudo' + 'Open')
A	QVPSUDC	RENAME(QVC13) + COLHDG('Pseudo' + 'Close')
A	QVODPI	RENAME(QVC14) + COLHDG('ODP' + 'Implementation')
A	QVDYNCS	RENAME(QVC21) + COLHDG('Dynamic' + 'Replan' + 'Subtype Code')
A	QVCMMT	RENAME(QVC41) + COLHDG('Commit' + 'Level')
A	QVBLKE	RENAME(QVC15) + COLHDG('Blocking' + 'Enabled')

A	QVDLYPR	RENAME(QVC16) + COLHDG('Delay' + 'Prep')
A	QVEXPLF	RENAME(QVC1C) + COLHDG('SQL' + 'Statement' + 'Explainable')
A	QVNAMC	RENAME(QVC17) + COLHDG('Naming' + 'Convention')
A	QVDYNTY	RENAME(QVC18) + COLHDG('Type of' + 'Dynamic' + 'Processing')
A	QVOLOB	RENAME(QVC19) + COLHDG('Optimize' + 'LOB' + 'Data Types')
A	QVUSRP	RENAME(QVC1A) + COLHDG('User' + 'Profile')
A	QVDUSRP	RENAME(QVC1B) + COLHDG('Dynamic' + 'User' + 'Profile')
A	QVDFTCL	RENAME(QVC1281) + COLHDG('Default' + 'Collection')
A	QVPROCN	RENAME(QVC1282) + COLHDG('Procedure' + 'Name on' + 'CALL')
A	QVPROCL	RENAME(QVC1283) + COLHDG('Procedure' + 'Library on' + 'CALL')
A	QVSPATH	RENAME(QVC1000) + COLHDG('SQL' + 'Path')
A	QVSPATHB	RENAME(QwC1000) + COLHDG('SQL' + 'Path' + 'Continued')
A	QVSPATHC	RENAME(QVC5001) + COLHDG('SQL' + 'Path' + 'Continued')
A	QVSPATHD	RENAME(QVC5002) + COLHDG('SQL' + 'Path' + 'Continued')
A	QVSPATHE	RENAME(QVC3001) + COLHDG('SQL' + 'Path' + 'Continued')
A	QVSPATHF	RENAME(QVC3002) + COLHDG('SQL' + 'Path' + 'Continued')
A	QVSPATHG	RENAME(QVC30013) + COLHDG('SQL' + 'Path' + 'Continued')
A	QVSCHEM	RENAME(QVC1284) + COLHDG('SQL' + 'Schema')
A*		

A* Environmental information about the SQL statement executed

A*

A	QVDFMT	RENAME(QVC42) + COLHDG('Date' + 'Format')
A	QVDSEP	RENAME(QVC11) + COLHDG('Date' + 'Separator')
A	QVTFMT	RENAME(QVC43) + COLHDG('Time' + 'Format')
A	QVTSEP	RENAME(QVC12) + COLHDG('Time' + 'Separator')
A	QVDPNT	RENAME(QVC13) + COLHDG('Decimal' + 'Point')
A	QVSRTSQ	RENAME(QVC104) + COLHDG('Sort' + 'Sequence' + 'Table')
A	QVSRTSL	RENAME(QVC105) + COLHDG('Sort' + 'Sequence' + 'Library')
A	QVLNGID	RENAME(QVC44) + COLHDG('Language' + 'ID')
A	QVCNTID	RENAME(QVC23) + COLHDG('Country' + 'ID')
A	QVFNROW	RENAME(QQIA) + COLHDG('FIRST n' + 'ROWS Value')
A	QVOPTRW	RENAME(QQF1) + COLHDG('OPTIMIZE FOR' + 'n ROWS Value')
A	QVRAPRC	RENAME(QVC22) + COLHDG('SQL Access' + 'Plan Rebuild' + 'Reason Code')
A	QVNOSV	RENAME(QVC24) + COLHDG('Access Plan' + 'Not Saved' + 'Reason Code')
A	QVCTXT	RENAME(QVC81) + COLHDG('Transaction' + 'Context' + 'ID')
A	QVAGMRK	RENAME(QVP152) + COLHDG('Activation' + 'Group' + 'Mark')
A	QVCURTHR	RENAME(QVP153) + COLHDG('Open Cursor' + 'Threshold')
A	QVCURCNT	RENAME(QVP154) + COLHDG('Open Cursor' + 'Close' + 'Count')
A	QVLCKLMT	RENAME(QVP155) + COLHDG('Commit' + 'Lock' + 'Limit')
A	QVSQLMIXED	RENAME(QWC15) + COLHDG('SQL' + 'Mixed' +

```

|           A           QVSQLSUPP           'Constants')
|           |           |           RENAME(QWC16) +
|           |           |           COLHDG('SQL' +
|           |           |           'Suppress' +
|           |           |           'Warnings')
|           A           QVSQLASCII          RENAME(QWC17) +
|           |           |           COLHDG('SQL' +
|           |           |           'Translate' +
|           |           |           'ASCII')
|           A           QVSQLCACHE          RENAME(QWC18) +
|           |           |           COLHDG('SQL' +
|           |           |           'Statement' +
|           |           |           'Cache')
|           A           K QQJFL
|           A           S QQRID            CMP(EQ 1000)
|

```

Table 13. QQQ1000 - Summary row for SQL Information

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQRcnt	QQI5	Unique refresh counter
QQUDEF	QQUDEF	User defined column
QQSTN	QQSTN	Statement number (unique per statement)
QQSTF	QQC11	Statement function: <ul style="list-style-type: none"> • S - Select • U - Update • I - Insert • D - Delete • L - Data definition language • O - Other

Table 13. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QQSTOP	QQC21	Statement operation: <ul style="list-style-type: none"> • AL - Alter table • CA - Call • CC - Create collection • CD - Create type • CF - Create function • CG - Create trigger • CI - Create index • CL - Close • CM - Commit • CN - Connect • CO - Comment on • CP - Create procedure • CS - Create alias/synonym • CT - Create table • CV - Create view • DE - Describe • DI - Disconnect • DL - Delete • DM - Describe parameter marker • DP - Declare procedure • DR - Drop • DT - Describe table • EI - Execute immediate • EX - Execute • FE - Fetch • FL - Free locator • GR - Grant • HC - Hard close • HL - Hold locator • IN - Insert • JR - Server job reused • LK - Lock • LO - Label on • MT - More text • OP - Open • PD - Prepare and describe • PR - Prepare • RB - Rollback Savepoint • RE - Release • RO - Rollback

Table 13. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QQSTOP (continued)	QQC21	<ul style="list-style-type: none"> • RS - Release Savepoint • RT - Rename table • RV - Revoke • SA - Savepoint • SC - Set connection • SI - Select into • SP - Set path • SR - Set result set • SS - Set current schema • ST - Set transaction • SV - Set variable • UP - Update • VI - Values into
QQSTTY	QQC12	Statement type: <ul style="list-style-type: none"> • D - Dynamic statement • S - Static statement
QQPARS	QQC13	Parse required (Y/N)
QQPNAM	QQC103	Name of the package or name of the program that contains the current SQL statement
QQPLIB	QQC104	Name of the library containing the package
QQCNAM	QQC181	Name of the cursor corresponding to this SQL statement, if applicable
QQSNAM	QQC182	Name of statement for SQL statement, if applicable
QQSTIM	QQSTIM	Time this statement entered
QQSTTX	QQ1000	Statement text
QQSTOC	QQC14	Statement outcome <ul style="list-style-type: none"> • S - Successful • U - Unsuccessful
QQROWR	QQI2	Number of result rows returned

Table 13. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QQDYNR	QQC22	<p>Dynamic replan (access plan rebuilt)</p> <ul style="list-style-type: none"> • NA - No replan. • NR - SQL QDT rebuilt for new release. • A1 - A table or member is not the same object as the one referenced when the access plan was last built. Some reasons why they could be different are: <ul style="list-style-type: none"> – Object was deleted and recreated. – Object was saved and restored. – Library list was changed. – Object was renamed. – Object was moved. – Object was overridden to a different object. – This is the first run of this query after the object containing the query has been restored. • A2 - Access plan was built to use a reusable Open Data Path (ODP) and the optimizer chose to use a non-reusable ODP for this call. • A3 - Access plan was built to use a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP for this call. • A4 - The number of rows in the table member has changed by more than 10% since the access plan was last built. • A5 - A new index exists over one of the tables in the query. • A6 - An index that was used for this access plan no longer exists or is no longer valid. • A7 - OS/400 Query requires the access plan to be rebuilt because of system programming changes. • A8 - The CCSID of the current job is different than the CCSID of the job that last created the access plan. • A9 - The value of one or more of the following is different for the current job than it was for the job that last created this access plan: <ul style="list-style-type: none"> – date format – date separator – time format – time separator • AA - The sort sequence table specified is different than the sort sequence table that was used when this access plan was created. • AB - Storage pool changed or DEGREE parameter of CHGQRYA command changed. • AC - The system feature DB2 multisystem has been installed or removed. • AD - The value of the degree query attribute has changed. • AE - A view is either being opened by a high level language or a view is being materialized. • AF - A user-defined type or user-defined function is not the same object as the one referred to in the access plan, or, the SQL Path is not the same as when the access plan was built.

Table 13. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QQDYNR (continued)	QQC22	<ul style="list-style-type: none"> B0 - The options specified have changed as a result of the query options file. B1 - The access plan was generated with a commitment control level that is different in the current job. B2 - The access plan was generated with a static cursor answer set size that is different than the previous access plan.
QQDACV	QQC16	Data conversion <ul style="list-style-type: none"> N - No. 0 - Not applicable. 1 - Lengths do not match. 2 - Numeric types do not match. 3 - C host variable is NUL-terminated. 4 - Host variable or column is variable length and the other is not variable length. 5 - CCSID conversion. 6 - DRDA and NULL capable, variable length, contained in a partial row, derived expression, or blocked fetch with not enough host variables. 7 - Data, time, or timestamp column. 8 - Too many host variables. 9 - Target table of an insert is not an SQL table.
QOTTIM	QQI4	Total time for this statement, in milliseconds. For fetches, this includes all fetches for this OPEN of the cursor.
QQRWF	QQI3	Total rows fetched for cursor
QQETIM	QQETIM	Time SQL request completed
QTTIMM	QQI6	Total time for this statement, in microseconds. For fetches, this includes all fetches for this OPEN of the cursor.
QQSTMTLN	QQI7	Length of SQL Statement
QQIUCNT	QQI1	Unique query count for the QDT associated with the INSERT. QQUCNT contains the unique query count for the QDT associated with the WHERE part of the statement.
QVSQCOD	QQI8	SQL return code
QVSQST	QQC81	SQLSTATE
QVCLSCR	QVC101	Close Cursor. Possible values are: <ul style="list-style-type: none"> *ENDJOB - SQL cursors are closed when the job ends. *ENDMOD - SQL cursors are closed when the module ends *ENDPGM - SQL cursors are closed when the program ends. *ENDSQL - SQL cursors are closed when the first SQL program on the call stack ends. *ENDACTGRP - SQL cursors are closed when the activation group ends.
QVALWCY	QVC11	ALWCPYDTA setting (Y/N/O) <ul style="list-style-type: none"> Y - A copy of the data may be used. N - Cannot use a copy of the data. O - The optimizer can choose to use a copy of the data for performance.

Table 13. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVPSUDO	QVC12	Pseudo Open (Y/N) for SQL operations that can trigger opens. <ul style="list-style-type: none"> • OP - Open • IN - Insert • UP - Update • DL - Delete • SI - Select Into • SV - Set • VI - Values into For all operations it can be blank.
QVPSUDC	QVC13	Pseudo Close (Y/N) for SQL operations that can trigger a close. <ul style="list-style-type: none"> • CL - Close • IN - Insert • UP - Update • DL - Delete • SI - Select Into • SV - Set • VI - Values into For all operations it can be blank.
QVODPI	QVC14	ODP implementation <ul style="list-style-type: none"> • R - Reusable ODP • N - Nonreusable ODP • ' ' - Column not used
QQDYNSC	QVC21	Dynamic replan, subtype reason code
QVCMMT	QVC41	Commitment control level. Possible values are: <ul style="list-style-type: none"> • NC • UR • CS • CSKL • RS • RR
QVBLKE	QVC15	Type of blocking . Possible value are: <ul style="list-style-type: none"> • S - Single row, ALWBLK(*READ) • F - Force one row, ALWBLK(*NONE) • L - Limited block, ALWBLK(*ALLREAD)
QVDLYPR	QVC16	Delay Prep (Y/N)
QVEXPLF	QVC1C	The SQL statement is explainable (Y/N).
QVNAMEC	QVC17	Naming convention. Possibles values: <ul style="list-style-type: none"> • N - System naming convention • S - SQL naming convention
QVDYNTY	QVC18	Type of dynamic processing. <ul style="list-style-type: none"> • E - Extended dynamic • S - System wide cache • L - Local prepared statement

Table 13. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVOLOB	QVC19	Optimize LOB data types (Y/N)
QVUSRP	QVC1A	User profile used when compiled programs are executed. Possible values are: <ul style="list-style-type: none"> • N = User Profile is determined by naming conventions. For *SQL, USRPRF(*OWNER) is used. For *SYS, USRPRF(*USER) is used. • U = USRPRF(*USER) is used. • O = USRPRF(*OWNER) is used.
QVDUSR	QVC1B	User profile used for dynamic SQL statements. <ul style="list-style-type: none"> • U = USRPRF(*USER) is used. • O = USRPRF(*OWNER) is used.
QVDFTCL	QVC1281	Name of the default collection.
QVPROC	QVC1282	Procedure name on CALL to SQL.
QVPROCL	QVC1283	Procedure library on CALL to SQL.
QVSPATH	QVC1000	Path used to find procedures, functions, and user defined types for static SQL statements.
QVSPATHB	QWC1000	Continuation of SQL path, if needed. Contains bytes 1001-2000 of the SQL path.
QVSPATHC	QWC5001	Continuation of SQL path, if needed. Contains bytes 2001-2500 of the SQL path.
QVSPATHD	QWC5002	Continuation of SQL path, if needed. Contains bytes 2501-3000 of the SQL path.
QVSPATHE	QWC3001	Continuation of SQL path, if needed. Contains bytes 3001-3300 of the SQL path.
QVSPATHF	QWC3002	Continuation of SQL path, if needed. Contains bytes 3301-3600 of the SQL path.
QVSPATHG	QWC3003	Continuation of SQL path, if needed. Contains bytes 3601-3900 of the SQL path.
QVSCHEM	QVC1284	SQL Current Schema
QVDFMT	QVC42	Date Format. Possible values are: <ul style="list-style-type: none"> • ISO • USA • EUR • JIS • MDY • DMY • YMD
QVDSEP	QWC11	Date Separator. Possible values are: <ul style="list-style-type: none"> • "/" • "." • "," • "-" • " "

Table 13. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVTFMT	QVC43	Time Format. Possible values are: <ul style="list-style-type: none"> • ISO • USA • EUR • JIS • HMS
QVTSEP	QWC12	Time Separator. Possible values are: <ul style="list-style-type: none"> • ". " • " " • " " • " "
QVDPNT	QWC13	Decimal Point. Possible values are: <ul style="list-style-type: none"> • ". " • " "
QVSRTSQ	QVC104	Sort Sequence Table
QVSRTSL	QVC105	Sort Sequence Library
QVLNGID	QVC44	Language ID
QVCNTID	QVC23	Country ID
QVFNROW	QQIA	Value specified on the FIRST n ROWS clause.
QVOPTRW	QQF1	Value specified on the OPTIMIZE FOR n ROWS clause.
QVRAPRC	QVC22	SQL access plan rebuild reason code. Possible reasons are: (add list of reasons)
QVNOSV	QVC24	Access plan not saved reason code. Possible reasons are: (add list of reasons)
QVCTXT	QVC81	Transaction context ID.
QVAGMRK	QVP152	Activation Group Mark
QVCCURTHR	QVP153	Open cursor threshold
QVCCURCNT	QVP154	Open cursor close count
QVLCKLMT	QVP155	Commitment control lock limit
QVSQLMIXED	QWC15	Using SQL mixed constants (Y/N)
QVSQLSUPP	QWC16	Suppress SQL warning messages (Y/N)
QVSQLASCII	QWC17	Translate ASCII to job (Y/N)
QVSQLCACHE	QWC18	Using system-wide SQL statement cache (Y/N)

Database monitor logical table 3000 - Summary Row for Table Scan

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|   A*
|   A* DB Monitor logical table 3000 - Summary Row for Table Scan
|   A*
|   A      R QQQ3000          PTABLE(*CURLIB/QAQQDBMN)
|   A      QQRID
|   A      QQTIME
|   A      QQJFLD
|   A      QQRDBN

```

	A	QSYS	
	A	QQJOB	
	A	QQUSER	
	A	QQJNUM	
	A	QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')
	A	QQUCNT	
	A	QQUDEF	
	A	QQQDTN	
	A	QQQDTL	
	A	QQMATN	
	A	QQMATL	
	A	QQMATULVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')
	A	QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')
	A	QQQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')
	A	QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
	A	QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
	A	QQTLN	
	A	QQTFN	
	A	QQTMN	
	A	QQPTLN	
	A	QQPTFN	
	A	QQPTMN	
	A	QQTOTR	
	A	QQREST	
	A	QQAJN	
	A	QQEPT	
	A	QQJNP	
	A	QQJNDS	RENAME(QQI1) + COLHDG('Data Space' + 'Number')
	A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
	A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
	A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
	A	QQIDXK	RENAME(QQI2) + COLHDG('Advised' + 'Primary' + 'Keys')
	A	QQDSS	
	A	QQIDXA	
	A	QQRCD	
	A	QQIDXD	
	A	QVQTBL	
	A	QVQLIB	
	A	QVPTBL	
	A	QVPLIB	
	A	QVBNDY	
	A	QVRCNT	

```

|      A      QVJFANO
|      A      QVFILES
|      A      QVPARPF
|      A      QVPARPL
|      A      QVPARD
|      A      QVPARU
|      A      QVPARRC
|      A      QVCTIM
|      A      QVSKIPS          RENAME(QQC11) +
|                               COLHDG('Skip' +
|                               'Sequential')
|      A      QVTBLSZ          RENAME(QQI3) +
|                               COLHDG('Actual' +
|                               'Table' 'Size')
|      A      QVTSFLDS          RENAME(QVC3001) +
|                               COLHDG('Columns for' +
|                               'Data Space' +
|                               'Selection')
|      A      QVDVFLD          RENAME(QQC14) +
|                               COLHDG('Derived' +
|                               'Column' +
|                               'Selection')
|      A      QVDVFLDS          RENAME(QVC3002) +
|                               COLHDG('Columns for' +
|                               'Derived' +
|                               'Selection')
|      A      QVRDRG           RENAME(QQC18) +
|                               COLHDG('Read' +
|                               'Trigger')
|      A      QVCARD           RENAME(QVP157) +
|                               COLHDG('Cardinality')
|      A      QVUTSP           RENAME(QVC1281) +
|                               COLHDG('Specific' +
|                               'Name')
|      A      QVULSP           RENAME(QQC1282) +
|                               COLHDG('Specific' +
|                               'Schema')
|      A      K QQJFLD
|      A      S QQRID          CMP(EQ 3000)

```

Table 14. QQQ3000 - Summary Row for Table Scan

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QQSYS	QQSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number

Table 14. QQQ3000 - Summary Row for Table Scan (continued)

Logical Column Name	Physical Column Name	Description
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQTLN	QQTLN	Library of table queried
QQTFN	QQTFN	Name of table queried
QQTMN	QQTMN	Member name of table queried
QQPTLN	QQPTLN	Library name of base table
QQPTFN	QQPTFN	Name of base table for table queried
QQPTMN	QQPTMN	Member name of base table
QQTOTR	QQTOTR	Total rows in table
QQREST	QQREST	Estimated number of rows selected
QQAJN	QQAJN	Estimated number of joined rows
QQEPT	QQEPT	Estimated processing time, in seconds
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI1	dataspace number
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
QQIDXK	QQI2	Number of advised columns that use index scan-key positioning
QQDSS	QQDSS	dataspace selection <ul style="list-style-type: none"> • Y - Yes • N - No

Table 14. QQQ3000 - Summary Row for Table Scan (continued)

Logical Column Name	Physical Column Name	Description
QQIDXA	QQIDXA	Index advised <ul style="list-style-type: none"> • Y - Yes • N - No
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> • T1 - No indexes exist. • T2 - Indexes exist, but none could be used. • T3 - Optimizer chose table scan over available indexes.
QQIDXD	QQIDXD	Columns for the index advised
QVQTBL	QVQTBL	Queried table, long name
QVQLIB	QVQLIB	Library of queried table, long name
QVPTBL	QVPTBL	Base table, long name
QVPLIB	QVPLIB	Library of base table, long name
QVBNDY	QVBNDY	I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> • I - I/O bound • C - CPU bound
QVRCNT	QVRCNT	Unique refresh counter
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (Y/N)
QVPARD	QVPARD	Parallel degree requested
QVPARU	QVPARU	Parallel degree used
QVPARRC	QVPARRC	Reason parallel processing was limited
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QVSKIPS	QQC11	Skip sequential table scan (Y/N)
QVTBLSZ	QQI3	Size of table being queried
QVTSFLDS	QVC3001	Columns used for dataspace selection
QVDVFLD	QQC14	Derived column selection (Y/N)
QVDVFLDS	QVC3002	Columns used for derived column selection
QVRDTRG	QQC18	Read Trigger (Y/N)
QVCard	QVP157	User-defined table function Cardinality
QVUTSP	QVC1281	User-defined table function specific name
QVULSP	QVC1282	User-defined table function specific schema

Database monitor logical table 3001 - Summary Row for Index Used

```

|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|
| A*
| A* DB Monitor logical table 3001 - Summary Row for Index Used
| A*
| A          R QQQ3001          PTABLE(*CURLIB/QAQQDBMN)
| A          QQRID
| A          QQTIME
| A          QQJFLD
| A          QQRDBN
| A          QQSYS
| A          QQJOB
| A          QQUSER
| A          QQJNUM
| A          QQTHRD          RENAME(QQI9) +
|                               COLHDG('Thread' +
|                                   'Identifier')
|
| A          QQCNT
| A          QQUDEF
| A          QQQDTN
| A          QQQDTL
| A          QQMATN
| A          QQMATL
| A          QQMATLVL          RENAME(QVP15E) +
|                               COLHDG('Materialized' +
|                                   'Union' +
|                                   'Level')
|
| A          QDQDTN          RENAME(QVP15A) +
|                               COLHDG('Decomposed' +
|                                   'Subselect' +
|                                   'Number')
|
| A          QDQDTT          RENAME(QVP15B) +
|                               COLHDG('Number of' +
|                                   'Decomposed' +
|                                   'Subselects')
|
| A          QDQDTR          RENAME(QVP15C) +
|                               COLHDG('Decomposed' +
|                                   'Reason' +
|                                   'Code')
|
| A          QDQDTS          RENAME(QVP15D) +
|                               COLHDG('Starting' +
|                                   'Decomposed' +
|                                   'Subselect')
|
| A          QQTLN
| A          QQTFN
| A          QQTMN
| A          QQPTLN
| A          QQPTFN
| A          QQPTMN
| A          QQILNM
| A          QQIFNM
| A          QQIMNM
| A          QQTOTR
| A          QQREST
| A          QQFKEY
| A          QQKSEL
| A          QQAJN
| A          QQEPT
| A          QQJNP
| A          QQJNDS          RENAME(QQI1) +
|                               COLHDG('Data Space' +
|                                   'Number')
|
| A          QQJNMT          RENAME(QQC21) +
|                               COLHDG('Join' 'Method')
|
| A          QQJNTY          RENAME(QQC22) +
|                               COLHDG('Join' 'Type')

```

	A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
	A	QQIDXP	RENAME(QQI2) + COLHDG('Advised' + 'Primary' + 'Keys')
	A	QQKP	
	A	QQKPN	RENAME(QQI3) + COLHDG('Number of' 'Key' + 'Positioning' + 'Columns')
	A	QQKS	
	A	QQDSS	
	A	QQIDXA	
	A	QQRCOD	
	A	QQIDXD	
	A	QQCST	RENAME(QQC11) + COLHDG('Index' + 'Is a' + 'Constraint')
	A	QQCSTN	RENAME(QQ1000) + COLHDG('Constraint' + 'Name')
	A*		
	A	QVQTBL	
	A	QVQLIB	
	A	QVPTBL	
	A	QVPLIB	
	A	QVINAM	
	A	QVILIB	
	A	QVBNDY	
	A	QVRCNT	
	A	QVJFANO	
	A	QVFILES	
	A	QVPARPF	
	A	QVPARPL	
	A	QVPARD	
	A	QVPARU	
	A	QVPARRC	
	A	QVCTIM	
	A	QVKOA	RENAME(QVC14) + COLHDG('Index' + 'Only' + 'Access')
	A	QVIDXM	RENAME(QQC12) + COLHDG('Index' + 'fits in' + 'Memory')
	A	QVIDXTY	RENAME(QQC15) + COLHDG('Index' + 'Type')
	A	QVIDXUS	RENAME(QVC12) + COLHDG('Index' + 'Usage')
	A	QVIDXN	RENAME(QQI4) + COLHDG('Number' + 'Index' + 'Entries')
	A	QVIDXUQ	RENAME(QQI5) + COLHDG('Number' + 'Unique' + 'Values')
	A	QVIDXPO	RENAME(QQI6) + COLHDG('Percent' + 'Overflow')
	A	QVIDXVZ	RENAME(QQI7) + COLHDG('Vector' +

A	QVIDXSZ	RENAME(QQI8) + COLHDG('Index' + 'Size')
A	QVIDXPZ	RENAME(QQIA) + COLHDG('Index' + 'Page' + 'Size')
A	QQPSIZ	RENAME(QVP154) + COLHDG('Pool' + 'Size')
A	QQPID	RENAME(QVP155) + COLHDG('Pool' + 'ID')
A	QVTBLSZ	RENAME(QVP156) + COLHDG('Base' + 'Table' + 'Size')
A	QVSKIPS	RENAME(QQC16) + COLHDG('Skip' + 'Sequential')
A	QVIDXTR	RENAME(QVC13) + COLHDG('Tertiary' + 'Indexes' + 'Exist')
A	QVTSFLDS	RENAME(QVC3001) + COLHDG('Columns for' + 'Data Space' + 'Selection')
A	QVDVFLD	RENAME(QVC12 + COLHDG('Derived' + 'Column' + 'Selection')
A	QVDVFLDS	RENAME(QVC3002) + COLHDG('Columns for' + 'Derived' + 'Selection')
A	QVSKEYP	RENAME(QVC3003) + COLHDG('Key' + 'Positioning' + 'Columns')
A	QVSKEYS	RENAME(QVC3004) + COLHDG('Key' + 'Selection' + 'Columns')
A	QVJKEYS	RENAME(QVC3005) + COLHDG('Join' + 'Selection' + 'Columns')
A	QVOKEYS	RENAME(QVC3006) + COLHDG('Ordering' + 'Columns')
A	QVGKEYS	RENAME(QVC3007) + COLHDG('Grouping' + 'Columns')
A	QVRDRG	RENAME(QQC18) + COLHDG('Read' + 'Trigger')
A	QVCard	RENAME(QVP157) + COLHDG('Cardinality')
A	QVUTSP	RENAME(QVC1281) + COLHDG('Specific' + 'Name')
A	QVULSP	RENAME(QVC1282) + COLHDG('Specific' +

```

|                                     'Schema')
|      A      K QQJFLD
|      A      S QQRID                CMP(EQ 3001)

```

| *Table 15. QQQ3001 - Summary Row for Index Used*

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYSYS	QSYSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTR	QVP15B	Total number of decomposed subselects
QDQDTS	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQTTLN	QQTTLN	Library of table queried
QQTFFN	QQTFFN	Name of table queried
QQTMMN	QQTMMN	Member name of table queried
QQPTLN	QQPTLN	Library name of base table
QQPTFN	QQPTFN	Name of base table for table queried
QQPTMMN	QQPTMMN	Member name of base table
QQILNM	QQILNM	Library name of index used for access
QQIFNM	QQIFNM	Name of index used for access
QQIMNM	QQIMNM	Member name of index used for access
QQTOTR	QQTOTR	Total rows in base table
QQRST	QQRST	Estimated number of rows selected
QQFKEY	QQFKEY	Columns selected thru index scan-key positioning
QQKSEL	QQKSEL	Columns selected thru index scan-key selection
QQAJN	QQAJN	Estimated number of joined rows
QQEPT	QQEPT	Estimated processing time, in seconds

Table 15. QQQ3001 - Summary Row for Index Used (continued)

Logical Column Name	Physical Column Name	Description
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI1	dataspace number
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> NL - Nested loop MF - Nested loop with selection HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> IN - Inner join PO - Left partial outer join EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product
QQIDXP	QQI2	Number of advised key columns that use index scan-key positioning
QQKP	QQKP	Index scan-key positioning <ul style="list-style-type: none"> Y - Yes N - No
QQKPN	QQI3	Number of columns that use index scan-key positioning for the index used
QQKS	QQKS	Index scan-key selection <ul style="list-style-type: none"> Y - Yes N - No
QQDSS	QQDSS	dataspace selection <ul style="list-style-type: none"> Y - Yes N - No
QQIDXA	QQIDXA	Index advised <ul style="list-style-type: none"> Y - Yes N - No
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> I1 - Row selection I2 - Ordering/Grouping I3 - Row selection and Ordering/Grouping I4 - Nested loop join I5 - Row selection using bitmap processing
QQIDXD	QQIDXD	Columns for index advised
QQCST	QQC11	Index is a constraint (Y/N)
QQCSTN	QQ1000	Constraint name
QVQTBL	QVQTBL	Queried table, long name

| Table 15. QQQ3001 - Summary Row for Index Used (continued)

Logical Column Name	Physical Column Name	Description
QVQLIB	QVQLIB	Library of queried table, long name
QVPTBL	QVPTBL	Base table, long name
QVPLIB	QVPLIB	Library of base table, long name
QVINAM	QVINAM	Name of index (or constraint) used, long name
QVILIB	QVILIB	Library of index used, long name
QVBNDY	QVBNDY	I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> • I - I/O bound • C - CPU bound
QVRCNT	QVRCNT	Unique refresh counter
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (Y/N)
QVPARD	QVPARD	Parallel degree requested
QVPARU	QVPARU	Parallel degree used
QVPARRC	QVPARRC	Reason parallel processing was limited
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QVKOA	QVC14	Index only access (Y/N)
QVIDXM	QQC12	Index fits in memory (Y/N)
QVIDXTY	QQC15	Type of Index. Possible values are: <ul style="list-style-type: none"> • B - Binary Radix Index • C - Constraint (Binary Radix) • E - Encoded Vector Index (EVI) • X - Query created temporary index
QVIDXUS	QVC12	Index Usage. Possible values are: <ul style="list-style-type: none"> • P - Primary Index • T - Tertiary (AND/OR) Index
QVIDXN	QQI4	Number of index entries
QVIDXUQ	QQI5	Number of unique key values
QVIDXPO	QQI6	Percent overflow
QVIDXVZ	QQI7	Vector size
QVIDXSZ	QQI8	Index size
QVIDXPZ	QQIA	Index page size
QQPSIZ	QVP154	Pool size
QQPID	QVP155	Pool id

Table 15. QQQ3001 - Summary Row for Index Used (continued)

Logical Column Name	Physical Column Name	Description
QVTBLSZ	QVP156	Table size
QVSKIPS	QQC16	Skip sequential table scan (Y/N)
QVIDXTR	QVC13	Tertiary indexes exist (Y/N)
QVTSFLDS	QVC3001	Columns used for dataspace selection
QVDVFLD	QQC14	Derived column selection (Y/N)
QVDVFLDS	QVC3002	Columns used for derived column selection
QVSKEYP	QVC3003	Columns used for index scan-key positioning
QVSKEYS	QVC3004	Columns used for index scan-key selection
QVJKEYS	QVC3005	Columns used for Join selection
QVOKEYS	QVC3006	Columns used for Ordering
QVGKEYS	QVC3007	Columns used for Grouping
QVRDTRG	QQC18	Read Trigger (Y/N)
QVCard	QVP157	User-defined table function Cardinality
QVUTSP	QVC1281	User-defined table function specific name
QVULSP	QVC1282	User-defined table function specific schema

Database monitor logical table 3002 - Summary Row for Index Created

```

| |...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...8
| |
| | A*
| | A* DB Monitor logical table 3002 - Summary Row for Index Created
| | A*
| | A          R QQQ3002          PTABLE(*CURLIB/QAQQDBMN)
| | A          QQRID
| | A          QQTIME
| | A          QQJFLD
| | A          QQRDBN
| | A          QSYS
| | A          QQJOB
| | A          QQUSER
| | A          QQJNUM
| | A          QQTHRD          RENAME(QQI9) +
| |                               COLHDG('Thread' +
| |                               'Identifier')
| |
| | A          QQCNT
| | A          QQUDEF
| | A          QQQDTN
| | A          QQQDTL
| | A          QQMATN
| | A          QQMATL
| | A          QQMATLVL          RENAME(QVP15E) +
| |                               COLHDG('Materialized' +
| |                               'Union' +
| |                               'Level')
| |
| | A          QDQDTN          RENAME(QVP15A) +
| |                               COLHDG('Decomposed' +
| |                               'Subselect' +
| |                               'Number')
| |
| | A          QDQDTT          RENAME(QVP15B) +
| |                               COLHDG('Number of' +
| |                               'Decomposed' +
| |                               'Subselects')
| |
| | A          QDQDTR          RENAME(QVP15C) +

```

			COLHDG('Decomposed' +
			'Reason' +
			'Code')
	A	QQDQTS	RENAME(QVP15D) +
			COLHDG('Starting' +
			'Decomposed' +
			'Subselect')
	A	QQTLN	
	A	QQTFN	
	A	QQTMN	
	A	QQPTLN	
	A	QQPTFN	
	A	QQPTMN	
	A	QQILNM	
	A	QQIFNM	
	A	QQIMNM	
	A	QQNTNM	
	A	QQNLNM	
	A	QQSTIM	
	A	QQETIM	
	A	QQTOTR	
	A	QQRIDX	
	A	QQREST	
	A	QQFKEY	
	A	QQKSEL	
	A	QQAJN	
	A	QQEPT	
	A	QQJNP	
	A	QQJNDS	RENAME(QQI1) +
			COLHDG('Data Space' +
			'Number')
	A	QQJNMT	RENAME(QQC21) +
			COLHDG('Join' 'Method')
	A	QQJNTY	RENAME(QQC22) +
			COLHDG('Join' 'Type')
	A	QQJNOP	RENAME(QQC23) +
			COLHDG('Join' 'Operator')
	A	QQIDXK	RENAME(QQI2) +
			COLHDG('Advised' +
			'Primary' 'Keys')
	A	QQKP	
	A	QQKPN	RENAME(QQI3) +
			COLHDG('Number' 'Key' +
			'Positioning' +
			'Columns')
	A	QQKS	
	A	QQDSS	
	A	QQIDXA	
	A	QQRCOD	
	A	QQIDXD	
	A	QQCRTK	RENAME(QQ1000) +
			COLHDG('Key Columns' +
			'of Index' +
			'Created')
	A	QVQTBL	
	A	QVQLIB	
	A	QVPTBL	
	A	QVPLIB	
	A	QVINAM	
	A	QVILIB	
	A	QVBNDY	
	A	QVRCNT	
	A	QVJFANO	
	A	QVFILES	
	A	QVPARPF	
	A	QVPARPL	
	A	QVPARD	

A	QVPARU	
A	QVPARRC	
A	QVCTIM	
A	QVTIXN	RENAME(QQC101) + COLHDG('Name of' + 'Index' + 'Created')
A	QVTIXL	RENAME(QQC102) + COLHDG('Library of' + 'Index' + 'Created')
A	QVTIXPZ	RENAME(QQI4) + COLHDG('Page Size' + 'of Index' + 'Created')
A	QVTIXRZ	RENAME(QQI5) + COLHDG('Row Size' + 'of Index' + 'Created')
A	QVTIXACF	RENAME(QQC14) + COLHDG('ACS' + 'Table' + 'Used')
A	QVTIXACS	RENAME(QQC103) + COLHDG('Alternate' + 'Collating' + 'Sequence' + 'Table')
A	QVTIXACL	RENAME(QQC104) + COLHDG('Alternate' + 'Collating' + 'Sequence' + 'Library')
A	QVTIXRU	RENAME(QVC13) + COLHDG('Index + 'Created is' + 'Reusable')
A	QVTIXSP	RENAME(QVC14) + COLHDG('Index + 'Created is' + 'Sparse')
A	QVTIXTY	RENAME(QVC1F) + COLHDG('Type of' + 'Index' + 'Created')
A	QVTIXUQ	RENAME(QVP15F) + COLHDG('Number of' + 'Unique Values' + 'Index Created')
A	QVTIXPO	RENAME(QVC15) + COLHDG('Permanent' + 'Index' + 'Created')
A	QVTIXFX	RENAME(QVC16) + COLHDG('Index' + 'From' + 'Index')
A	QVTIXPD	RENAME(QVP151) + COLHDG('Parallel' + 'Degree' + 'Requested')
A	QVTIXPU	RENAME(QVP152) + COLHDG('Parallel' + 'Degree' + 'Used')
A	QVTIXPRC	RENAME(QVP153) + COLHDG('Parallel' +

A	QVKOA	'Degree ' + 'Limited') RENAME(QVC17) + COLHDG('Index' + 'Only' + 'Access')
A	QVIDXM	RENAME(QVC18) + COLHDG('Index' + 'fits in' + 'Memory')
A	QVIDXTY	RENAME(QVC1B) + COLHDG('Index' + 'Type')
A	QVIDXN	RENAME(QQI6) + COLHDG('Entries in' + 'Index' + 'Used')
A	QVIDXUQ	RENAME(QQI7) + COLHDG('Number' + 'Unique' + 'Values')
A	QVIDXPO	RENAME(QVP158) + COLHDG('Percent' + 'Overflow')
A	QVIDXVZ	RENAME(QVP159) + COLHDG('Vector' + 'Size')
A	QVIDXSZ	RENAME(QQI8) + COLHDG('Size of' + 'Index' + 'Used')
A	QVIDXPZ	RENAME(QVP156) + COLHDG('Page Size' + 'Index' + 'Used')
A	QQPSIZ	RENAME(QVP154) + COLHDG('Pool' + 'Size')
A	QQPID	RENAME(QVP155) + COLHDG('Pool' + 'ID')
A	QVTBLSZ	RENAME(QVP157) + COLHDG('Table' + 'Size')
A	QVSKIPS	RENAME(QVC1C) + COLHDG('Skip' + 'Sequential')
A	QVTSFLDS	RENAME(QVC3001) + COLHDG('Columns for' + 'Data Space' + 'Selection')
A	QVDVFLD	RENAME(QVC1E + COLHDG('Derived' + 'Column' + 'Selection')
A	QVDVFLDS	RENAME(QVC3002) + COLHDG('Columns for' + 'Derived' + 'Selection')
A	QVSKEYP	RENAME(QVC3003) + COLHDG('Columns Used' + 'for Key' + 'Positioning')
A	QVSKEYS	RENAME(QVC3004) + COLHDG('Columns Used' + 'for Key' + 'Selection')

```

|      A          QVRDTRG          RENAME(QQC18) +
|                                     COLHDG('Read' +
|                                     'Trigger')
|      A          K QQJFLD
|      A          S QQRID          CMP(EQ 3002)

```

Table 16. QQQ3002 - Summary row for Index Created

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHR	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATLVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQTLN	QQTLN	Library of table queried
QQTFN	QQTFN	Name of table queried
QQTMN	QQTMN	Member name of table queried
QQPTLN	QQPTLN	Library name of base table
QQPTFN	QQPTFN	Name of base table for table queried
QQPTMN	QQPTMN	Member name of base table
QQILNM	QQILNM	Library name of index used for access
QQIFNM	QQIFNM	Name of index used for access
QQIMNM	QQIMNM	Member name of index used for access
QQNTNM	QQNTNM	NLSS library
QQNLNM	QQNLNM	NLSS table
QQSTIM	QQSTIM	Start timestamp
QQETIM	QQETIM	End timestamp
QQTOTR	QQTOTR	Total rows in table

| Table 16. QQQ3002 - Summary row for Index Created (continued)

Logical Column Name	Physical Column Name	Description
QQRIDX	QQRIDX	Number of entries in index created
QQREST	QQREST	Estimated number of rows selected
QQFKEY	QQFKEY	Keys selected thru index scan-key positioning
QQKSEL	QQKSEL	Keys selected thru index scan-key selection
QQAJN	QQAJN	Estimated number of joined rows
QQEPT	QQEPT	Estimated processing time, in seconds
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI1	dataspace number
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
QQIDXK	QQI2	Number of advised key columns that use index scan-key positioning
QQKP	QQKP	Index scan-key positioning <ul style="list-style-type: none"> • Y - Yes • N - No
QQKPN	QQI3	Number of columns that use index scan-key positioning for the index used
QQKS	QQKS	Index scan-key selection <ul style="list-style-type: none"> • Y - Yes • N - No
QQDSS	QQDSS	dataspace selection <ul style="list-style-type: none"> • Y - Yes • N - No
QQIDXA	QQIDXA	Index advised <ul style="list-style-type: none"> • Y - Yes • N - No

Table 16. QQQ3002 - Summary row for Index Created (continued)

Logical Column Name	Physical Column Name	Description
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> • I1 - Row selection • I2 - Ordering/Grouping • I3 - Row selection and Ordering/Grouping • I4 - Nested loop join
QQIDXD	QQIDXD	Key columns for index advised
QQCRTK	QQ1000	Key columns for index created
QVQTBL	QVQTBL	Queried table, long name
QVQLIB	QVQLIB	Library of queried table, long name
QVPTBL	QVPTBL	Base table, long name
QVPLIB	QVPLIB	Library of base table, long name
QVINAM	QVINAM	Name of index (or constraint) used, long name
QVILIB	QVILIB	Library of index used, long name
QVBNDY	QVBNDY	I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> • I - I/O bound • C - CPU bound
QVRCNT	QVRCNT	Unique refresh counter
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (index used)
QVPARD	QVPARD	Parallel degree requested (index used)
QVPARU	QVPARU	Parallel degree used (index used)
QVPARRC	QVPARRC	Reason parallel processing was limited (index used)
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QVTIXN	QQC101	Name of index created
QVTIXL	QQC102	Library of index created
QVTIXPZ	QQI4	Page size of index created
QVTIXRZ	QQI5	Row size of index created
QVTIXACS	QQC103	Alternate Collating Sequence table of index created.
QVTIXACL	QQC104	Alternate Collating Sequence library of index created.
QVTIXRU	QVC13	Index created is reusable (Y/N)
QVTIXSP	QVC14	Index created is sparse index (Y/N)

| Table 16. QQQ3002 - Summary row for Index Created (continued)

Logical Column Name	Physical Column Name	Description
QVTIXTY	QVC1F	Type of index created. Possible values: • B - Binary Radix Index • E - Encoded Vector Index (EVI)
QVTIXUQ	QVP15A	Number of unique values of index created if index created is an EVI index.
QVTIXPO	QVC15	Permanent index created (Y/N)
QVTIXFX	QVC16	Index from index (Y/N)
QVTIXPD	QVP151	Parallel degree requested (index created)
QVTIXPU	QVP152	Parallel degree used (index created)
QVTIXPRC	QVP153	Reason parallel processing was limited (index created)
QVKOA	QVC17	Index only access (Y/N)
QVIDXM	QVC18	Index fits in memory (Y/N)
QVIDXTY	QVC1B	Type of Index. Possible values are: • B - Binary Radix Index • C - Constraint (Binary Radix) • E - Encoded Vector Index (EVI) • T - Tertiary (AND/OR) Index
QVIDXN	QQI6	Number of index entries, index used
QVIDXUQ	QQI7	Number of unique key values, index used
QVIDXPO	QVP158	Percent overflow, index used
QVIDXVZ	QVP159	Vector size, index used
QVIDXSZ	QQI8	Size of index used.
QVIDXPZ	QVP156	Index page size
QQPSIZ	QVP154	Pool size
QQPID	QVP155	Pool id
QVTBLSZ	QVP157	Table size
QVSKIPS	QVC1C	Skip sequential table scan (Y/N)
QVTSFLDS	QVC3001	Columns used for dataspace selection
QVDVFLD	QVC1E	Derived column selection (Y/N)
QVDVFLDS	QVC3002	Columns used for derived column selection
QVSKEYP	QVC3003	Columns used for index scan-key positioning
QVSKEYS	QVC3004	Columns used for index scan-key selection
QVRDTRG	QQC18	Read Trigger (Y/N)

Database monitor logical table 3003 - Summary Row for Query Sort

```
| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|   A*
|   A* DB Monitor logical table 3003 - Summary Row for Query Sort
|   A*
|   A           R QQQ3003           TABLE(*CURLIB/QAQQDBMN)
|   A           QQRID
|   A           QQTIME
|   A           QQJFLD
```

	A	QQRDBN	
	A	QSYS	
	A	QQJOB	
	A	QQUSER	
	A	QQJNUM	
	A	QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')
	A	QQUCNT	
	A	QQUDEF	
	A	QQQDTN	
	A	QQQDTL	
	A	QQMATN	
	A	QQMATL	
	A	QQMATULVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')
	A	QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')
	A	QQQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')
	A	QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
	A	QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
	A	QQSTIM	
	A	QQETIM	
	A	QQRSS	
	A	QQSSIZ	RENAME(QQI1) + COLHDG('Size of' + 'Sort' + 'Space')
	A	QQPSIZ	RENAME(QQI2) + COLHDG('Pool' + 'Size')
	A	QQPID	RENAME(QQI3) + COLHDG('Pool' + 'ID')
	A	QQIBUF	RENAME(QQI4) + COLHDG('Internal' + 'Buffer' + 'Length')
	A	QQEBUF	RENAME(QQI5) + COLHDG('External' + 'Buffer' + 'Length')
	A	QQRCD	
	A	QQRCSUB	RENAME(QQI7)
	A	QVBNDY	
	A	QVRCNT	
	A	QVPARPF	
	A	QVPARPL	
	A	QVPARD	
	A	QVPARU	
	A	QVPARRC	
	A	QQEPT	
	A	QVCTIM	
	A	QQAJN	

	A	QQJNP	
	A	QQJNDS	RENAME(QQI6) + COLHDG('Data Space' + 'Number')
	A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
	A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
	A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
	A	QVJFANO	
	A	QVFILES	
	A	K QQJFLD	
	A	S QQRID	CMP(EQ 3003)

Table 17. QQQ3003 - Summary Row for Query Sort

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QQSYS	QQSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQSTIM	QQSTIM	Start timestamp
QQETIM	QQETIM	End timestamp
QQRSS	QQRSS	Number of rows selected or sorted
QQSSIZ	QQI1	Size of sort space
QQPSIZ	QQI2	Pool size
QQPID	QQI3	Pool id
QQIBUF	QQI4	Internal sort buffer length
QQEBUF	QQI5	External sort buffer length

Table 17. QQQ3003 - Summary Row for Query Sort (continued)

Logical Column Name	Physical Column Name	Description
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> F1 - Query contains grouping columns (GROUP BY) from more than one table, or contains grouping columns from a secondary table of a join query that cannot be reordered. F2 - Query contains ordering columns (ORDER BY) from more than one table, or contains ordering columns from a secondary table of a join query that cannot be reordered. F3 - The grouping and ordering columns are not compatible. F4 - DISTINCT was specified for the query. F5 - UNION was specified for the query. F6 - Query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 key columns specified for ordering. F7 - Query optimizer chose to use a sort rather than an index to order the results of the query. F8 - Perform specified row selection to minimize I/O wait time. FC - The query contains grouping fields and there is a read trigger on at least one of the physical files in the query.
QQRCSUB	QQI7	Reason subcode for Union: <ul style="list-style-type: none"> 51 - Query contains UNION and ORDER BY 52 - Query contains UNION ALL
QVBNDY	QVBNDY	I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> I - I/O bound C - CPU bound
QVRCNT	QVRCNT	Unique refresh counter
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (index used)
QVPARD	QVPARD	Parallel degree requested (index used)
QVPARU	QVPARU	Parallel degree used (index used)
QVPARRC	QVPARRC	Reason parallel processing was limited (index used)
QQEPT	QQEPT	Estimated processing time, in seconds
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QQAJN	QQAJN	Estimated number of joined rows
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> NL - Nested loop MF - Nested loop with selection HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> IN - Inner join PO - Left partial outer join EX - Exception join

Table 17. QQQ3003 - Summary Row for Query Sort (continued)

Logical Column Name	Physical Column Name	Description
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined

Database monitor logical table 3004 - Summary Row for Temp Table

```

|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|
| A*
| A* DB Monitor logical table 3004 - Summary Row for Temp Table
| A*
| A      R QQQ3004          PTABLE(*CURLIB/QAQQDBMN)
| A      QQRID
| A      QQTIME
| A      QQJFLD
| A      QQRDBN
| A      QQSYS
| A      QQJOB
| A      QQUSER
| A      QQJNUM
| A      QQTHRD          RENAME(QQI9) +
|                          COLHDG('Thread' +
|                                'Identifier')
|
| A      QQUCNT
| A      QQUDEF
| A      QQQDTN
| A      QQQDTL
| A      QQMATN
| A      QQMATL
| A      QQMATLVL       RENAME(QVP15E) +
|                          COLHDG('Materialized' +
|                                'Union' +
|                                'Level')
|
| A      QDQDTN         RENAME(QVP15A) +
|                          COLHDG('Decomposed' +
|                                'Subselect' +
|                                'Number')
|
| A      QDQDTR         RENAME(QVP15B) +
|                          COLHDG('Number of' +
|                                'Decomposed' +
|                                'Subselects')
|
| A      QDQDTR         RENAME(QVP15C) +
|                          COLHDG('Decomposed' +
|                                'Reason' +

```

			'Code')
	A	QDQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
	A	QQTLN	
	A	QQTFN	
	A	QQTMN	
	A	QQPTLN	
	A	QQPTFN	
	A	QQPTMN	
	A	QQSTIM	
	A	QQETIM	
	A	QQDFVL	RENAME(QQC11) + COLHDG('Default' + 'Values')
	A	QQTMPR	
	A	QQRCD	
	A	QVQTBL	
	A	QVQLIB	
	A	QVPTBL	
	A	QVPLIB	
	A	QVTTBLN	RENAME(QQC101) + COLHDG('Temporary' + 'Table' + 'Name')
	A	QVTTBLL	RENAME(QQC102) + COLHDG('Temporary' + 'Table' + 'Library')
	A	QVBNDY	
	A	QVRCNT	
	A	QVPARPF	
	A	QVPARPL	
	A	QVPARD	
	A	QVPARU	
	A	QVPARRC	
	A	QQEPT	
	A	QVCTIM	
	A	QQAJN	
	A	QQJNP	
	A	QQJNDS	RENAME(QQI6) + COLHDG('Data Space' + 'Number')
	A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
	A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
	A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
	A	QVJFANO	
	A	QVFILES	
	A	QVTTRSZ	RENAME(QQI2) + COLHDG('Row Size' + 'Temporary' + 'Table')
	A	QVTTISZ	RENAME(QQI3) + COLHDG('Table Size' + 'Temporary' + 'Table')
	A	QVTTTRST	RENAME(QQC12) + COLHDG('Temporary' + 'Result')
	A	QVTTDST	RENAME(QQC13) + COLHDG('Distributed' + 'Table')
	A	QVTTNOD	RENAME(QVC3001) +

A	QMATDLVL	COLHDG('Data' + 'Nodes') RENAME(QQI7) + COLHDG('Materialized' + 'Subquery') + 'Level')
A	QMATDULVL	RENAME(QQI8) + COLHDG('Materialized' + 'Union') + 'Level')
A	QQUNVW	RENAME(QQC14) + COLHDG('Union' + 'In A View')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3004)

Table 18. QQQ3004 - Summary Row for Temp Table

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYSYS	QSYSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQTLN	QQTLN	Library of table queried
QQTFN	QQTFN	Name of table queried
QQTMN	QQTMN	Member name of table queried
QQPTLN	QQPTLN	Library name of base table
QQPTFN	QQPTFN	Name of base table for table queried
QQPTMN	QQPTMN	Member name of base table
QQSTIM	QQSTIM	Start timestamp
QQETIM	QQETIM	End timestamp

Table 18. QQQ3004 - Summary Row for Temp Table (continued)

Logical Column Name	Physical Column Name	Description
QQDFVL	QQC11	Default values may be present in temporary <ul style="list-style-type: none"> • Y - Yes • N - No
QQTMPR	QQTMPR	Number of rows in the temporary
QQRCOD	QQRCOD	Reason code. Possible values are: <ul style="list-style-type: none"> • F1 - Query contains grouping columns (GROUP BY) from more than one table, or contains grouping columns from a secondary table of a join query that cannot be reordered. • F2 - Query contains ordering columns (ORDER BY) from more than one table, or contains ordering columns from a secondary table of a join query that cannot be reordered. • F3 - The grouping and ordering columns are not compatible. • F4 - DISTINCT was specified for the query. • F5 - UNION was specified for the query. • F6 - Query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 key columns specified for ordering. • F7 - Query optimizer chose to use a sort sort rather than an index to order the results of the query. • F8 - Perform specified row selection to minimize I/O wait time. • F9 - The query optimizer chose to use a hashing algorithm rather than an index to perform the grouping. • FA - The query contains a join condition that requires a temporary table • FB - The query optimizer creates a run-time temporary file in order to implement certain correlated group by queries. • FC - The query contains grouping fields and there is a read trigger on at least one of the physical files in the query. • FD - The query optimizer creates a runtime temporary file for a static-cursor request. • H1 - Table is a join logical file and its join type does not match the join type specified in the query. • H2 - Format specified for the logical table references more than one base table. • H3 - Table is a complex SQL view requiring a temporary table to contain the the results of the SQL view. • H4 - For an update-capable query, a subselect references a column in this table which matches one of the columns being updated. • H5 - For an update-capable query, a subselect references an SQL view which is based on the table being updated. • H6 - For a delete-capable query, a subselect references either the table from which rows are to be deleted, an SQL view, or an index based on the table from which rows are to be deleted • H7 - A user-defined table function was materialized.
QVQTBL	QVQTBL	Queried table, long name
QVQLIB	QVQLIB	Library of queried table, long name
QVPTBL	QVPTBL	Base table, long name

| Table 18. QQQ3004 - Summary Row for Temp Table (continued)

Logical Column Name	Physical Column Name	Description
QVPLIB	QVPLIB	Library of base table, long name
QVTTBLN	QQC101	Temporary table name
QVTTBLL	QQC102	Temporary table library
QVBNDY	QVBNDY	I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> • I - I/O bound • C - CPU bound
QVRCNT	QVRCNT	Unique refresh counter
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (Y/N)
QVPARD	QVPARD	Parallel degree requested
QVPARU	QVPARU	Parallel degree used
QVPARRC	QVPARRC	Reason parallel processing was limited
QQEPT	QQEPT	Estimated processing time, in seconds
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QQAJN	QQAJN	Estimated number of joined rows
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
QVTRSZ	QQI2	Row size of temporary table, in bytes

Table 18. QQQ3004 - Summary Row for Temp Table (continued)

Logical Column Name	Physical Column Name	Description
QVTTISIZ	QQI3	Size of temporary table, in bytes
QVTRRST	QQC12	Temporary result table that contains the results of the query. (Y/N)
QVTTDST	QQC13	Distributed Table (Y/N)
QVTTNOD	QVC3001	Data nodes of temporary table
QMATDLVL	QQI7	Materialized subquery QDT level
QMATDULVL	QQI8	Materialized Union QDT level
QQUNVW	QQC14	Union in a view (Y/N)

Database monitor logical table 3005 - Summary Row for Table Locked

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
| A*
| A* DB Monitor logical table 3005 - Summary Row for Table Locked
| A*
| A      R QQQ3005          PTABLE(*CURLIB/QAQQDBMN)
| A      QQRID
| A      QQTIME
| A      QQJFLD
| A      QQRDBN
| A      QQSYS
| A      QQJOB
| A      QQUSER
| A      QQJNUM
| A      QQTHRD          RENAME(QQI9) +
|                          COLHDG('Thread' +
|                              'Identifier')
|
| A      QQUCNT
| A      QQUDEF
| A      QQQDTN
| A      QQQDTL
| A      QQMATN
| A      QQMATL
| A      QQMATULVL      RENAME(QVP15E) +
|                          COLHDG('Materialized' +
|                              'Union' +
|                              'Level')
|
| A      QDQDTN          RENAME(QVP15A) +
|                          COLHDG('Decomposed' +
|                              'Subselect' +
|                              'Number')
|
| A      QDQDTR          RENAME(QVP15B) +
|                          COLHDG('Number of' +
|                              'Decomposed' +
|                              'Subselects')
|
| A      QDQDTS          RENAME(QVP15C) +
|                          COLHDG('Decomposed' +
|                              'Reason' +
|                              'Code')
|
| A      QDQDTS          RENAME(QVP15D) +
|                          COLHDG('Starting' +
|                              'Decomposed' +
|                              'Subselect')
|
| A      QQTLN
| A      QQTFN
| A      QQTMN
| A      QQPTLN
| A      QQPTFN
| A      QQPTMN

```

A	QQLCKF	RENAME(QQC11) + COLHDG('Lock' + 'Indicator')
A	QQULCK	RENAME(QQC12) + COLHDG('Unlock' + 'Request')
A	QQRCD	
A	QVQTBL	
A	QVQLIB	
A	QVPTBL	
A	QVPLIB	
A	QQJNP	
A	QQJNDS	RENAME(QQI6) + COLHDG('Data Space' + 'Number')
A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
A	QVJFANO	
A	QVFILES	
A	QVRCNT	
A	K QQJFLD	
A	S QQRID	CMP(EQ 3005)

Table 19. QQQ3005 - Summary Row for Table Locked

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQTLN	QQTLN	Library of table queried

Table 19. QQQ3005 - Summary Row for Table Locked (continued)

Logical Column Name	Physical Column Name	Description
QQTFN	QQTFN	Name of table queried
QQTMN	QQTMN	Member name of table queried
QQPTLN	QQPTLN	Library name of base table
QQPTFN	QQPTFN	Name of base table for table queried
QQPTMN	QQPTMN	Member name of base table
QQLCKF	QQC11	Successful lock indicator <ul style="list-style-type: none"> • Y - Yes • N - No
QQULCK	QQC12	Unlock request <ul style="list-style-type: none"> • Y - Yes • N - No
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> • L1 - UNION with *ALL or *CS with Keep Locks • L2 - DISTINCT with *ALL or *CS with Keep Locks • L3 - No duplicate keys with *ALL or *CS with Keep Locks • L4 - Temporary needed with *ALL or *CS with Keep Locks • L5 - System Table with *ALL or *CS with Keep Locks • L6 - Orderby > 2000 bytes with *ALL or *CS with Keep Locks • L9 - Unknown • LA - User-defined table function with *ALL or *CS with Keep Locks
QVQTBL	QVQTBL	Queried table, long name
QVQLIB	QVQLIB	Library of queried table, long name
QVPTBL	QVPTBL	Base table, long name
QVPLIB	QVPLIB	Library of base table, long name
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product

Table 19. QQQ3005 - Summary Row for Table Locked (continued)

Logical Column Name	Physical Column Name	Description
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVRCNT	QVRCNT	Unique refresh counter

Database monitor logical table 3006 - Summary Row for Access Plan Rebuilt

1	2	3	4	5	6	7	8
A*	A* DB Monitor logical table 3006 - Summary Row for Access Plan Rebuilt						
A*							
A	R	QQQ3006	PTABLE(*CURLIB/QAQQDBMN)				
A		QQRID					
A		QQTIME					
A		QQJFLD					
A		QQRDBN					
A		QSYSYS					
A		QQJOB					
A		QQUSER					
A		QQJNUM					
A		QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')				
A		QQUCNT					
A		QQUDEF					
A		QQQDTN					
A		QQQDTL					
A		QQMATN					
A		QQMATL					
A		QQMATLVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')				
A		QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')				
A		QQQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')				
A		QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')				
A		QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')				
A		QQINLN					
A		QQINFN					
A		QQRCOD					

A	QVSUBRC	RENAME(QQC21) + COLHDG('Subtype' + 'Reason' + 'Code')
A	QVRCNT	
A	QVRPTS	RENAME(QQTIM1) + COLHDG('Timestamp' + 'Last' + 'Rebuild')
A	QRQDOPT	RENAME(QQC11) + COLHDG('Access' + 'Plan' + 'Reoptimized')
A	QRCODES	RENAME(QVC22) + COLHDG('Previous' + 'Reason' + 'Code')
A	QVSUBRCS	RENAME(QVC23) + COLHDG('Previous' + 'Reason' + 'Subcode')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3006)

Table 20. QQQ3006 - Summary Row for Access Plan Rebuilt

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHR	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATLVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect

Table 20. QQQ3006 - Summary Row for Access Plan Rebuilt (continued)

Logical Column Name	Physical Column Name	Description
QQRCOD	QQRCOD	<p>Reason code why access plan was rebuilt</p> <ul style="list-style-type: none"> • A1 - A table or member is not the same object as the one referenced when the access plan was last built. Some reasons they could be different are: <ul style="list-style-type: none"> – Object was deleted and recreated. – Object was saved and restored. – Library list was changed. – Object was renamed. – Object was moved. – Object was overridden to a different object. – This is the first run of this query after the object containing the query has been restored. • A2 - Access plan was built to use a reusable Open Data Path (ODP) and the optimizer chose to use a non-reusable ODP for this call. • A3 - Access plan was built to use a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP for this call. • A4 - The number of rows in the table has changed by more than 10% since the access plan was last built. • A5 - A new index exists over one of the tables in the query • A6 - An index that was used for this access plan no longer exists or is no longer valid. • A7 - OS/400 Query requires the access plan to be rebuilt because of system programming changes. • A8 - The CCSID of the current job is different than the CCSID of the job that last created the access plan. • A9 - The value of one or more of the following is different for the current job than it was for the job that last created this access plan: <ul style="list-style-type: none"> – date format – date separator – time format – time separator. • AA - The sort sequence table specified is different than the sort sequence table that was used when this access plan was created. • AB - Storage pool changed or DEGREE parameter of CHGQRYA command changed. • AC - The system feature DB2 multisystem has been installed or removed. • AD - The value of the degree query attribute has changed. • AE - A view is either being opened by a high level language or a view is being materialized. • AF - A user-defined type or user-defined function is not the same object as the one referred to in the access plan, or, the SQL Path is not the same as when the access plan was built. • B0 - The options specified have changed as a result of the query options file. • B1 - The access plan was generated with a commitment control level that is different in the current job. • B2 - The access plan was generated with a static cursor answer set size that is different than the previous access plan.
QVSUBRC	QQC21	<p>If the access plan rebuild reason code was A7 this two-byte hex value identifies which specific reason for A7 forced a rebuild.</p>

Table 20. QQQ3006 - Summary Row for Access Plan Rebuilt (continued)

Logical Column Name	Physical Column Name	Description
QVRCNT	QVRCNT	Unique refresh counter
QVRPTS	QQTIM1	Timestamp of last access plan rebuild
QRQDOPT	QQC11	Required optimization for this plan. <ul style="list-style-type: none"> • Y - Yes, plan was really optimized. • N - No, the plan was not reoptimized because of the QAQQINI option for the REOPTIMIZE_ACCESS_PLAN parameter value
QRCODES	QVC22	Previous reason code
QVSUBRCS	QVC23	Previous reason subcode

Database monitor logical table 3007 - Summary Row for Optimizer Timed Out

1	2	3	4	5	6	7	8	
A*	A* DB Monitor logical table 3007 - Summary Row for Optimizer Timed Out							
A*	A*							
A	R	QQQ3007	PTABLE(*CURLIB/QAQQDBMN)					
A		QQRID						
A		QQTIME						
A		QQJFLD						
A		QQRDBN						
A		QSYSYS						
A		QQJOB						
A		QQUSER						
A		QQJNUM						
A		QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')					
A		QQUCNT						
A		QQUDEF						
A		QQQDTN						
A		QQQDTL						
A		QQMATN						
A		QQMATL						
A		QQMATULVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')					
A		QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')					
A		QQQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')					
A		QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')					
A		QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')					
A		QQTLN						
A		QQTFN						
A		QQTMN						

	A	QQPTLN	
	A	QQPTFN	
	A	QQPTMN	
	A	QQIDXN	RENAME(QQ1000) + COLHDG('Index' + 'Names')
	A	QQTOUT	RENAME(QQC11) + COLHDG('Optimizer' + 'Timed Out')
	A	QQISRN	RENAME(QQC301) + COLHDG('Index' + 'Reason' +
	A	QVQTBL	
	A	QVQLIB	
	A	QVPTBL	
	A	QVPLIB	
	A	QQJNP	
	A	QQJNDS	RENAME(QQI6) + COLHDG('Data Space' + 'Number')
	A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
	A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
	A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
	A	QVJFANO	
	A	QVFILES	
	A	QVRCNT	
	A	K QQJFLD	
	A	S QQRID	CMP(EQ 3007)

Table 21. QQQ3007 - Summary Row for Optimizer Timed Out

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYSYS	QSYSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code

Table 21. QQQ3007 - Summary Row for Optimizer Timed Out (continued)

Logical Column Name	Physical Column Name	Description
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQTLN	QQTLN	Library of table queried
QQTFN	QQTFN	Name of table queried
QQTMN	QQTMN	Member name of table queried
QQPTLN	QQPTLN	Library name of base table
QQPTFN	QQPTFN	Name of base table for table queried
QQPTMN	QQPTMN	Member name of base table
QQIDXN	QQ1000	Index names
QQTOUT	QQC11	Optimizer timed out <ul style="list-style-type: none"> • Y - Yes • N - No
QQISRN	QQC301	List of unique reason codes used by the indexes that timed out (each index has a corresponding reason code associated with it)
QVQTBL	QVQTBL	Queried table, long name
QVQLIB	QVQLIB	Library of queried table, long name
QVPTBL	QVPTBL	Base table, long name
QVPLIB	QVPLIB	Library of base table, long name
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.

Table 21. QQQ3007 - Summary Row for Optimizer Timed Out (continued)

Logical Column Name	Physical Column Name	Description
QVFILES	QVFILES	Number of tables joined
QVRCNT	QVRCNT	Unique refresh counter

Database monitor logical table 3008 - Summary Row for Subquery Processing

```

|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|
| A*
| A* DB Monitor logical table 3008 - Summary Row for Subquery Processing
| A*
| A          R QQQ3008          PTABLE(*CURLIB/QAQQDBMN)
| A          QQRID
| A          QQTIME
| A          QQJFLD
| A          QQRDBN
| A          QSYS
| A          QQJOB
| A          QQUSER
| A          QQJNUM
| A          QQTHRD          RENAME(QQI9) +
|                               COLHDG('Thread' +
|                                   'Identifier')
|
| A          QQUCNT
| A          QQUDEF
| A          QQQDTN
| A          QQQDTL
| A          QQMATN
| A          QQMATL
| A          QQMATLVL          RENAME(QVP15E) +
|                               COLHDG('Materialized' +
|                                   'Union' +
|                                   'Level')
|
| A          QDQDTN          RENAME(QVP15A) +
|                               COLHDG('Decomposed' +
|                                   'Subselect' +
|                                   'Number')
|
| A          QDQDTT          RENAME(QVP15B) +
|                               COLHDG('Number of' +
|                                   'Decomposed' +
|                                   'Subselects')
|
| A          QDQDTR          RENAME(QVP15C) +
|                               COLHDG('Decomposed' +
|                                   'Reason' +
|                                   'Code')
|
| A          QDQDTS          RENAME(QVP15D) +
|                               COLHDG('Starting' +
|                                   'Decomposed' +
|                                   'Subselect')
|
| A          QQORGQ          RENAME(QQI1) +
|                               COLHDG('Original' +
|                                   'Number' +
|                                   'of QDTs')
|
| A          QQMRGQ          RENAME(QQI2) +
|                               COLHDG('Number' +
|                                   'of QDTs' +
|                                   'Merged')
|
| A          QQFNLQ          RENAME(QQI3) +
|                               COLHDG('Final' +
|                                   'Number' +
|                                   'of QDTs')

```



```

|      A      QVRCNT
|      A      K  QQJFLD
|      A      S  QQRID                CMP(EQ 3008)

```

| Table 22. QQQ3008 - Summary Row for Subquery Processing

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQORGQ	QQI1	Original number of QDTs
QQMRGQ	QQI2	Number of QDTs merged
QQFNLQ	QQI3	Final number of QDTs
QVRCNT	QVRCNT	Unique refresh counter

Database monitor logical table 3010 - Summary for HostVar & ODP Implementation

```

|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
A*
A* DB Monitor logical table 3010 - Summary for HostVar & ODP Implementation
A*
A      R  QQQ3010                PTABLE(*CURLIB/QAQQDBMN)
A      QQRID
A      QQTIME
A      QQJFLD
A      QQRDBN
A      QSYS
A      QQJOB
A      QQUSER
A      QQJNUM

```

A	QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')
A	QQUCNT	RENAME(QQI5) +
A	QQRcnt	COLHDG('Refresh' + 'Counter')
A	QQUDEF	RENAME(QQC11) +
A	QQODPI	COLHDG('ODP' + 'Implementation')
A	QQHVI	RENAME(QQC12) + COLHDG('Host Variable' + 'Implementation')
A	QQHVAR	RENAME(QQ1000) + COLHDG('Host Variable' + 'Values')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3010)

Table 23. QQQ3010 - Summary for HostVar & ODP Implementation

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QQSYS	QQSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQRcnt	QQI5	Unique refresh counter
QQUDEF	QQUDEF	User defined column
QQODPI	QQC11	ODP implementation <ul style="list-style-type: none"> • R - Reusable ODP • N - Nonreusable ODP • ' ' - Column not used
QQHVI	QQC12	Host variable implementation <ul style="list-style-type: none"> • I - Interface supplied values (ISV) • V - Host variables treated as constants (V2) • U - Table management row positioning (UP)
QQHVAR	QQ1000	Host variable values

Database monitor logical table 3014 - Summary Row for Generic QQ Information

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|   A*
|   A* DB Monitor logical table 3014 - Summary Row for Generic QQ Information
|   A*
|   A          R  QQ3014          PTABLE(*CURLIB/QAQQDBMN)

```

	A	QQRID	
	A	QQTIME	
	A	QQJFLD	
	A	QQRDBN	
	A	QSYS	
	A	QQJOB	
	A	QQUSER	
	A	QQJNUM	
	A	QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')
	A	QQUCNT	
	A	QQUDEF	
	A	QQQDTN	
	A	QQQDTL	
	A	QQMATN	
	A	QQMATL	
	A	QQMATLVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')
	A	QDQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')
	A	QDQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')
	A	QDQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
	A	QDQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
	A	QREST	
	A	QQEPT	
	A	QQQTIM	RENAME(QQI1) + COLHDG('ODP' + 'Open' 'Time')
	A	QQORDG	
	A	QQGRPG	
	A	QQJNG	
	A	QQJNTY	RENAME(QQC22) + COLHDG('Join' + 'Type')
	A	QQUNIN	
	A	QQSUBQ	
	A	QQSSUB	RENAME(QWC1F) COLHDG('Scalar' + 'Subselects')
	A	QQHSTV	
	A	QQRCDL	
	A	QQGVNE	RENAME(QQC11) + COLHDG('Query' + 'Governor' + 'Enabled')
	A	QQGVNS	RENAME(QQC12) + COLHDG('Stopped' + 'by Query' + 'Governor')
	A	QQOPID	RENAME(QQC101) + COLHDG('Query' + 'Open ID')
	A	QQINLN	RENAME(QQC102) +

		COLHDG('Query' + 'Options' + 'Library')
A	QQINFN	RENAME(QQC103) + COLHDG('Query' + 'Options' + 'File')
A	QQEE	RENAME(QQC13) + COLHDG('Early' + 'Exit' + 'Indicator')
A	QVRCNT	
A	QVOPTIM	RENAME(QQI5) + COLHDG('Optimization' + 'Time')
A	QVAPRT	RENAME(QQTIM1) + COLHDG('Access Plan' + 'Rebuild' 'Timestamp')
A	QVOBYIM	RENAME(QVC11) + COLHDG('Ordering' + 'Implementation')
A	QVGBYIM	RENAME(QVC12) + COLHDG('Grouping' + 'Implementation')
A	QVJONIM	RENAME(QVC13) + COLHDG('Join' + 'Implementation')
A	QVDIST	RENAME(QVC14) + COLHDG('Distinct' + 'Query')
A	QVDSTRB	RENAME(QVC15) + COLHDG('Distributed' + 'Query')
A	QVDSTND	RENAME(QVC3001) + COLHDG('Distributed' + 'Nodes')
A	QVNLST	RENAME(QVC105) + COLHDG('Sort' + 'Sequence' + 'Table')
A	QVNLSSL	RENAME(QVC106) + COLHDG('Sort' + 'Sequence' + 'Library')
A	QVALWCY	RENAME(QVC16) + COLHDG('ALWCPYDTA' + 'Setting')
A	QVVAPRC	RENAME(QVC21) + COLHDG('Access Plan' + 'Rebuilt' + 'Code')
A	QVVAPSC	RENAME(QVC22) + COLHDG('Access Plan' + 'Rebuilt' + 'Subcode')
A	QVIMPLN	RENAME(QVC3002) + COLHDG('Implementation' + 'Summary')
A	QVUNIONL	RENAME(QWC16) + COLHDG('Last' + 'Part of' + 'Union')
A	DCMPFNLBLT	RENAME(QQC14) + COLHDG('Decomposed' + 'Final Cursor' + 'was Built')

A	DCMPFNLTMP	RENAME(QQC15) + COLHDG('This is' + 'Decomposed' + 'Final Cursor')
A	QQPSIZ	RENAME(QVP154) + COLHDG('Pool' + 'Size')
A	QQPID	RENAME(QVP155) + COLHDG('Pool' + 'ID')
A*	CHGQRYA or INI environment attributes used during execution of query	
A*		
A	QVMAXT	RENAME(QQI2) + COLHDG('Query' + 'Time' + 'Limit')
A	QVPARA	RENAME(QVC81) + COLHDG('Specified' + 'Parallel' + 'Option')
A	QVTASKN	RENAME(QQI3) + COLHDG('Mamimum' + 'Number of' + 'Tasks')
A	QVAPLYR	RENAME(QVC17) + COLHDG('Apply' + 'CHGQRYA' + 'Remotely')
A	QVASYNC	RENAME(QVC82) + COLHDG('Asynchronous' + 'Remote' + 'Job Usage')
A	QVFRCJO	RENAME(QVC18) + COLHDG('Join' + 'Order' + 'Forced')
A	QVDMGS	RENAME(QVC19) + COLHDG('Display' + 'DEBUG' + 'Messages')
A	QVPMCNV	RENAME(QVC1A) + COLHDG('Parameter' + 'Marker' + 'Conversion')
A	QVUDFTL	RENAME(QQI4) + COLHDG('UDF' + 'Time' + 'Limit')
A	QVOLMTS	RENAME(QVC1283) + COLHDG('Query' + 'Optimizer' + 'Limitations')
A	QVREOPT	RENAME(QVC1E) + COLHDG('Reoptimize' + 'Access' 'Plan')
A	QVOPALL	RENAME(QVC87) + COLHDG('Optimize' + 'All' 'Indexes')
A	QVDFQDTF	RENAME(QQC14) + COLHDG('Final' + 'Decomposed' + 'QDT Built')
A	QVDFQDT	RENAME(QQC15) + COLHDG('Final' + 'Decomposed' +

A	QVRDTRG	RENAME(QQC18) + COLHDG('Read' + 'Trigger')
A	QVSTRJN	RENAME(QQC81) + COLHDG('Star' + 'Join')
A	OPTGOAL	RENAME(QVC23) + COLHDG('Optimization' + 'Goal')
A	DIAGLIKE	RENAME(QVC24) + COLHDG('Visual' + 'Explain' + 'Diagram')
A	UNIONVIEW	RENAME(QQC23) + COLHDG('Union' + 'in a' + 'View')
A	SUBQTYPE	RENAME(QQC21) COLHDG('Type of' + 'Subselect')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3014)

Table 24. QQQ3014 - Summary Row for Generic QQ Information

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHR	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATLVL	QVP15E	Materialized view union level
QQQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QQQDTT	QVP15B	Total number of decomposed subselects
QQQDTR	QVP15C	Decomposed query subselect reason code
QQQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQREST	QQREST	Estimated number of rows selected
QQEPT	QQEPT	Estimated processing time, in seconds
QQQTIM	QQI1	Time spent to open cursor, in milliseconds

Table 24. QQQ3014 - Summary Row for Generic QQ Information (continued)

Logical Column Name	Physical Column Name	Description
QQORDG	QQORDG	Ordering (Y/N)
QQGRPG	QQGRPG	Grouping (Y/N)
QQJNG	QQJNG	Join Query (Y/N)
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join
QQUNIN	QQUNIN	Union Query (Y/N)
QSSUBQ	QSSUBQ	Subquery (Y/N)
QSSUB	QWC1F	Scalar Subselects (Y/N)
QQHSTV	QQHSTV	Host variables (Y/N)
QQRCD	QQRCD	Row selection (Y/N)
QQGVNE	QQC11	Query governor enabled (Y/N)
QQGVNS	QQC12	Query governor stopped the query (Y/N)
QQOPID	QQC101	Query open ID
QVINLN	QQC102	Query Options library name
QVINFN	QQC103	Query Options file name
QQEE	QQC13	Query early exit value
QVRCNT	QVRCNT	Unique refresh counter
QVOPTIM	QQI5	Time spent in optimizer, in milliseconds
QVAPRT	QQTIM1	Access Plan rebuilt timestamp, last time access plan was rebuilt.
QVOBYIM	QVC11	Ordering implementation. Possible values are: <ul style="list-style-type: none"> • I - Index • S - Sort
QVGBYIM	QVC12	Grouping implementation. Possible values are: <ul style="list-style-type: none"> • I - Index • H - Hash grouping
QVJONIM	QVC13	Join Implementation. Possible values are: <ul style="list-style-type: none"> • N - Nested Loop join • H - Hash join • C - Combination of Nested Loop and Hash
QVDIST	QVC14	Distinct query (Y/N)
QVDSTRB	QVC15	Distributed query (Y/N)
QVDSTND	QVC3001	Distributed nodes
QVNLST	QVC105	Sort Sequence Table
QVNLSSL	QVC106	Sort Sequence Library
QVALWC	QVC16	ALWCPYDTA setting
QVVAPRC	QVC21	Reason code why access plan was rebuilt
QVVAPSC	QVC22	Subcode why access plan was rebuilt
QVIMPLN	QVC3002	Summary of query implementation. Shows dataspace number and name of index used for each table being queried.

Table 24. QQQ3014 - Summary Row for Generic QQ Information (continued)

Logical Column Name	Physical Column Name	Description
QVUNIONL	QWC16	Last part (last QDT) of Union (Y/N)
DCMPFNLBLT	QWC14	A decomposed final temporary cursor was built (Y/N)
DCMPFNLTMP	QWC15	This is the decomposed final temporary cursor (final temporary QDT). (Y/N)
QVMAXT	QQI2	Query time limit
QVPARA	QVC81	Parallel Degree <ul style="list-style-type: none"> • *SAME - Don't change current setting • *NONE - No parallel processing is allowed • *I/O - Any number of tasks may be used for I/O processing. SMP parallel processing is not allowed. • *OPTIMIZE - The optimizer chooses the number of tasks to use for either I/O or SMP parallel processing. • *MAX - The optimizer chooses to use either I/O or SMP parallel processing. • *SYSVAL - Use the current system value to process the query. • *ANY - Has the same meaning as *I/O. • *NBRTASKS - The number of tasks for SMP parallel processing is specified in column QVTASKN.
QVTASKN	QQI3	Max number of tasks
QVAPLYR	QVC17	Apply CHGQRYA remotely (Y/N)
QVASYNC	QVC82	Asynchronous job usage <ul style="list-style-type: none"> • *SAME - Don't change current setting • *DIST - Asynchronous jobs may be used for queries with distributed tables • *LOCAL - Asynchronous jobs may be used for queries with local tables only • *ANY - Asynchronous jobs may be used for any database query • *NONE - No asynchronous jobs are allowed
QVFRCJO	QVC18	Force join order (Y/N)
QVDMSGs	QVC19	Print debug messages (Y/N)
QVPMCNV	QVC1A	Parameter marker conversion (Y/N)
QVUDFTL	QQI4	User Defined Function time limit
QVOLMTS	QVC1281	Optimizer limitations. Possible values: <ul style="list-style-type: none"> • *PERCENT followed by 2 byte integer containing the percent value • *MAX_NUMBER_OF_RECORDS followed by an integer value that represents the maximum number of rows
QVREOPT	QVC1E	Reoptimize access plan requested. Possible values are: <ul style="list-style-type: none"> • 'O' - Only reoptimize the access plan when absolutely required. Do not reoptimize for subjective reasons. • 'Y' - Yes, force the access plan to be reoptimized. • 'N' - No, do not reoptimize the access plan, unless optimizer determines that it is necessary. May reoptimize for subjective reasons.

Table 24. QQQ3014 - Summary Row for Generic QQ Information (continued)

Logical Column Name	Physical Column Name	Description
QVOPALL	QVC87	Optimize all indexes requested <ul style="list-style-type: none"> *SAME - Don't change current setting *YES - Examine all indexes *NO - Allow optimizer to time-out *TIMEOUT - Force optimizer to time-out
QVDFQDTF	QQC14	Final decomposed QDT built indicator (Y/N)
QVDFQDT	QQC15	This is the final decomposed QDT indicator (Y/N)
QVRDTRG	QQC18	One of the files contains a read trigger (Y/N)
QVSTRJN	QQC81	Star join optimization requested. <ul style="list-style-type: none"> *NO - Star join optimization will not be performed. *COST - The optimizer will determine if any EVIs can be used for star join optimization. *FORCE - The optimizer will add any EVIs that can be used for star join optimization.
OPTGOAL	QVC23	Byte 1 = Optimization goal. Possible values are: <ul style="list-style-type: none"> 'F' - First I/O, optimize the query to return the first screen full of rows as quickly as possible. 'A' - All I/O, optimize the query to return all rows as quickly as possible.
DIAGLIKE	QVC24	Byte 1 = Type of Visual Explain diagram. Possible values are: <ul style="list-style-type: none"> 'D' - Detail 'B' - Basic Byte 2 - Ignore LIKE redundant shifts. Possible values are: <ul style="list-style-type: none"> 'O' - Optimize, the query optimizer determines which redundant shifts to ignore. 'A' - All, all redundant shifts will be ignored.
UNIONVIEW	QQC23	Byte 1 = This QDT is part of a UNION that is contained within a view (Y/N) Byte 2 = This QDT is the last subselect of the UNION that is contained within a view (Y/N)
SUBQTYPE	QQC21	Type of Subselect. Possible values are: <ul style="list-style-type: none"> 'SS' - Scalar subselect 'SU' - Update with Set subselect 'SQ' - Subquery

Database monitor logical table 3015 - Summary Row for Statistics Information

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|   A*
|   A* DB Monitor logical table 3015 - Summary Row for Statistics Information
|   A*
|   A      R QQQ3015      PTABLE(*CURLIB/QAQQDBMN)
|   A      QQRID
|   A      QQTIME
|   A      QQJFLD

```

A	QQRDBN	
A	QSYS	
A	QQJOB	
A	QQUSER	
A	QQJNUM	
A	QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')
A	QQUCNT	
A	QQUDEF	
A	QQQDTN	
A	QQQDTL	
A	QQMATN	
A	QQMATL	
A	QQMATULVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')
A	QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')
A	QQQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')
A	QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
A	QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
A	QQTLN	
A	QQTFN	
A	QQTMN	
A	QQPTFN	
A	QQPTMN	
A	QVQTBL	
A	QVQLIB	
A	QVPTBL	
A	QVPLIB	
A	QQNTNM	
A	QQNLNM	
A	QVSTATUS	RENAME(QQC11) + COLHDG('Statistic' + 'Status')
A	QVSTATIMP	RENAME(QQi2) + COLHDG('Statistic' + 'Importance')
A	QVSTATCOL	RENAME(QQ1000) + COLHDG('Column' + 'Names')
A	QVSTATID	RENAME(QVC1000) + COLHDG('Statistic' + 'Identifier')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3015)

Table 25. QQQ3015 - Summary Row for Statistic Information

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created

Table 25. QQQ3015 - Summary Row for Statistic Information (continued)

Logical Column Name	Physical Column Name	Description
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QQSYS	QQSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQTLN	QQTLN	Library of table queried
QQTFN	QQTFN	Name of table queried
QQTMN	QQTMN	Member name of table queried
QQPTLN	QQPTLN	Library name of base table
QQPTFN	QQPTFN	Name of the base table queried
QQPTMN	QQPTMN	Member name of base table
QVQTBL	QVQTBL	Queried table, long name
QVQLIB	QVQLIB	Library of queried table, long name
QVPTBL	QVPTBL	Base table, long name
QVPLIB	QVPLIB	Library of base table, long name
QQVTNM	QQNTNM	NLSS table
QQNLNM	QQNLNM	NLSS library
QVSTATUS	QQC11	Statistic Status. Possible values are: <ul style="list-style-type: none"> • 'N' - No statistic • 'S' - Stale statistic • '' - Unknown
QVSTATIMP	QQI2	Importance of this statistic
QVSTATCOL	QQ1000	Columns for the statistic advised
QVSTATID	QVC1000	Statistic identifier

Database monitor logical table 3018 - Summary Row for STRDBMON/ENDDDBMON

```

|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
A*
A* DB Monitor logical table 3018 - Summary Row for STRDBMON/ENDDDBMON
A*
A      R  QQQ3018          PTABLE(*CURLIB/QAQQDBMN)
A      QQRID
A      QQTIME
A      QQJFLD
A      QQRDBN
A      QQSYS
A      QQJOB
A      QQUSER
A      QQJNUM
A      QQTHRD          RENAME(QQI9) +
                        COLHDG('Thread' +
                        'Identifier')
A      QQJOBT          RENAME(QQC11)+
                        COLHDG('Job' +
                        'Type')
A      QQCMDT          RENAME(QQC12) +
                        COLHDG('Command' +
                        'Type')
A      QQJOBI          RENAME(QQC301) +
                        COLHDG('Job' +
                        'Info')
A      K  QQJFLD
A      S  QQRID          CMP(EQ 3018)

```

Table 26. QQQ3018 - Summary Row for STRDBMON/ENDDDBMON

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QQSYS	QQSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQJOBT	QQC11	Type of job monitored <ul style="list-style-type: none"> • C - Current • J - Job name • A - All
QQCMDT	QQC12	Command type <ul style="list-style-type: none"> • S - STRDBMON • E - ENDDDBMON
QQJOBI	QQC301	Monitored job information <ul style="list-style-type: none"> • * - Current job • Job number/User/Job name • *ALL - All jobs

Database monitor logical table 3019 - Detail Row for Rows Retrieved

```

|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|
| A*
| A* DB Monitor logical table 3019 - Detail Row for Rows Retrieved
| A*
| A          R QQQ3019          PTABLE(*CURLIB/QAQQDBMN)
| A          QQRID
| A          QQTIME
| A          QQJFLD
| A          QQRDBN
| A          QQSYS
| A          QQJOB
| A          QQUSER
| A          QQJNUM
| A          QQTHRD          RENAME(QQ19) +
|                               COLHDG('Thread' +
|                                   'Identifier')
|
| A          QQUCNT
| A          QQUDEF
| A          QQQDTN
| A          QQQDTL
| A          QQMATN
| A          QQMATL
| A          QQMATULVL      RENAME(QVP15E) +
|                               COLHDG('Materialized' +
|                                   'Union' +
|                                   'Level')
|
| A          QDQDTN          RENAME(QVP15A) +
|                               COLHDG('Decomposed' +
|                                   'Subselect' +
|                                   'Number')
|
| A          QDQDTT          RENAME(QVP15B) +
|                               COLHDG('Number of' +
|                                   'Decomposed' +
|                                   'Subselects')
|
| A          QDQDTR          RENAME(QVP15C) +
|                               COLHDG('Decomposed' +
|                                   'Reason' +
|                                   'Code')
|
| A          QDQDTS          RENAME(QVP15D) +
|                               COLHDG('Starting' +
|                                   'Decomposed' +
|                                   'Subselect')
|
| A          QQCPUT          RENAME(QQ11) +
|                               COLHDG('Row' +
|                                   'Retrieval' +
|                                   'CPU Time')
|
| A          QQCLKT          RENAME(QQ12) +
|                               COLHDG('Row' +
|                                   'Retrieval' +
|                                   'Clock Time')
|
| A          QQSYNR          RENAME(QQ13) +
|                               COLHDG('Synch' +
|                                   'Reads')
|
| A          QQSYNW          RENAME(QQ14) +
|                               COLHDG('Synch' +
|                                   'Writes')
|
| A          QQASYR          RENAME(QQ15) +
|                               COLHDG('Asynch' +
|                                   'Reads')
|
| A          QQASYW          RENAME(QQ16) +
|                               COLHDG('Asynch' +
|                                   'Writes')
|
| A          QQRCDR          RENAME(QQ17) +
|                               COLHDG('Rows' +
|                                   'Returned')

```

```

|      A          QQGETC          RENAME(QQ18) +
|                                     COLHDG('Number' +
|                                     'of Gets')
|
|      A          K QQJFLD
|      A          S QQRID          CMP(EQ 3019)

```

Table 27. QQQ3019 - Detail Row for Rows Retrieved

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATLVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QQCPUT	QQI1	CPU time to return all rows, in milliseconds
QQCLKT	QQI2	Clock time to return all rows, in milliseconds
QQSYNR	QQI3	Number of synchronous database reads
QQSYNW	QQI4	Number of synchronous database writes
QQASYR	QQI5	Number of asynchronous database reads
QQASYW	QQI6	Number of asynchronous database writes
QQRCDR	QQI7	Number of rows returned
QQGETC	QQI8	Number of calls to retrieve rows returned

Database monitor logical table 3021 - Summary Row for Bitmap Created

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
| A*
| A* DB Monitor logical table 3021 - Summary Row for Bitmap Created
| A*
| A* New row added for Visual Explain

```

	A*		
	A	R QQQ3021	PTABLE(*CURLIB/QAQQDBMN)
	A	QQRID	
	A	QQTIME	
	A	QQJFLD	
	A	QQRDBN	
	A	QQSYS	
	A	QQJOB	
	A	QQUSER	
	A	QQJNUM	
	A	QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')
	A	QQUCNT	
	A	QQUDEF	
	A	QQQDTN	
	A	QQQDTL	
	A	QQMATN	
	A	QQMATL	
	A	QQMATULVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')
	A	QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')
	A	QQQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')
	A	QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
	A	QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
	A	QVRCNT	
	A	QVPARPF	
	A	QVPARPL	
	A	QVPARD	
	A	QVPARU	
	A	QVPARRC	
	A	QQEPT	
	A	QVCTIM	
	A	QQREST	
	A	QQAJN	
	A	QQJNP	
	A	QQJNDS	RENAME(QQI6) + COLHDG('Data Space' + 'Number')
	A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
	A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
	A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
	A	QVJFANO	
	A	QVFILES	
	A	QVBMSIZ	RENAME(QQI2) + COLHDG('Bitmap' + 'Size')
	A	QVBCMCNT	RENAME(QVP151) + COLHDG('Number of' + 'Bitmaps')

```

|                                     'Created')
|      A          QVBMIDS             RENAME(QVC3001) +
|                                     COLHDG('Internal' +
|                                     'Bitmap' 'IDs')
|      A          K QQJFLD
|      A          S QQRID             CMP(EQ 3021)

```

Table 28. QQQ3021 - Summary Row for Bitmap Created

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHR	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATLVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QVRCNT	QVRCNT	Unique refresh counter
QVPRPF	QVPRPF	Parallel Prefetch (Y/N)
QVPRPL	QVPRPL	Parallel Preload (index used)
QVPARD	QVPARD	Parallel degree requested (index used)
QVPARU	QVPARU	Parallel degree used (index used)
QVPARRC	QVPARRC	Reason parallel processing was limited (index used)
QQEPT	QQEPT	Estimated processing time, in seconds
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QQREST	QQREST	Estimated rows selected
QQAJN	QQAJN	Estimated number of joined rows
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number/Original table position

Table 28. QQQ3021 - Summary Row for Bitmap Created (continued)

Logical Column Name	Physical Column Name	Description
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> NL - Nested loop MF - Nested loop with selection HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> IN - Inner join PO - Left partial outer join EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVBMSIZ	QQI2	Bitmap size
QVBMCNT	QVP151	Number of bitmaps created
QVBMIDS	QVC3001	Internal bitmap IDs

Database monitor logical table 3022 - Summary Row for Bitmap Merge

```

|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
A*
A* DB Monitor logical table 3022 - Summary Row for Bitmap Merge
A*
A* New row added for Visual Explain
A*
A      R QQQ3022                PTABLE(*CURLIB/QAQQDBMN)
A      QQRID
A      QQTIME
A      QQJFLD
A      QQRDBN
A      QQSYS
A      QQJOB
A      QQUSER
A      QQJNUM
A      QQTHRD                RENAME(QQI9) +
                              COLHDG('Thread' +
                              'Identifier')
A      QQCNT
A      QQDEF

```

A	QQQDTN	
A	QQQDTL	
A	QQMATN	
A	QQMATL	
A	QQMATULVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')
A	QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')
A	QQQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')
A	QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
A	QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
A	QVRCNT	
A	QVPARPF	
A	QVPARPL	
A	QVPARD	
A	QVPARU	
A	QVPARRC	
A	QQEPT	
A	QVCTIM	
A	QQREST	
A	QQAJN	
A	QQJNP	
A	QQJNDS	RENAME(QQI6) + COLHDG('Data Space' + 'Number')
A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
A	QVJFANO	
A	QVFILES	
A	QVBMSIZ	RENAME(QQI2) + COLHDG('Bitmap' + 'Size')
A	QVBMID	RENAME(QVC101) + COLHDG('Internal' + 'Bitmap' 'ID')
A	QVBMIDMG	RENAME(QVC3001) + COLHDG('Bitmaps' + 'Merged')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3022)

Table 29. QQQ3022 - Summary Row for Bitmap Merge

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)

Table 29. QQQ3022 - Summary Row for Bitmap Merge (continued)

Logical Column Name	Physical Column Name	Description
QQRDBN	QQRDBN	Relational database name
QQSYS	QQSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMQTULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QVRCNT	QVRCNT	Unique refresh counter
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (index used)
QVPARD	QVPARD	Parallel degree requested (index used)
QVPARU	QVPARU	Parallel degree used (index used)
QVPARRC	QVPARRC	Reason parallel processing was limited (index used)
QQEPT	QQEPT	Estimated processing time, in seconds
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QQREST	QQREST	Estimated rows selected
QQAJN	QQAJN	Estimated number of joined rows
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number/Original table position
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join

Table 29. QQQ3022 - Summary Row for Bitmap Merge (continued)

Logical Column Name	Physical Column Name	Description
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVBMSIZ	QQI2	Bitmap size
QVBMID	QVC101	Internal bitmap ID
QVBMIDMG	QVC3001	IDs of bitmaps merged together

Database monitor logical table 3023 - Summary for Temp Hash Table Created

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|   A*
|   A* DB Monitor logical table 3023 - Summary for Temp Hash Table Created
|   A*
|   A* New row added for Visual Explain
|   A*
|   A      R QQQ3023                PTABLE(*CURLIB/QAQQDBMN)
|   A      QQRID
|   A      QQTIME
|   A      QQJFLD
|   A      QQRDBN
|   A      QSYS
|   A      QQJOB
|   A      QQUSER
|   A      QQJNUM
|   A      QQTHRD                RENAME(QQI9) +
|                                   COLHDG('Thread' +
|                                       'Identifier')
|
|   A      QQCNT
|   A      QQUDEF
|   A      QQQDTN
|   A      QQQDTL
|   A      QQMATN
|   A      QQMATL
|   A      QQMATLVL            RENAME(QVP15E) +
|                                   COLHDG('Materialized' +
|                                       'Union' +
|                                       'Level')
|
|   A      QQQDTN            RENAME(QVP15A) +
|                                   COLHDG('Decomposed' +

```

		'Subselect' + 'Number')
A	QDQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')
A	QDQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
A	QDQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
A	QVRCNT	
A	QVPARPF	
A	QVPARPL	
A	QVPARD	
A	QVPARU	
A	QVPARRC	
A	QQEPT	
A	QVCTIM	
A	QQREST	
A	QQAJN	
A	QQJNP	
A	QQJNDS	RENAME(QQI8) + COLHDG('Data Space' + 'Number')
A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
A	QVJFANO	
A	QVFILES	
A	QVHTRC	RENAME(QVC1F) + COLHDG('Hash' + 'Table' + 'Reason Code')
A	QVHTENT	RENAME(QQI2) + COLHDG('Hash' + 'Table' + 'Entries')
A	QVHTSIZ	RENAME(QQI3) + COLHDG('Hash' + 'Table' + 'Size')
A	QVHTRSIZ	RENAME(QQI4) + COLHDG('Hash' + 'Table' + 'Row' 'Size')
A	QVHTKSIZ	RENAME(QQI5) + COLHDG('Hash' + 'Key' + 'Size')
A	QVHTESIZ	RENAME(QQI6) + COLHDG('Hash' + 'Element' + 'Size')
A	QVHTPSIZ	RENAME(QQI7) + COLHDG('Pool' + 'Size')
A	QVHTPID	RENAME(QQI8) + COLHDG('Pool' + 'ID')
A	QVHTNAM	RENAME(QVC101) +

```

|                                     COLHDG('Hash' +
|                                     'Table' +
|                                     'Name')
|     A          QVHTLIB              RENAME(QVC102) +
|                                     COLHDG('Hash' +
|                                     'Table' +
|                                     'Library')
|     A          QVHTCOL              RENAME(QVC3001) +
|                                     COLHDG('Hash' +
|                                     'Table' +
|                                     'Columns')
|     A          K QQJFLD
|     A          S QQRID              CMP(EQ 3023)

```

Table 30. QQQ3023 - Summary for Temp Hash Table Created

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYSYS	QSYSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QVRCNT	QVRCNT	Unique refresh counter
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (index used)
QVPARD	QVPARD	Parallel degree requested (index used)
QVPARU	QVPARU	Parallel degree used (index used)
QVPARRC	QVPARRC	Reason parallel processing was limited (index used)
QQEPT	QQEPT	Estimated processing time, in seconds
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QQREST	QQREST	Estimated rows selected

Table 30. QQQ3023 - Summary for Temp Hash Table Created (continued)

Logical Column Name	Physical Column Name	Description
QQAJN	QQAJN	Estimated number of joined rows
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number/Original table position
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> NL - Nested loop MF - Nested loop with selection HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> IN - Inner join PO - Left partial outer join EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVHTRC	QVC1F	Hash table reason code <ul style="list-style-type: none"> J - Created for hash join G - Created for hash grouping
QVHTENT	QQI2	Hash table entries
QVHTSIZ	QQI3	Hash table size
QVHTRSIZ	QQI4	Hash table row size
QVHTKSIZ	QQI5	Hash table key size
QVHTESIZ	QQIA	Hash table element size
QVHTPSIZ	QQI7	Hash table pool size
QVHTPID	QQI8	Hash table pool ID
QVHTNAM	QVC101	Hash table internal name
QVHTLIB	QVC102	Hash table library
QVHTCOL	QVC3001	Columns used to create hash table

Database monitor logical table 3025 - Summary Row for Distinct Processing

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
| |
| |A*
| |A* DB Monitor logical table 3025 - Summary Row for Distinct Processing
| |A*
| |A* New row added for Visual Explain
| |A*
| |A          R  QQQ3025          PTABLE(*CURLIB/QAQQDBMN)
| |A          QQRID
| |A          QQTIME
| |A          QQJFLD
| |A          QQRDBN
| |A          QQSYS
| |A          QQJOB
| |A          QQUSER
| |A          QQJNUM
| |A          QQTHRD          RENAME(QQI9) +
| |                          COLHDG('Thread' +
| |                          'Identifier')
| |
| |A          QQUCNT
| |A          QQUDEF
| |A          QQQDTN
| |A          QQQDTL
| |A          QQMATN
| |A          QQMATL
| |A          QQMATULVL      RENAME(QVP15E) +
| |                          COLHDG('Materialized' +
| |                          'Union' +
| |                          'Level')
| |
| |A          QQQDTN          RENAME(QVP15A) +
| |                          COLHDG('Decomposed' +
| |                          'Subselect' +
| |                          'Number')
| |
| |A          QQQDTT          RENAME(QVP15B) +
| |                          COLHDG('Number of' +
| |                          'Decomposed' +
| |                          'Subselects')
| |
| |A          QQQDTR          RENAME(QVP15C) +
| |                          COLHDG('Decomposed' +
| |                          'Reason' +
| |                          'Code')
| |
| |A          QQQDTS          RENAME(QVP15D) +
| |                          COLHDG('Starting' +
| |                          'Decomposed' +
| |                          'Subselect')
| |
| |A          QVRCNT
| |A          QVPARPF
| |A          QVPARPL
| |A          QVPARD
| |A          QVPARU
| |A          QVPARRC
| |A          QQEPT
| |A          QVCTIM
| |A          QQREST
| |A          K  QQJFLD
| |A          S  QQRID          CMP(EQ 3025)

```

Table 31. QQQ3025 - Summary Row for Distinct Processing

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created

Table 31. QQQ3025 - Summary Row for Distinct Processing (continued)

Logical Column Name	Physical Column Name	Description
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect
QVRCNT	QVRCNT	Unique refresh counter
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (index used)
QVPARD	QVPARD	Parallel degree requested (index used)
QVPARU	QVPARU	Parallel degree used (index used)
QVPARRC	QVPARRC	Reason parallel processing was limited (index used)
QQEPT	QQEPT	Estimated processing time, in seconds
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QQREST	QQREST	Estimated rows selected

Database monitor logical table 3027 - Summary Row for Subquery Merge

```

| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
|   A*
|   A* DB Monitor logical table 3027 - Summary Row for Subquery Merge
|   A*
|   A* New row added for Visual Explain
|   A*
|   A      R QQQ3027          PTABLE(*CURLIB/QAQQDBMN)
|   A      QQRID
|   A      QQTIME
|   A      QQJFLD
|   A      QQRDBN
|   A      QSYS

```

	A	QQJOB	
	A	QQUSER	
	A	QQJNUM	
	A	QQTHRD	RENAME(QQI9) + COLHDG('Thread' + 'Identifier')
	A	QQUCNT	
	A	QQUDEF	
	A	QQQDTN	
	A	QQQDTL	
	A	QQMATN	
	A	QQMATL	
	A	QQMATULVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')
	A	QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')
	A	QQQDTT	RENAME(QVP15B) + COLHDG('Number of' + 'Decomposed' + 'Subselects')
	A	QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
	A	QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
	A	QVRCNT	
	A	QVPARPF	
	A	QVPARPL	
	A	QVPARD	
	A	QVPARU	
	A	QVPARRC	
	A	QQEPT	
	A	QVCTIM	
	A	QQREST	
	A	QQAJN	
	A	QQJNP	
	A	QQJNDS	RENAME(QQI6) + COLHDG('Data Space' + 'Number')
	A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
	A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
	A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
	A	QVJFANO	
	A	QVFILES	
	A	QVIQDTN	RENAME(QVP151) + COLHDG('Subselect' + 'Number' + 'Inner')
	A	QVIQDTL	RENAME(QVP152) + COLHDG('Subselect' + 'Level' + 'Inner')
	A	QVIMATN	RENAME(QVP153) + COLHDG('View' + 'Number' + 'Inner')
	A	QVIMATL	RENAME(QVP154) +

Table 32. QQQ3027 - Summary Row for Subquery Merge (continued)

Logical Column Name	Physical Column Name	Description
QVPARD	QVPARD	Parallel degree requested (index used)
QVPARU	QVPARU	Parallel degree used (index used)
QVPARRC	QVPARRC	Reason parallel processing was limited (index used)
QQEPT	QQEPT	Estimated processing time, in seconds
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QQREST	QQREST	Estimated rows selected
QQAJN	QQAJN	Estimated number of joined rows
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number/Original table position
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVIQDTN	QVP151	Subselect number for inner subquery
QVIQDTL	QVP152	Subselect level for inner subquery
QVIMATN	QVP153	Materialized view subselect number for inner subquery
QVIMATL	QVP154	Materialized view subselect level for inner subquery
QVIMATUL	QVP155	Materialized view union level for inner subquery

Table 32. QQQ3027 - Summary Row for Subquery Merge (continued)

Logical Column Name	Physical Column Name	Description
QVSUBOP	QQC101	Subquery operator. Possible values are: <ul style="list-style-type: none"> EQ - Equal NE - Not Equal LT - Less Than or Equal LT - Less Than GE - Greater Than or Equal GT - Greater Than IN LIKE EXISTS NOT - Can precede IN, LIKE or EXISTS
QVSSUBTYP	QVC21	Subquery type. Possible values are: <ul style="list-style-type: none"> SQ - Subquery SS - Scalar subselect SU - Set Update
QVCORRI	QQC11	Correlated columns exist (Y/N)
QVCORRC	QVC3001	List of correlated columns with corresponding QDT number

Database monitor logical table 3028 - Summary Row for Grouping

```

| |...+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
| A*
| A* DB Monitor logical table 3028 - Summary Row for Grouping
| A*
| A* New row added for Visual Explain
| A*
| A          R  QQQ3028          PTABLE(*CURLIB/QAQQDBMN)
| A          QQRID
| A          QQTIME
| A          QQJFLD
| A          QQRDBN
| A          QSYS
| A          QQJOB
| A          QQUSER
| A          QQJNUM
| A          QQTHRD          RENAME(QQI9) +
|                               COLHDG('Thread' +
|                                   'Identifier')
|
| A          QQUCNT
| A          QQUDEF
| A          QQQDTN
| A          QQQDTL
| A          QQMATN
| A          QQMATL
| A          QQMATLVL          RENAME(QVP15E) +
|                               COLHDG('Materialized' +
|                                   'Union' +
|                                   'Level')
|
| A          QDQDTN          RENAME(QVP15A) +
|                               COLHDG('Decomposed' +
|                                   'Subselect' +
|                                   'Number')
|
| A          QDQDTT          RENAME(QVP15B) +
|                               COLHDG('Number of' +

```

		'Decomposed' + 'Subselects')
A	QQQDTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
A	QQQDTS	RENAME(QVP15D) + COLHDG('Starting' + 'Decomposed' + 'Subselect')
A	QVRCNT	
A	QVPARPF	
A	QVPARPL	
A	QVPARD	
A	QVPARU	
A	QVPARRC	
A	QQEPT	
A	QVCTIM	
A	QQREST	
A	QQAJN	
A	QQJNP	
A	QQJNDS	RENAME(QQI6) + COLHDG('Data Space' + 'Number')
A	QQJNMT	RENAME(QQC21) + COLHDG('Join' 'Method')
A	QQJNTY	RENAME(QQC22) + COLHDG('Join' 'Type')
A	QQJNOP	RENAME(QQC23) + COLHDG('Join' 'Operator')
A	QVJFANO	
A	QVFILES	
A	QVGBYIM	RENAME(QQC11) + COLHDG('Grouping' + 'Implementation')
A	QVGBYIT	RENAME(QQC15) + COLHDG('Index' + 'Type')
A	QVGBYIX	RENAME(QQC101) + COLHDG('Grouping' + 'Index')
A	QVGBYIL	RENAME(QQC102) + COLHDG('Grouping' + 'Index' + 'Library')
A	QVGBYIXL	RENAME(QVINAM) + COLHDG('Grouping' + 'Index' + 'Long Name')
A	QVGBYILL	RENAME(QVILIB) + COLHDG('Grouping' + 'Library' + 'Long Name')
A	QVGBYHV	RENAME(QQC12) + COLHDG('Having' + 'Selection' + 'Exists')
A	QVGBYHW	RENAME(QQC13) + COLHDG('Having to' + 'Where' + 'Conversion')
A	QVGBYN	RENAME(QQI2) + COLHDG('Estimated' + 'Number of' + 'Groups')
A	QVGBYNA	RENAME(QQI3) + COLHDG('Average' +

		'Rows per' + 'Group')
A	QVGBYCOL	RENAME(QVC3001) + COLHDG('Grouping' + 'Columns')
A	QVGBYMIN	RENAME(QVC3002) + COLHDG('MIN' + 'Columns')
A	QVGBYMAX	RENAME(QVC3003) + COLHDG('MAX' + 'Columns')
A	QVGBYSUM	RENAME(QVC3004) + COLHDG('SUM' + 'Columns')
A	QVGBYCNT	RENAME(QVC3005) + COLHDG('COUNT' + 'Columns')
A	QVGBYAVG	RENAME(QVC3006) + COLHDG('AVG' + 'Columns')
A	QVGBYSTD	RENAME(QVC3007) + COLHDG('STDDEV' + 'Columns')
A	QVGBYVAR	RENAME(QVC3008) + COLHDG('VAR' + 'Columns')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3028)

Table 33. QQQ3028 - Summary Row for Grouping

Logical Column Name	Physical Column Name	Description
QQRID	QQRID	Row identification
QQTIME	QQTIME	Time row was created
QQJFLD	QQJFLD	Join column (unique per job)
QQRDBN	QQRDBN	Relational database name
QSYS	QSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number
QQTHRD	QQI9	Thread identifier
QQUCNT	QQUCNT	Unique count (unique per query)
QQUDEF	QQUDEF	User defined column
QQQDTN	QQQDTN	Unique subselect number
QQQDTL	QQQDTL	Subselect nested level
QQMATN	QQMATN	Materialized view subselect number
QQMATL	QQMATL	Materialized view nested level
QQMATULVL	QVP15E	Materialized view union level
QDQDTN	QVP15A	Decomposed query subselect number, unique across all decomposed subselects
QDQDTT	QVP15B	Total number of decomposed subselects
QDQDTR	QVP15C	Decomposed query subselect reason code
QDQDTS	QVP15D	Decomposed query subselect number for the first decomposed subselect

| Table 33. QQQ3028 - Summary Row for Grouping (continued)

Logical Column Name	Physical Column Name	Description
QVRCNT	QVRCNT	Unique refresh counter
QVPARPF	QVPARPF	Parallel Prefetch (Y/N)
QVPARPL	QVPARPL	Parallel Preload (index used)
QVPARD	QVPARD	Parallel degree requested (index used)
QVPARU	QVPARU	Parallel degree used (index used)
QVPARRC	QVPARRC	Reason parallel processing was limited (index used)
QQEPT	QQEPT	Estimated processing time, in seconds
QVCTIM	QVCTIM	Estimated cumulative time, in seconds
QQREST	QQREST	Estimated rows selected
QQAJN	QQAJN	Estimated number of joined rows
QQJNP	QQJNP	Join position
QQJNDS	QQI1	dataspace number/original table position
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.
QVFILES	QVFILES	Number of tables joined
QVGBYI	QQC11	Groupby implementation <ul style="list-style-type: none"> • ' ' - No grouping • I - Index • H - Hash

| Table 33. QQQ3028 - Summary Row for Grouping (continued)

Logical Column Name	Physical Column Name	Description
QVGBYIT	QQC15	Type of Index. Possible values are: <ul style="list-style-type: none"> • B - Binary Radix Index • C - Constraint (Binary Radix) • E - Encoded Vector Index (EVI) • X - Query created temporary index
QVGBYIX	QQC101	Index, or constraint, used for grouping
QVGBYIL	QQC102	Library of index used for grouping
QVGBYIXL	QVINAM	Long name of index, or constraint, used for grouping
QVGBYILL	QVILIB	Long name of index, or constraint, library used for grouping
QVGBYHV	QQC12	Having selection exists (Y/N)
QVGBYHW	QQC13	Having to Where conversion (Y/N)
QVGBYN	QQI2	Estimated number of groups
QVGBYNA	QQI3	Average number of rows in each group
QVGBYCOL	QVC3001	Grouping columns
QVGBYMIN	QVC3002	MIN columns
QVGBYMAX	QVC3003	MAX columns
QVGBYSUM	QVC3004	SUM columns
QVGBYCNT	QVC3005	COUNT columns
QVGBYAVG	QVC3006	AVG columns
QVGBYSTD	QVC3007	STDDEV columns
QVGBYVAR	QVC3008	VAR columns

Appendix B. Memory Resident Database Monitor: DDS

This appendix contains the following DDS that is used to create the memory resident database monitor physical and logical files.

- “External table description (QAQQQRYI) - Summary Row for SQL Information”
- “External table description (QAQQTEXT) - Summary Row for SQL Statement” on page 229
- “External table description (QAQQ3000) - Summary Row for Arrival Sequence” on page 229
- “External table description (QAQQ3001) - Summary row for Using Existing Index” on page 231
- “External table description (QAQQ3002) - Summary Row for Index Created” on page 233
- “External table description (QAQQ3003) - Summary Row for Query Sort” on page 235
- “External table description (QAQQ3004) - Summary Row for Temporary Table” on page 236
- “External table description (QAQQ3007) - Summary Row for Optimizer Information” on page 238
- “External table description (QAQQ3008) - Summary Row for Subquery Processing” on page 239
- “External table description (QAQQ3010) - Summary Row for Host Variable and ODP Implementation” on page 239

External table description (QAQQQRYI) - Summary Row for SQL Information

Table 34. QAQQQRYI - Summary Row for SQL Information

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQJOB	Job name
QQUSER	Job user
QQJNUM	Job number
QQTHID	Thread Id
QQUDEF	User defined column
QQPLIB	Name of the library containing the program or package
QQCNAM	Cursor name
QQPNAM	Name of the package or name of the program that contains the current SQL statement
QQSNAM	Name of the statement for SQL statement, if applicable
QQCNT	Statement usage count
QQAVGT	Average runtime (ms)
QQMINT	Minimum runtime (ms)
QQMAXT	Maximum runtime (ms)
QQOPNT	Open time for most expensive execution (ms)
QQFETT	Fetch time for most expensive execution (ms)
QQCLST	Close time for most expensive execution (ms)
QQOHT	Other time for most expensive execution (ms)
QQLTU	Time statement last used
QQMETU	Most expensive time used

| *Table 34. QAQQRYI - Summary Row for SQL Information (continued)*

Column Name	Description
QQAPRT	Access plan rebuild time
QQFULO	Number of full opens
QQPSUO	Number of pseudo-opens
QQTOTR	Total rows in table if non-join
QQRROW	Number of result rows returned
QQRROW	Statement function
	S - Select - Update
	I - Insert
	D - Delete
	L - Data definition language
	O - Other

Table 34. QAQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQSTOP	Statement operation
	<ul style="list-style-type: none"> • AL - Alter table • CA - Call • CC - Create collection • CD - Create type • CF - Create function • CG - Create trigger • CI - Create index • CL - Close • CM - Commit • CN - Connect • CO - Comment on • CP - Create procedure • CS - Create alias/synonym • CT - Create table • CV - Create view • DE - Describe • DI - Disconnect • DL - Delete • DM - Describe parameter marker • DP - Declare procedure • DR - Drop • DT - Describe table • EI - Execute immediate • EX - Execute • FE - Fetch • FL - Free locator • GR - Grant • HC - Hard close • HL - Hold locator • IN - Insert • JR - Server job reused • LK - Lock • LO - Label on • MT - More text • OP - Open • PD - Prepare and describe • PR - Prepare • RB - Rollback Savepoint • RE - Release • RO - Rollback

| Table 34. QAQQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQSTOP (continued)	<ul style="list-style-type: none"> • RS - Release Savepoint • RT - Rename table • RV - Revoke • SA - Savepoint • SC - Set connection • SI - Select into • SP - Set path • SR - Set result set • SS - Set current schema • ST - Set transaction • SV - Set variable • UP - Update • VI - Values into
QQODPI	ODP implementation
	<ul style="list-style-type: none"> R - Reusable ODP (ISV) N - Non-reusable ODP (V2)
QQHVI	Host variable implementation
	<ul style="list-style-type: none"> I - Interface supplied values (ISV) V - Host variables treated as constants (V2) U - Table management row positioning (UP)

Table 34. QAQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQAPR	Access plan rebuilt
	<p>A1 A table or member is not the same object as the one referenced when the access plan was last built. Some reasons they could be different are:</p> <ul style="list-style-type: none"> • Object was deleted and recreated. • Object was saved and restored. • Library list was changed. • Object was renamed. • Object was moved. • Object was overridden to a different object. • This is the first run of this query after the object containing the query has been restored.
	<p>A2 Access plan was built to use a reusable Open Data Path (ODP) and the optimizer chose to use a non-reusable ODP for this call.</p>
	<p>A3 Access plan was built to use a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP for this call.</p>
	<p>A4 The number of rows in the table has changed by more than 10% since the access plan was last built.</p>
	<p>A5 A new index exists over one of the tables in the query.</p>
	<p>A6 An index that was used for this access plan no longer exists or is no longer valid.</p>
	<p>A7 OS/400 Query requires the access plan to be rebuilt because of system programming changes.</p>
	<p>A8 The CCSID of the current job is different than the CCSID of the job that last created the access plan.</p>
	<p>A9 The value of one or more of the following is different for the current job than it was for the job that last created this access plan:</p> <ul style="list-style-type: none"> • date format • date separator • time format • time separator

Table 34. QAQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
	AA The sort sequence table specified is different than the sort sequence table that was used when this access plan was created.
	AB Storage pool changed or DEGREE parameter of CHGQRYA command changed.
	AC The system feature DB2 multisystem has been installed or removed.
	AD The value of the degree query attribute has changed.
	AE A view is either being opened by a high level language or a view is being materialized.
	AF A user-defined type or user-defined function is not the same object as the one referred to in the access plan, or, the SQL Path is not the same as when the access plan was built.
	B0 The options specified have changed as a result of the query options file QAQQINI.
	B1 The access plan was generated with a commitment control level that is different in the current job.
	B2 The access plan was generated with a static cursor answer set size that is different than the previous access plan.
QQDACV	Data conversion
	N No.
	0 Not applicable.
	1 Lengths do not match.
	2 Numeric types do not match.
	3 C host variable is NUL-terminated.
	4 Host variable or column is variable length and the other s not variable length.
	5 CCSID conversion.
	6 DRDA and NULL capable, variable length, contained in a partial row, derived expression, or blocked fetch with not enough host variables.
	7 Data, time, or timestamp column.
	8 Too many host variables.
	9 Target table of an insert is not an SQL table.
QQCTS	Statement table scan usage count
QQCIU	Statement index usage count
QQCIC	Statement index creation count
QQCSO	Statement sort usage count
QQCTF	Statement temporary table count
QQCIA	Statement index advised count
QQCAPR	Statement access plan rebuild count
QQARSS	Average result set size
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved

Table 34. QAQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQC1000	Reserved

External table description (QAQQTEXT) - Summary Row for SQL Statement

Table 35. QAQQTEXT - Summary Row for SQL Statement

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together with row identification
QQTIME	Time row was created
QQSTTX	Statement text
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3000) - Summary Row for Arrival Sequence

Table 36. QAQQ3000 - Summary Row for Arrival Sequence

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together with row identification
QQTIME	Time row was created
QQQDTN	QDT number (unique per ODT)
QQQDTL	QDT subquery nested level
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTNL	Library
QQTFN	Table
QQPTLN	Physical library

Table 36. QAQQ3000 - Summary Row for Arrival Sequence (continued)

Column Name	Description
QQPTFN	Physical table
QQTOTR	Total rows in table
QQREST	Estimated number of rows selected
QQAJN	Estimated number of joined rows
QQEPT	Estimated processing time, in seconds
QQJNP	Join position - when available
QQJNDS	Dataspace number
QQJNMT	Join method - when available
	NL - Nested loop
	MF - Nested loop with selection
	HJ - Hash join
QQJNTY	Join type - when available
	IN - Inner join
	PO - Left partial outer join
	EX - Exception join
QQJNOP	Join operator - when available
	EQ - Equal
	NE - Not equal
	GT - Greater than
	GE - Greater than or equal
	LT - Less than
	LE - Less than or equal
	CP - Cartesian product
QQDSS	Dataspace selection
	Y - Yes
	N - No
QQIDXA	Index advised
	Y - Yes
	N - No
QQRCOD	Reason code
	T1 - No indexes exist.
	T2 - Indexes exist, but none could be used.
	T3 - Optimizer chose table scan over available indexes.
QQLTLN	Library-long
QQLTFN	Table-long
QQLPTL	Physical library-long
QQLPTF	Table-long
QQIDXD	Key columns for the index advised
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved

Table 36. QQQ3000 - Summary Row for Arrival Sequence (continued)

Column Name	Description
QQC22	Reserved
QQI1	Number of advised key columns that use index scan-key positioning.
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QQQ3001) - Summary row for Using Existing Index

Table 37. QQQ3001 - Summary Row for Using Existing Index

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQPTLN	Physical library
QQPTFN	Physical table
QQILNM	Index library
QQIFNM	Index
QQTOTR	Total rows in table
QQREST	Estimated number of rows selected
QQFKEY	Number of key positioning keys
QQKSEL	Number of key selection keys
QQAJN	Join position - when available
QQEPT	Estimated processing time, in seconds
QQJNP	Join position - when available
QQJNDS	Dataspace number
QQJNMT	Join method - when available
	NL - Nested loop
	MF - Nested loop with selection
	HJ - Hash join
QQJNTY	Join type - when available
	IN - Inner join
	PO - Left partial outer join
	EX - Exception join

Table 37. QQQ3001 - Summary Row for Using Existing Index (continued)

Column Name	Description
QQJNOP	Join operator - when available
	EQ - Equal
	NE - Not equal
	GT - Greater than
	GE - Greater than or equal
	LT - Less than
	LE - Less than or equal
	CP - Cartesian product
QQIDXK	Number of advised key columns that use index scan-key positioning
QQKP	Index scan-key positioning
	Y - Yes
	N - No
QQKPN	Number of key positioning columns
QQKS	Index scan-key selection
	Y - Yes
	N - No
QQDSS	Dataspace selection
	Y - Yes
	N - No
QQIDXA	Index advised
	Y - Yes
	N - No
QQRCOD	Reason code
	I1 - Row selection
	I2 - Ordering/Grouping
	I3 - Row selection and Ordering/Grouping
	I4 - Nested loop join
	I5 - Row selection using bitmap processing
QQCST	Constraint indicator
	Y - Yes
	N - No
QQCSTN	Constraint name
QQLTLN	Library-long
QQLTFN	Table-long
QQLPTL	Physical library-long
QQLPTF	Table-long
QQLILN	Index library – long
QQLIFN	Index – long
QQIDXDXD	Key columns for the index advised
QQC11	Reserved

Table 37. QQQ3001 - Summary Row for Using Existing Index (continued)

Column Name	Description
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAAA3002) - Summary Row for Index Created

Table 38. QQQ3002 - Summary Row for Index Created

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQPTLN	Physical library
QQPTFN	Physical table
QQILNM	Index library
QQIFNM	Index
QQNTNM	NLSS table
QQNLNM	NLSS library
QQTOTR	Total rows in table
QQRIDX	Number of entries in index created
QQREST	Estimated number of rows selected
QQFKEY	Number of index scan-key positioning keys
QQKSEL	Number of index scan-key selection keys
QQAJN	Estimated number of joined rows
QQJNP	Join position - when available
QQJNDS	Dataspace number
QQJNMT	Join method - when available

NL - Nested loop
 MF - Nested loop with selection
 HJ - Hash join

Table 38. QQQ3002 - Summary Row for Index Created (continued)

Column Name	Description
QQJNTY	Join type - when available
	IN - Inner join PO - Left partial outer join EX - Exception join
QQJNOP	Join operator - when available
	EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product
QQIDXK	Number of advised key columns that use index scan-key positioning
QQEPT	Estimated processing time, in seconds
QQKP	Index scan-key positioning
	Y - Yes N - No
QQKPN	Number of index scan-key positioning columns
QQKS	Index scan-key selection
	Y - Yes N - No
QQDSS	Dataspace selection
	Y - Yes N - No
QQIDXA	Index advised
	Y - Yes N - No
QQCST	Constraint indicator
	Y - Yes N - No
QQCSTN	Constraint name
QQRCOD	Reason code
	I1 - Row selection I2 - Ordering/Grouping I3 - Row selection and Ordering/Grouping I4 - Nested loop join I5 - Row selection using bitmap processing
QQTIM	Index create time
QQLTLN	Library-long
QQLTFN	Table-long

Table 38. QQQ3002 - Summary Row for Index Created (continued)

Column Name	Description
QQLPTL	Physical library-long
QQLPTF	Table-long
QQLILN	Index library-long
QQLIFN	Index-long
QQLNTN	NLSS table-long
QQLNLN	NLSS library-long
QQIDXD	Key columns for the index advised
QQCRTK	Key columns for index created
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAAA3003) - Summary Row for Query Sort

Table 39. QQQ3003 - Summary Row for Query Sort

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTTIM	Sort time
QQRSS	Number of rows selected or sorted
QQSIZ	Size of sort space
QQPSIZ	Pool size
QQPID	Pool id
QQIBUF	Internal sort buffer length
QQEBUF	External sort buffer length

Table 39. QQQ3003 - Summary Row for Query Sort (continued)

Column Name	Description
QQRCD	Reason code
	F1 Query contains grouping columns (Group By) from more than one table, or contains grouping columns from a secondary table of a join query that cannot be reordered.
	F2 Query contains ordering columns (Order By) from more than one table, or contains ordering columns from a secondary table of a join query that cannot be reordered.
	F3 The grouping and ordering columns are not compatible.
	F4 DISTINCT was specified for the query.
	F5 UNION was specified for the query.
	F6 Query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 columns specified for ordering.
	F7 Query optimizer chose to use a sort rather than an index to order the results of the query.
	F8 Perform specified row selection to minimize I/O wait time.
	FC The query contains grouping fields and there is a read trigger on at least one of the physical files in the query.
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3004) - Summary Row for Temporary Table

Table 40. QQQ3004 - Summary Row for Temporary Table

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQTTIM	Temporary table create time

Table 40. QQQ3004 - Summary Row for Temporary Table (continued)

Column Name	Description
QQTMPR	Number of rows in temporary
QQRCOD	Reason code
	F1 Query contains grouping columns (Group By) from more than one table, or contains grouping columns from a secondary table of a join query that cannot be reordered.
	F2 Query contains ordering columns (Order By) from more than one table, or contains ordering columns from a secondary table of a join query that cannot be reordered.
	F3 The grouping and ordering columns are not compatible.
	F4 DISTINCT was specified for the query.
	F5 UNION was specified for the query.
	F6 Query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 columns specified for ordering.
	F7 Query optimizer chose to use a sort rather than an index to order the results of the query.
	F8 Perform specified row selection to minimize I/O wait time.
	F9 The query optimizer chose to use a hashing algorithm rather than an access path to perform the grouping for the query.
	FA The query contains a join condition that requires a temporary file.
	FB The query optimizer creates a run-time temporary file in order to implement certain correlated group by queries.
	FC The query contains grouping fields and there is a read trigger on at least one of the physical files in the query.
	FD The query optimizer creates a runtime temporary file for a static-cursor request.
	H1 Table is a join logical file and its join type does not match the join type specified in the query.
	H2 Format specified for the logical table references more than one base table.
	H3 Table is a complex SQL view requiring a temporary results of the SQL view.
	H4 For an update-capable query, a subselect references a column in this table which matches one of the columns being updated.
	H5 For an update-capable query, a subselect references an SQL view which is based on the table being updated.
	H6 For a delete-capable query, a subselect references either the table from which rows are to be deleted, an SQL view, or an index based on the table from which rows are to be deleted.
	H7 A user-defined table function was materialized.
QQDFVL	Default values may be present in temporary
	Y - Yes N - No
QQLTLN	Library-long
QQLTFN	Table-long
QQC11	Reserved

| *Table 40. QQQ3004 - Summary Row for Temporary Table (continued)*

Column Name	Description
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

| **External table description (QAAA3007) - Summary Row for Optimizer Information**

| *Table 41. QQQ3007 - Summary Row for Optimizer Information*

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQPTLN	Physical library
QQPTFN	Table
QQTOUT	Optimizer timed out
	Y - Yes N - No.
QQIRSN	Reason code
QQLTLN	Library-long
QQLTFN	Table-long
QQPTL	Physical library-long
QQPTF	Table-long
QQIDXN	Index names
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved

Table 41. QQQ3007 - Summary Row for Optimizer Information (continued)

Column Name	Description
QQC302	Reserved
QQ1000	Reserved

External table description (QAAA3008) - Summary Row for Subquery Processing

Table 42. QQQ3008 - Summary Row for Subquery Processing

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQORGQ	Materialized view QDT number
QQMRGQ	Materialized view nested level
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAAA3010) - Summary Row for Host Variable and ODP Implementation

Table 43. QQQ3010 - Summary Row for Host Variable and ODP Implementation

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQHVAR	Host variable values
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved

| *Table 43. QQQ3010 - Summary Row for Host Variable and ODP Implementation (continued)*

Column Name	Description
QQC301	Reserved
QQC302	Reserved

Index

A

- access method
 - bitmap processing access 24
 - hashing access 23
 - index only access 21
 - index scan-key positioning 15
 - index scan-key selection 13
 - index-from-index 22
 - parallel index scan-key positioning 19
 - parallel index scan-key selection access method 14
 - parallel preload 22
 - parallel table prefetch 10
 - parallel table scan 11
 - row selection method 3
 - sort access 28
 - summary table 5
 - table scan 8
- access path
 - index 3
 - sequential 3
- access plan
 - validation 34
- access plan rebuilt
 - summary row 183, 184, 185
- advisor
 - query optimizer index 72
- ALLOCATE clause
 - performance implications 131
- allow copy data (ALWCPYDTA) parameter 117
- ALWCPYDTA (allow copy data) parameter 117
- ALWCPYDTA parameter
 - effect on query optimizer 32
- Analyzing queries with the Statistics Manager 100
- APIs
 - Statistics Manager 100

B

- bitmap created
 - summary row 204, 205
- bitmap merge
 - summary row 206, 207, 208
- bitmap processing access method 24
- blocking consideration
 - using, affect on performance 125
- blocking, SQL
 - improving performance 125

C

- calls, number
 - using
 - FETCH statement 124
- cancelling a query 95
- Change Query Attribute (CHGQRYA) command 11
- Change query attributes 83
- Change Query Attributes (CHGQRYA) command 60

- changing
 - query options file 87
- CHGQRYA (Change Query Attributes) command 60
- CLOSQLCSR parameter
 - using 121
- command
 - CHGQRYA 83
 - CHGQRYA command 83
 - QAQQINI 84
 - QAQQINI command 84
- command (CL)
 - Change Query Attribute (CHGQRYA) command 11
 - Change Query Attributes (CHGQRYA) 60
 - CHGQRYA (Change Query Attribute) command 11
 - CHGQRYA (Change Query Attributes) 60
 - Delete Override (DLTOVR) 119
 - Display Job (DSPJOB) 60
 - Display Journal (DSPJRN) 120
 - DLTOVR (Delete Override) 119
 - DSPJOB (Display Job) 60
 - DSPJRN (Display Journal) 120
 - Override Database File (OVRDBF) 125
 - OVRDBF (Override Database File) 125
 - Print SQL Information (PRTSQLINF) 60, 96
 - QAQQINI 87
 - Start Database Monitor (STRDBMON) 60
 - STRDBMON (Start Database Monitor) 60
 - Trace Job (TRCJOB) 120
 - TRCJOB (Trace Job) 120
- commands
 - End Database Monitor (ENDDDBMON) 71
 - Start Database Monitor (STRDBMON) 70
- commitment control
 - displaying 60
- controlling parallel processing 97
- copy of the data
 - using to improve performance 130
- cost estimation
 - query optimizer 32
- create statistics with iSeries Navigator 100
- cursor
 - positions
 - retaining across program call 120, 121
 - rules for retaining 120
 - using to improve performance 120, 121

D

- data
 - paging
 - interactively displayed to improve performance 127
 - selecting from multiple tables
 - affect on performance 50
- data path, open 66
- database monitor
 - end 71
 - examples 73, 76

- database monitor (*continued*)
 - logical file DDS 141
 - physical file DDS 133
 - start 70
- database monitor performance rows 72
- database query performance
 - monitoring 69
- dataspace
 - definition 4
- DDS
 - database monitor logical file 141
 - database monitor physical file 133
- default filter factors 33
- definitions
 - bitmap processing method 24
 - dataspace 4
 - default filter factors 33
 - dial 35
 - hashing access method 23
 - implementation cost 32
 - index 3
 - index only access method 21
 - index scan-key positioning access method 15
 - index scan-key selection access method 13
 - index-from-index access method 22
 - isolatable 46
 - miniplan 34
 - open data path 66
 - parallel index scan-key positioning access method 19
 - parallel index scan-key selection access method 14
 - parallel table prefetch access method 10
 - parallel table scan method 11
 - primary table 35
 - secondary tables 35
 - sort access method 28
 - symmetrical multiprocessing 4
 - table scan 3
- Delete Override (DLTOVR) command 119
- deleted rows
 - getting rid of using REUSEDLT(*YES) 8
 - getting rid of using RGZPFM 8
- detail row
 - rows retrieved 202
- Display Job (DSPJOB) command 60
- Display Journal (DSPJRN) command 120
- distinct processing
 - summary row 212, 213
- DSPJOB (Display Job) command 60

E

- End Database Monitor (ENDDBMON) command 71
- ENDDBMON (end database monitor) command 71
- examples
 - database monitor 73, 76
 - governor 96
 - index 103
 - performance analysis 73, 74, 75
 - reducing the number of open database operation 118

- examples (*continued*)
 - selecting data from multiple tables 50

F

- file
 - query options 87
- filter factors, default
 - in query optimization 33

G

- generic query information
 - summary row 194, 195, 196, 197, 198, 199
- governor 93
 - *DFT 95
 - *RQD 95
 - *SYSRPYL 95
 - CHGQRYA 94
 - JOB 95
 - QRYTIMLMT 94
 - time limit 95
- grouping
 - summary row 219, 220, 221
- grouping optimization 50

H

- hash join 36
- hash table
 - summary row 210, 211
- hashing access method 23
- host variable and ODP implementation
 - summary row 190

I

- improving performance 117, 130
 - blocking, using 125
 - join queries 47
 - paging interactively displayed data 127
 - PREPARE statement 127
 - retaining cursor positions across program call 120, 121
 - SELECT statements, using effectively 126
 - selecting data from multiple tables 50
 - SQL blocking 125
 - using
 - close SQL cursor (CLOSQLCSR) 120, 121
 - FETCH FOR n ROWS 124
 - INSERT n ROWS 125
 - precompile options 129
- index
 - access path 3
 - columns used for keys 3
 - creating
 - from another index 22
 - temporary keyed
 - from index 45
 - from the table 44

- index (*continued*)
 - using effectively, example 103
- index advisor
 - query optimizer 72
- index created
 - summary row 167, 168, 169, 170
- index only access method 21
- index scan-key positioning
 - access method 15
- index scan-key selection
 - access method 13
- index-from-index
 - access method 22
- indexes
 - using with sort sequence 106
- information messages
 - open data path 66, 67
 - performance 61, 66
- INSERT n ROWS
 - improving performance 125
- interactively displayed data, paging
 - affect on performance 127
- iSeries Navigator
 - creating
 - SQL performance monitor 81
 - SQL performance monitor 80
 - analyzing data 82
 - pausing 82
 - saving data 82

J

- JOB 95
- join
 - hash 36
 - optimization 35
- join optimization
 - performance tips 47
- join order
 - optimization 41
- join position 64
- join secondary dials
 - costing 42

K

- key range estimate 34

L

- limit, time 95
- live data
 - using to improve performance 117
- locks
 - analyzing 60
- logical file DDS
 - database monitor 141
- long object names
 - performance 129

M

- manage statistical information with iSeries Navigator 100
- message
 - cause and user response 60
 - open data path information 66, 67
 - performance information 61, 66
 - running in debug mode 60
- monitor (ENDDBMON) command, end database 71
- monitoring
 - database query performance 69
- multiple
 - table
 - improving performance when selecting data from 50

N

- nested loop join 35
- number of calls
 - using a FETCH statement 124
- number of open database operations
 - improving performance by reducing 118

O

- ODP implementation and host variable
 - summary row 190
- open
 - closing 119
 - determining number 120
 - effect on performance 118
 - reducing number 118
- open data path
 - definition 66
 - information messages 66
- OPNQRYF (Open Query File) command 59
- optimization 31
 - grouping 50
 - join 35
 - join order 41
 - nested loop join 35
- OPTIMIZE FOR n ROWS clause
 - effect on query optimizer 32
- optimizer
 - operation 31
 - query index advisor 72
- optimizer timed out
 - summary row 186, 187, 188
- options, precompile
 - improving performance by using 129
- output
 - all queries that performed table scans 74, 75
 - SQL queries that performed table scans 74
- Override Database File (OVRDBF) command 125

P

- page fault 4

- paging
 - interactively displayed data 127
- parallel index scan-key positioning access method 19
- parallel index scan-key selection access method 14
- parallel preload
 - index-based 22
 - table-based 22
- parallel processing
 - controlling
 - in jobs (CHGQRYA command) 98
 - system wide (QQRYPDEGREE) value 97
- parallel table prefetch
 - access method 10
- parallel table scan
 - access method 11
- parameters, command
 - ALWCPYDTA (allow copy data) 117, 130
 - CLOSQLCSR (close SQL cursor) 120, 121
- path, open data 66
- performance 31
 - information messages 61, 66
 - monitoring 59
 - monitoring query 69
 - open data path messages 66, 67
 - OPNQRYF 59
 - optimizing 59
 - tools 59
 - using long object names 129
- performance analysis
 - example 1 73
 - example 2 74
 - example 3 75
- performance considerations 96, 118
- performance improvement
 - blocking, using 125
 - paging interactively displayed data 127
 - PREPARE statement 127
 - reducing number of open database operation 118
 - retaining cursor positions across program call 120, 121
 - SELECT statements, using effectively 126
 - selecting data from multiple tables 50
 - SQL blocking 125
 - using copy of the data 130
 - using INSERT n ROWS 125
 - using live data 117
 - using precompile options 129
- performance rows
 - database monitor 72
- physical file DDS
 - database monitor 133
- pre-fetching 8
- precompile options
 - improving performance, using 129
- precompiler command
 - default 120, 121
- precompiler parameter
 - ALWCPYDTA 117
 - CLOSQLCSR 121
- predicate
 - transitive closure 45

- Predictive Query Governor 93
- PREPARE statement
 - improving performance 127
- Print SQL Information (PRTSQLINF) 60, 96
- problems
 - join query performance 47
- program calls
 - rules for retaining cursor positions 121

Q

- QAQQINI 87
- QDT 34
- QRYTIMLMT parameter
 - CHGQRYA (Change Query Attributes) command 60
- query
 - cancelling 95
- Query Definition Template (QDT) 34
- query optimizer 31
 - cost estimation 32
 - decision-making rules 31
 - default filter factors 33
 - optimization goals 32
- query optimizer index advisor 72
- query options
 - file 87
- query options file
 - changing 87
- Query options file 84
- query performance
 - monitoring 69
- query sort
 - summary row 172, 173, 174
- query time limit 95

R

- reducing number of open database operations
 - improving performance, example 118
- Reorganize Physical File Member (RGZPFM) command
 - effect on variable-length columns 132
 - getting rid of deleted rows 8
- resource
 - optimization 31
- retaining cursor positions
 - across program call
 - improving performance 120, 121
 - all program calls
 - rules 121
- rows
 - database monitor performance 72
- rows retrieved
 - detail row 202
- ROWS, INSERT n
 - improving performance 125
- rule
 - retaining cursor positions
 - program calls 121

S

- SELECT statement
 - using effectively to improve performance 126
- selecting
 - data from multiple tables 50
- setting query time limit 96
- sort access method 28
- sort sequence
 - using indexes 106
- SQL blocking
 - improving performance 125
- SQL information
 - summary row 145, 146, 147, 148, 149, 150, 151, 152, 223, 224, 225, 226, 227, 228, 229
- SQL performance monitor 80
 - analyzing data 82
 - creating 81
 - pausing 82
 - saving data 82
- Start Database Monitor (STRDBMON) command 60, 70
- statements
 - FETCH
 - FOR n ROWS 124
 - number of calls 124
 - INSERT
 - n ROWS 125
 - PREPARE
 - improving performance 127
- statistics
 - create with iSeries Navigator 100
 - manage with iSeries Navigator 100
 - update with iSeries Navigator 101
 - view with iSeries Navigator 101
- Statistics Manager
 - analyzing queries 100
 - APIs 100
- STRDBMON (Start Database Monitor) command 60, 70
- STRDBMON/ENDDDBMON commands
 - summary row 200
- subquery merge
 - summary row 215, 216, 217
- subquery processing
 - summary row 189
- summary row
 - access plan rebuilt 183, 184, 185
 - bitmap created 204, 205
 - bitmap merge 206, 207, 208
 - distinct processing 212, 213
 - generic query information 194, 195, 196, 197, 198, 199
 - grouping 219, 220, 221
 - hash table 210, 211
 - host variable and ODP implementation 190
 - index created 167, 168, 169, 170
 - optimizer timed out 186, 187, 188
 - query sort 172, 173, 174
 - SQL information 145, 146, 147, 148, 149, 150, 151, 152
 - STRDBMON/ENDDDBMON commands 200

- summary row (*continued*)
 - subquery merge 215, 216, 217
 - subquery processing 189
 - table locked 180, 181, 182
 - table scan 154, 155, 156
 - temporary table 176, 177, 178, 179
 - using existing index 160, 161, 162, 163
- summary rows
 - SQL information 223, 224, 225, 226, 227, 228, 229
 - using existing index 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240
- symmetrical multiprocessing 4

T

- table
 - data management methods 5
 - multiple
 - improving performance when selecting data from 50
- table locked
 - summary row 180, 181, 182
- table scan 3
 - access method 8
 - summary row 154, 155, 156
- table scans
 - output for all queries 74, 75
 - output for SQL queries 74
- temporary index 44, 45
- temporary table
 - summary row 176, 177, 178, 179
- tools
 - performance 59
- Trace Job (TRCJOB) command 120
- transitive closure 45

U

- update statistics with iSeries Navigator 101
- using
 - a copy of the data 117, 130
 - allow copy data (ALWCOPYDATA) 117, 130
 - close SQL cursor (CLOSQLCSR) 117, 121
 - FETCH statement 124
- using existing index
 - summary row 160, 161, 162, 163, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240
- using JOB parameter 96
- using SQL
 - application programs 31

V

- variable-length data
 - tips 131
- view statistics with iSeries Navigator 101



Printed in U.S.A.