



@server

iSeries

DB2 Universal Database for iSeries Database  
Performance and Query Optimization

*Version 5 Release 3*







@server

iSeries

DB2 Universal Database for iSeries Database  
Performance and Query Optimization

*Version 5 Release 3*

**Note**

Before using this information and the product it supports, be sure to read the information in "Notices" on page 317.

**Fifth Edition (August 2005)**

| This edition applies to version 5, release 3, modification 0 of IBM Operating System/400® (product number  
| 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version  
| does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

© Copyright International Business Machines Corporation 1998, 2005. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Chapter 1. Query performance and query optimization . . . . .</b>	<b>1</b>
Code disclaimer . . . . .	2
<b>Chapter 2. What's new for V5R3 . . . . .</b>	<b>3</b>
<b>Chapter 3. Print this topic . . . . .</b>	<b>5</b>
<b>Chapter 4. Query Engine Overview . . . . .</b>	<b>7</b>
SQE and CQE Engines . . . . .	7
Query Dispatcher . . . . .	9
Statistics Manager . . . . .	10
Plan Cache . . . . .	11
<b>Chapter 5. Data access on DB2 UDB for iSeries: data access paths and methods . . . . .</b>	<b>13</b>
Permanent objects and access methods . . . . .	13
Table . . . . .	13
Radix Index . . . . .	16
Encoded Vector Index . . . . .	21
Temporary objects and access methods . . . . .	23
Temporary Hash Table . . . . .	23
Temporary Sorted List . . . . .	27
Temporary List . . . . .	31
Temporary Row Number List . . . . .	33
Temporary Bitmap . . . . .	37
Temporary Index . . . . .	40
Temporary Buffer . . . . .	44
Objects processed in parallel . . . . .	47
Spreading data automatically . . . . .	47
<b>Chapter 6. Processing queries: Overview . . . . .</b>	<b>49</b>
How the query optimizer makes your queries more efficient . . . . .	49
General query optimization tips . . . . .	49
Access plan validation . . . . .	50
Single table optimization . . . . .	50
Join optimization . . . . .	51
Grouping optimization . . . . .	61
Ordering optimization . . . . .	64
View implementation . . . . .	65
Materialized query table optimization . . . . .	67
<b>Chapter 7. Optimizing query performance using query optimization tools . . . . .</b>	<b>75</b>
Verify the performance of SQL applications . . . . .	75
Examine query optimizer debug messages in the job log . . . . .	76
Gather information about embedded SQL statements with the PRTSQLINF command . . . . .	77
Monitoring your queries using Start Database Monitor (STRDBMON) . . . . .	78
Start Database Monitor (STRDBMON) command . . . . .	79
End Database Monitor (ENDDDBMON) command . . . . .	79
Database monitor performance rows . . . . .	80
Using iSeries Navigator to start STRDBMON . . . . .	81
Query optimizer index advisor . . . . .	86
Database monitor examples . . . . .	87
Monitoring your queries using memory-resident database monitor . . . . .	92
Memory-resident database monitor external API description . . . . .	93
Memory-resident database monitor external table description . . . . .	93
Sample SQL queries . . . . .	94
Memory-resident database monitor row identification . . . . .	94
View the implementation of your queries with Visual Explain . . . . .	94
Starting Visual Explain . . . . .	95
Overview of information available from Visual Explain . . . . .	95
Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command . . . . .	96
Control queries dynamically with the query options file QAQQINI . . . . .	97
Specifying the QAQQINI file . . . . .	98
Creating the QAQQINI query options file . . . . .	98
QAQQINI query options . . . . .	100
Control long-running queries with the Predictive Query Governor . . . . .	107
Using the Query Governor . . . . .	108
Canceling a query with the Query Governor . . . . .	109
Query governor implementation considerations . . . . .	109
Controlling the default reply to the query governor inquiry message . . . . .	109
Testing performance with the query governor . . . . .	109
Examples of setting query time limits . . . . .	110
Control parallel processing for queries . . . . .	111
Controlling system wide parallel processing for queries . . . . .	111
Controlling job level parallel processing for queries . . . . .	112
Collecting statistics with the Statistics Manager . . . . .	113
Automatic statistics collection . . . . .	114
Automatic statistics refresh . . . . .	114
Viewing statistics requests . . . . .	115
Indexes versus column statistics . . . . .	115
Monitoring background statistics collection . . . . .	116
Replication of column statistics with CRTDUPOBJ versus CPYF . . . . .	116
Determining what column statistics exist . . . . .	116
Manually collecting and refreshing statistics . . . . .	117
Statistics Manager APIs . . . . .	118
Query optimization tools: Comparison table . . . . .	119

<b>Chapter 8. Creating an index strategy</b>	<b>121</b>
Index basics . . . . .	121
Binary radix indexes . . . . .	121
Encoded vector indexes . . . . .	122
Comparing Binary radix indexes and Encoded      vector indexes . . . . .	125
Indexes and the optimizer . . . . .	125
Instances where an index is not used . . . . .	126
Determining unnecessary indexes . . . . .	127
Indexing strategy . . . . .	127
Reactive approach to tuning . . . . .	128
Proactive approach to tuning . . . . .	128
Coding for effective indexes . . . . .	129
Avoid numeric conversions . . . . .	129
Avoid arithmetic expressions . . . . .	130
Avoid character string padding . . . . .	130
Avoid the use of like patterns beginning with %      or _ . . . . .	130
Using indexes with sort sequence . . . . .	131
Using indexes and sort sequence with selection,      joins, or grouping . . . . .	131
Using indexes and sort sequence with ordering . . . . .	131
Examples of indexes . . . . .	132
Index example: Equals selection with no sort      sequence table . . . . .	132
Index example: Equals selection with a      unique-weight sort sequence table . . . . .	133
Index example: Equal selection with a      shared-weight sort sequence table . . . . .	133
Index example: Greater than selection with a      unique-weight sort sequence table . . . . .	133
Index example: Join selection with a      unique-weight sort sequence table . . . . .	133
Index example: Join selection with a      shared-weight sort sequence table . . . . .	134
Index example: Ordering with no sort sequence      table . . . . .	134
Index example: Ordering with a unique-weight      sort sequence table . . . . .	134
Index example: Ordering with a shared-weight      sort sequence table . . . . .	135
Index example: Ordering with      ALWCPYDTA(*OPTIMIZE) and a      unique-weight sort sequence table . . . . .	135
Index example: Grouping with no sort sequence      table . . . . .	135
Index example: Grouping with a unique-weight      sort sequence table . . . . .	135
Index example: Grouping with a shared-weight      sort sequence table . . . . .	136
Index example: Ordering and grouping on the      same columns with a unique-weight sort      sequence table . . . . .	136
Index example: Ordering and grouping on the      same columns with ALWCPYDTA(*OPTIMIZE)      and a unique-weight sort sequence table . . . . .	136
Index example: Ordering and grouping on the      same columns with a shared-weight sort      sequence table . . . . .	137

Index example: Ordering and grouping on the same columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table . . . . .	137
Index example: Ordering and grouping on different columns with a unique-weight sort sequence table . . . . .	137
Index example: Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table . . . . .	138
Index example: Ordering and grouping on different columns with ALWCPYDTA(*OPTIMIZE) and a shared-weight sort sequence table . . . . .	138

**Chapter 9. Application design tips for database performance . . . . . 139**

Use live data . . . . .	139
Reduce the number of open operations . . . . .	140
Retain cursor positions . . . . .	142
Retaining cursor positions for non-ILE program calls . . . . .	142
Retaining cursor positions across ILE program calls . . . . .	143
General rules for retaining cursor positions for all program calls . . . . .	143

**Chapter 10. Programming techniques for database performance . . . . . 145**

Use the OPTIMIZE clause . . . . .	145
Use FETCH FOR n ROWS . . . . .	146
Improve SQL blocking performance when using FETCH FOR n ROWS . . . . .	147
Use INSERT n ROWS . . . . .	147
Control database manager blocking . . . . .	147
Optimize the number of columns that are selected with SELECT statements . . . . .	148
Eliminate redundant validation with SQL PREPARE statements . . . . .	149
Page interactively displayed data with REFRESH(*FORWARD) . . . . .	149

**Chapter 11. General DB2 UDB for iSeries performance considerations. . 151**

Effects on database performance when using long object names . . . . .	151
Effects of precompile options on database performance. . . . .	151
Effects of the ALWCPYDTA parameter on database performance. . . . .	152
Tips for using VARCHAR and VARGRAPHIC data types in databases. . . . .	153

**Chapter 12. Database Monitor DDS . . . . . 155**

Database monitor: DDS . . . . .	155
Database monitor physical file DDS . . . . .	155
Optional database monitor logical file DDS . . . . .	162
Memory Resident Database Monitor: DDS. . . . .	252

External table description (QAQQQRYI) - Summary Row for SQL Information . . . . .	253
External table description (QAQQTEXT) - Summary Row for SQL Statement . . . . .	259
External table description (QAQQ3000) - Summary Row for Arrival Sequence. . . . .	259
External table description (QAQQ3001) - Summary row for Using Existing Index . . . . .	261
External table description (QAQQ3002) - Summary Row for Index Created. . . . .	263
External table description (QAQQ3003) - Summary Row for Query Sort. . . . .	265
External table description (QAQQ3004) - Summary Row for Temporary Table. . . . .	266
External table description (QAQQ3007) - Summary Row for Optimizer Information. . . . .	268
External table description (QAQQ3008) - Summary Row for Subquery Processing . . . . .	269
External table description (QAQQ3010) - Summary Row for Host Variable and ODP Implementation . . . . .	269

**Chapter 13. Query optimizer messages reference . . . . . 271**

Query optimization performance information messages . . . . .	271
CPI4321 - Access path built for &18 &19 . . . . .	272
CPI4322 - Access path built from keyed file &1 . . . . .	273
CPI4323 - The OS/400 query access plan has been rebuilt . . . . .	274
CPI4324 - Temporary file built for file &1 . . . . .	276
CPI4325 - Temporary result file built for query . . . . .	277
CPI4326 - &12 &13 processed in join position &10 . . . . .	277
CPI4327 - File &12 &13 processed in join position &10. . . . .	278
CPI4328 - Access path of file &3 was used by query . . . . .	279
CPI4329 - Arrival sequence access was used for &12 &13 . . . . .	279
CPI432A - Query optimizer timed out for file &1 . . . . .	280
CPI432B - Subselects processed as join query . . . . .	282
CPI432C - All access paths were considered for file &1. . . . .	282
CPI432D - Additional access path reason codes were used . . . . .	283
CPI432F - Access path suggestion for file &1 . . . . .	284
CPI4330 - &6 tasks used for parallel &10 scan of file &1. . . . .	284
CPI4331 - &6 tasks used for parallel index created over file . . . . .	285
CPI4332 - &1 host variables used in query. . . . .	286
CPI4333 - Hashing algorithm used to process join. . . . .	287
CPI4334 - Query implemented as reusable ODP . . . . .	287
CPI4335 - Optimizer debug messages for hash join step &1 foil . . . . .	287
CPI4336 - Group processing generated . . . . .	288
CPI4337 - Temporary hash table build for hash join step &1 . . . . .	288

CPI4338 - &1 Access path(s) used for bitmap processing of file &2 . . . . .	288
CPI433D - Query options used to build the OS/400 query access plan . . . . .	289
CPI433F - Multiple join classes used to process join. . . . .	289
CPI4340 - Optimizer debug messages for join class step &1 foil . . . . .	289
CPI4341 - Performing distributed query . . . . .	290
CPI4342 - Performing distributed join for query . . . . .	290
CPI4343 - Optimizer debug messages for distributed query step &1 of &2 follow: . . . . .	290
CPI4345 - Temporary distributed result file &3 built for query . . . . .	290
CPI4346 - Optimizer debug messages for query join step &1 of &2 follow: . . . . .	291
CPI4347 - Query being processed in multiple steps . . . . .	291
CPI4348 - The ODP associated with the cursor was hard closed . . . . .	292
CPI4349 - Fast past refresh of the host variables values is not possible. . . . .	292
CPI434C - The OS/400 Query access plan was not rebuilt . . . . .	293
Query optimization performance information messages and open data paths . . . . .	293
SQL7910 - All SQL cursors closed . . . . .	294
SQL7911 - ODP reused . . . . .	295
SQL7912 - ODP created . . . . .	296
SQL7913 - ODP deleted . . . . .	296
SQL7914 - ODP not deleted . . . . .	296
SQL7915 - Access plan for SQL statement has been built . . . . .	297
SQL7916 - Blocking used for query . . . . .	297
SQL7917 - Access plan not updated . . . . .	297
SQL7918 - Reusable ODP deleted. . . . .	298
SQL7919 - Data conversion required on FETCH or embedded SELECT . . . . .	298
SQL7939 - Data conversion required on INSERT or UPDATE . . . . .	299
PRTSQLINF message reference . . . . .	300
SQL400A - Temporary distributed result file &1 was created to contain join result. . . . .	301
SQL400B - Temporary distributed result file &1 was created to contain join result. . . . .	302
SQL400C - Optimizer debug messages for distributed query step &1 and &2 follow . . . . .	302
SQL400D - GROUP BY processing generated . . . . .	302
SQL400E - Temporary distributed result file &1 was created while processing distributed subquery. . . . .	302
SQL4001 - Temporary result created . . . . .	303
SQL4002 - Reusable ODP sort used . . . . .	303
SQL4003 - UNION . . . . .	303
SQL4004 - SUBQUERY . . . . .	304
SQL4005 - Query optimizer timed out for table &1 . . . . .	304
SQL4006 - All indexes considered for table &1 . . . . .	304
SQL4007 - Query implementation for join position &1 table &2 . . . . .	304
SQL4008 - Index &1 used for table &2 . . . . .	305

SQL4009 - Index created for table &1 . . . . .	305	SQL4021 - Access plan last saved on &1 at &2	311
SQL401A - Processing grouping criteria for		SQL4022 - Access plan was saved with SRVQRY	
query containing a distributed table . . . . .	305	attributes active . . . . .	312
SQL401B - Temporary distributed result table		SQL4023 - Parallel table prefetch used . . . . .	312
&1 was created while processing grouping		SQL4024 - Parallel index preload access method	
criteria . . . . .	306	used . . . . .	312
SQL401C - Performing distributed join for query	306	SQL4025 - Parallel table preload access method	
SQL401D - Temporary distributed result table		used . . . . .	312
&1 was created because table &2 was directed	306	SQL4026 - Index only access used on table	
SQL401E - Temporary distributed result table		number &1 . . . . .	313
&1 was created because table &2 was broadcast.	307	SQL4027 - Access plan was saved with DB2	
SQL401F - Table &1 used in distributed join . . . . .	307	UDB Symmetric Multiprocessing installed on	
SQL4010 - Table scan access for table &1 . . . . .	307	the system . . . . .	313
SQL4011 - Index scan-key row positioning used		SQL4028 - The query contains a distributed	
on table &1 . . . . .	307	table . . . . .	313
SQL4012 - Index created from index &1 for table		SQL4029 - Hashing algorithm used to process	
&2 . . . . .	308	the grouping . . . . .	314
SQL4013 - Access plan has not been built . . . . .	308	SQL4030 - &1 tasks specified for parallel scan	
SQL4014 - &1 join column pair(s) are used for		on table &2. . . . .	314
this join position . . . . .	308	SQL4031 - &1 tasks specified for parallel index	
SQL4015 - From-column &1.&2, to-column		create over table &2 . . . . .	314
&3.&4, join operator &5, join predicate &6. . . . .	309	SQL4032 - Index &1 used for bitmap processing	
SQL4016 - Subselects processed as join query	309	of table &2 . . . . .	315
SQL4017 - Host variables implemented as		SQL4033 - &1 tasks specified for parallel bitmap	
reusable ODP . . . . .	309	create using &2. . . . .	315
SQL4018 - Host variables implemented as		SQL4034 - Multiple join classes used to process	
non-reusable ODP. . . . .	310	join. . . . .	315
SQL4019 - Host variables implemented as file		SQL4035 - Table &1 used in join class &2 . . . . .	316
management row positioning reusable ODP . . . . .	310		
SQL402A - Hashing algorithm used to process		<b>Notices . . . . .</b>	<b>317</b>
join. . . . .	310	Programming Interface Information . . . . .	319
SQL402B - Table &1 used in hash join step &2	310	Trademarks . . . . .	319
SQL402C - Temporary table created for hash join		Terms and conditions for downloading and	
results . . . . .	311	printing information . . . . .	319
SQL402D - Query attributes overridden from			
query options file &2 in library &1 . . . . .	311	<b>Index . . . . .</b>	<b>321</b>
SQL4020 - Estimated query run time is &1			
seconds . . . . .	311		



---

## Chapter 1. Query performance and query optimization

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 this topic, you will find the following information

**Chapter 2, "What's new for V5R3," on page 3**

This describes the new topics in V5R3

**Chapter 3, "Print this topic," on page 5**

This describes how to print

**Chapter 4, "Query Engine Overview," on page 7**

This describes query engine overview

**Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13**

Find out how the server determines the most efficient access method and what factors determine their selection by the server.

**Chapter 6, "Processing queries: Overview," on page 49**

Describes how to design queries that leverage the query optimizer's cost estimation and decision-making rules.

**Chapter 7, "Optimizing query performance using query optimization tools," on page 75**

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.

**Chapter 8, "Creating an index strategy," on page 121**

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.

**Chapter 9, "Application design tips for database performance," on page 139**

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.

**Chapter 10, "Programming techniques for database performance," on page 145**

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.

**Chapter 11, “General DB2 UDB for iSeries performance considerations,” on page 151**


Describes some general server considerations and how they affect the performance of your queries.

**Chapter 12, “Database Monitor DDS,” on page 155**

Reference information about database monitor DDS

**Chapter 13, “Query optimizer messages reference,” on page 271**

Reference information about query optimizer messages

You can also find more information about the V5R2 query engine in the  [Preparing for and Tuning the V5R2 SQL Query Engine on DB2 Universal Database™ for iSeries™](#).

**Notes:**

1. Many of the examples within this publication illustrate a query written through either an SQL or an OPNQRYF query interface. The interface chosen for a particular example does not indicate an operation exclusive to that query interface, unless explicitly noted. It is only an illustration of one possible query interface. Most examples can be easily rewritten into whatever query interface that you prefer.
2. Read the “Code disclaimer” for important legal information.

---

## **Code disclaimer**

This document contains programming examples.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM®, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

---

## Chapter 2. What's new for V5R3

The following information was added or updated in this release of the information:

New query engine information, see Chapter 4, "Query Engine Overview," on page 7 for details.

New data access methods, see Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for details.

"Look Ahead Predicate Generation" on page 57

New Visual Explain enhancements-see "View the implementation of your queries with Visual Explain" on page 94 for details.

Enhanced statistics information, see "Collecting statistics with the Statistics Manager" on page 113.



---


## Chapter 3. Print this topic

To view or download the PDF version of this document, select Database Performance and Query Optimization(about 3040KB).


### Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF in your browser (right-click the link above).
2. Click **Save Target As...** if you are using Internet Explorer. Click **Save Link As...** if you are using Netscape Communicator.
3. Navigate to the directory in which you would like to save the PDF.
4. Click **Save**.

You can also find more information about the V5R2 query engine in the  [Preparing for and Tuning the V5R2 SQL Query Engine on DB2 Universal Database for iSeries](#).

### Downloading Adobe Acrobat Reader

You need Adobe Acrobat Reader to view or print these PDFs. You can download a copy from the Adobe Web site ([www.adobe.com/products/acrobat/readstep.html](http://www.adobe.com/products/acrobat/readstep.html))  .



---

## Chapter 4. Query Engine Overview

DB2<sup>®</sup> UDB for iSeries provides two query engines to process queries: the Classic Query Engine (CQE) and the SQL Query Engine (SQE). The CQE processes queries originating from non-SQL interfaces: OPNQRYF, Query/400, and QQQQry API. SQL based interfaces, such as ODBC, JDBC, CLI, Query Manager, Net.Data<sup>®</sup>, RUNSQLSTM, and embedded or interactive SQL, run through the SQE. For ease of use, the routing decision for processing the query by either CQE or SQE is pervasive and under the control of the system. The requesting user or application program cannot control or influence this behavior. However, a better understanding of the engines and of the process that determines which path a query takes can lead you to a better understand of your query's performance. For a comparison of the two engines, see "SQE and CQE Engines."

Along with the new query engine, several more components were created and other existing components were updated. These components are:

- "Query Dispatcher" on page 9
- "Statistics Manager" on page 10
- "Plan Cache" on page 11

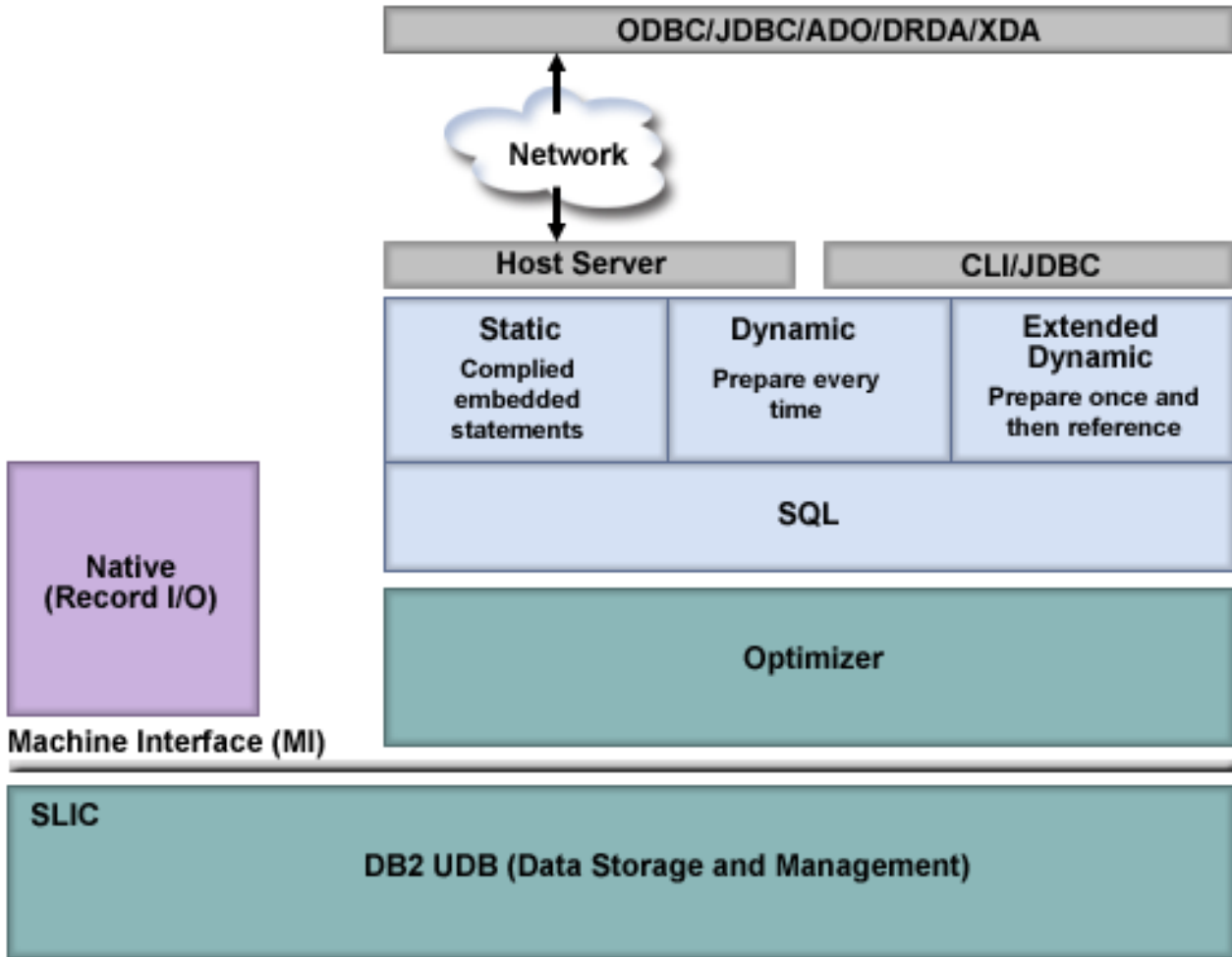
Additionally, new data access methods were created for SQE. These are discussed in Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13.

---

### SQE and CQE Engines

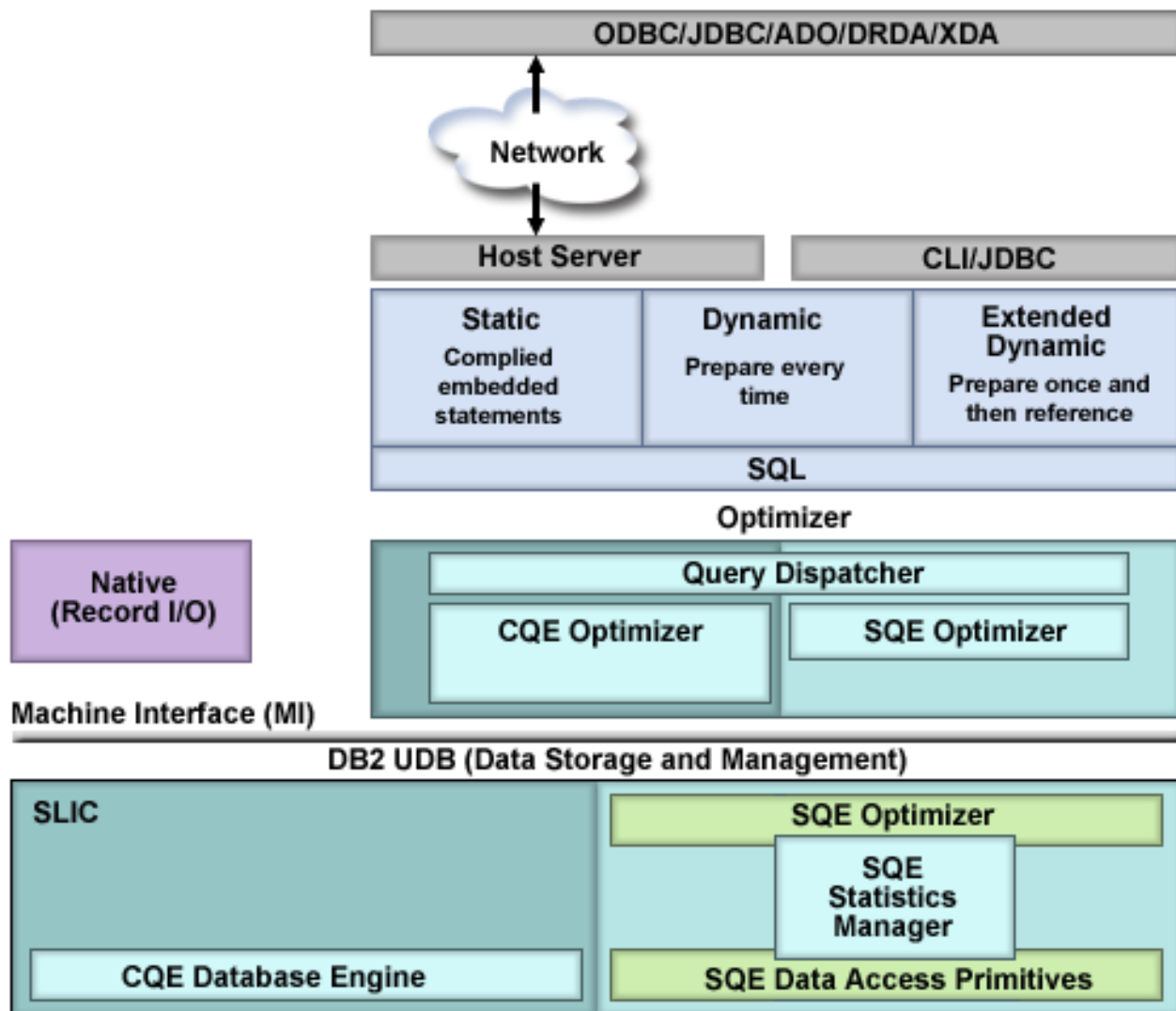
To fully understand the implementation of query management and processing in DB2 UDB for iSeries on OS/400<sup>®</sup> V5R2 and subsequent releases, it is important to see how the queries were implemented in releases of OS/400 previous to V5R2.

The figure below shows a high-level overview of the architecture of DB2 UDB for iSeries before OS/400 V5R2. The optimizer and database engine are implemented at different layers of the operating system. The interaction between the optimizer and the database engine occurs across the Machine Interface (MI).



The figure below shows an overview of the DB2 UDB for iSeries architecture on OS/400 V5R3 and where each SQE component fits. The functional separation of each SQE component is clearly evident. In line with design objectives, this division of responsibility enables IBM to more easily deliver functional enhancements to the individual components of SQE, as and when required. Notice that most of the SQE Optimizer components are implemented below the MI. This translates into enhanced performance efficiency.





## Query Dispatcher

The function of the Dispatcher is to route the query request to either CQE or SQE, depending on the attributes of the query. All queries are processed by the Dispatcher and you cannot bypass it.

Currently, the Dispatcher will route an SQL statement to CQE if it finds that the statement references or contains any of the following:

- INSERT WITH VALUES statement or the target of an INSERT with subselect statement
- Lateral correlation
- LIKE predicates
- Logical files
- LOB columns
- NLSS or CCSID translation between columns
- DB2 Multisystem tables
- non-SQL queries, for example the QQQQry API, Query/400, or OPNQRYF

| The Dispatcher also has the built-in capability to re-route an SQL query to CQE that was initially routed  
| to SQE. Unless the IGNORE\_DERIVED\_INDEX option with a parameter value of \*YES is specified, a  
| query will typically be reverted back to CQE from SQE whenever the Optimizer processes table objects  
| that have any of the following logical files or indexes defined:

- | • Logical files with the SELECT/OMIT DDS keyword specified
- | • Non-standard indexes or derived keys, for example logical files specifying the DDS keywords  
| RENAME or Alternate Collating Sequence (ACS) on any field referenced in the key
- | • Sort Sequence NLSS specified for the index or logical file

| As new functionality is added in the future, the Dispatcher will route more queries to SQE and  
| increasingly fewer to CQE.

---

## | **Statistics Manager**

| In releases before V5R2, the retrieval of statistics was a function of the Optimizer. When the Optimizer  
| needed to know information about a table, it looked at the table description to retrieve the row count and  
| table size. If an index was available, the Optimizer might then extract further information about the data  
| in the table. In V5R2, the collection of statistics was removed from the Optimizer and is now handled by  
| a separate component called the Statistics Manager.

| The Statistics Manager does not actually run or optimize the query. It controls the access to the metadata  
| and other information that is required to optimize the query. It uses this information to answer questions  
| posed by the query optimizer. The Statistics Manager always provides answers to the optimizer. In cases  
| where it cannot provide an answer based on actual existing statistics information, it is designed to  
| provide a predefined answer.

| The Statistics Manager typically gathers and keeps track of the following information:

### | **Cardinality of values**

| This is the number of unique or distinct occurrences of a specific value in a single column or  
| multiple columns of a table

### | **Selectivity**

| Also known as a histogram, this information is an indication of how many rows will be selected by  
| any given selection predicate or combination of predicates. Using sampling techniques, it describes  
| the selectivity and distribution of values in a given column of the table.

### | **Frequent values**

| This is the top *n* most frequent values of a column together with account of how frequently each  
| value occurs. This information is obtained by making use of statistical sampling techniques. Built-in  
| algorithms eliminate the possibility of data skewing; for example, NULL values and default values  
| that can influence the statistical values are not taken into account.

### | **Metadata information**

| This includes the total number of rows in the table, indexes that exist over the table, and which  
| indexes are useful for implementing the particular query.

### | **Estimate of IO operation**

| This is an estimate of the amount of IO operations that are required to process the table or the  
| identified index.

| The Statistics Manager uses a hybrid approach to manage database statistics. The majority of this  
| information can be obtained from existing indexes. In cases where the required statistics cannot be  
| gathered from existing indexes, statistical information is constructed of single columns of a table and  
| stored internally as part of the table. By default, this information is collected automatically by the system,

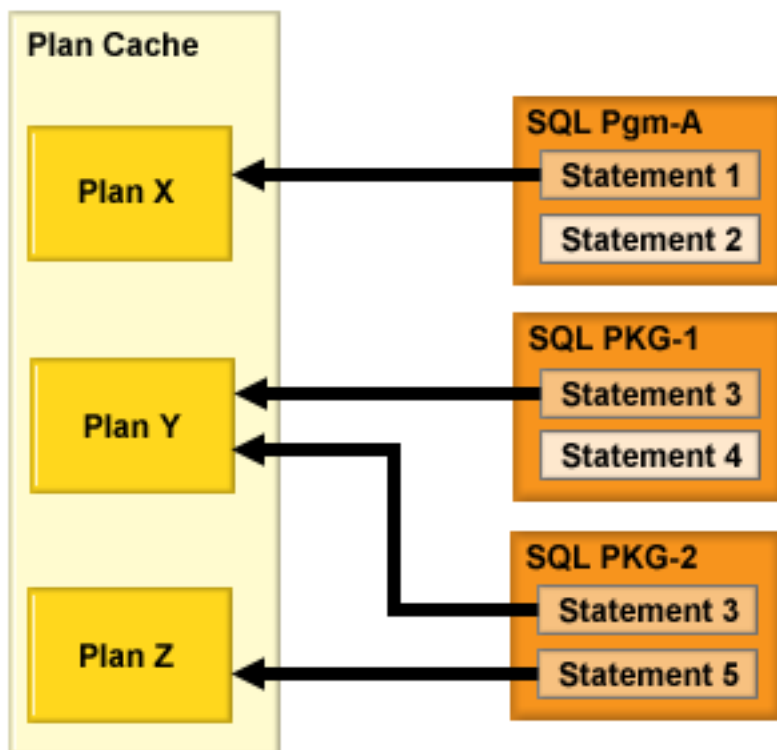
| but you can manually control the collection of statistics. Unlike indexes, however, statistics are not  
| maintained immediately as data in the tables change. For more information about statistics and the  
| Statistics Manager, see “Collecting statistics with the Statistics Manager” on page 113.

---

## | Plan Cache

| The Plan Cache is a repository that contains the access plans for queries that were optimized by the SQE.  
| The purpose of the Plan Cache is to facilitate the reuse of a query access plan at some future stage when  
| the same query, or a similar query, is executed. Once an access plan is created, it is available for use by  
| all users and all queries, regardless of where the query originates. Furthermore, when an access plan is  
| tuned, when creating an index for example, all queries can benefit from this updated access plan. This  
| eliminates the need to reoptimize the query, resulting in greater efficiency faster processing time.

| The graphic below shows the concept of reusability of the query access plans stored in the Plan Cache:



| As shown above, the Plan Cache is interrogated each time a query is executed in order to determine if a  
| valid access plan exists that satisfies the requirements of the query. If a valid access plan is found, it is  
| used to implement the query. Otherwise a new access plan is created and stored in the Plan Cache for  
| future use. The Plan Cache is automatically updated when new query access plans are created, or when  
| new statistics or indexes become available. However, access plans generated by CQE are not stored in the  
| Plan Cache; instead, they are stored in SQL Packages, the system-wide statement cache, and job cache.

| To illustrate this concept, assume that Statement 2 in the above diagram is executed for the first time by  
| SQE. The access plan for Statement 2 is stored in the Plan Cache. Statement 4 is issued by CQE. It is not  
| stored in the Plan Cache. It can, however, be stored in the SQL Package.

| The Plan Cache works in close conjunction with the system-wide statement cache, as well as SQL  
| Programs, SQL packages, and service programs. It is created with an overall size of 256 milling bytes and

| occupying approximately 250 Megabytes (MB). The Plan Cache contains the original query as well as the  
| optimized query access plan. Other objects stored with the query include the generated query runtime  
| object, as well as the query runtime information, which in turn stores the access plan usage information.  
| All systems are currently configured with the same size Plan Cache, regardless of the server size or the  
| hardware configuration.

| When the Plan Cache exceeds its designated size, a background task is automatically scheduled to  
| remove old access plans from the Plan Cache. Access plans are deleted based upon the age of the access  
| plan and how frequently it is being used. The total number of access plans stored in the Plan Cache  
| depends largely upon the complexity of the SQL statements that are being executed. In certain test  
| environments, there have been typically around 6,000 unique access plans stored in the Plan Cache. The  
| Plan Cache is cleared when a system Initial Program Load (IPL) is performed.

| Multiple access plans can be maintained for a single SQL statement. Although the SQL statement itself is  
| the primary hash key to the Plan Cache, different environmental settings can cause different access plans  
| to be stored in the Plan Cache. Examples of these environmental settings include:

- | • Different SMP Degree settings for the same query
- | • Different library lists specified for the query tables
- | • Different settings for the job's share of available memory in the current pool
- | • Different ALWCPYDTA settings

| Currently, the Plan Cache can maintain a maximum of 3 different access plans for the same SQL  
| statement. As new access plans are created for the same SQL statement, old and less frequently used  
| access plans are discarded to make room for the new access plans.

| There are, however, certain conditions that can cause an existing access plan to be invalidated. Examples  
| of these include:

- | • Specifying REOPTIMIZE\_ACCESS\_PLAN(\*YES) or (\*FORCE) in the QAQQINI table or in the SQL  
| Script
- | • Deleting or recreating the table that the access plan refers to
- | • Deleting an index that is used by the access plan

---

## Chapter 5. 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. In general, the query engine has two kinds of raw material with which to satisfy a query request:

- The database objects that contain the data to be queried
- The executable instructions or operations to retrieve and transform the data into usable information

There are actually only two types of permanent database objects that can be used as source material for a query — tables and indexes (binary radix and encoded vector indexes). In addition, the query engine may need to create temporary objects or data structures to hold interim results or references during the execution of an access plan. The DB2 UDB Symmetric Multiprocessing feature provides the optimizer with additional methods for retrieving data that include parallel processing. Finally, the optimizer uses certain methods to manipulate these objects. You can find more information about these objects and methods in the following topics:

- “Permanent objects and access methods”
- “Temporary objects and access methods” on page 23
- “Objects processed in parallel” on page 47
- “Spreading data automatically” on page 47

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

---

### Permanent objects and access methods

The database objects and access methods used by the query engine can be broken down into three basic types of operations that are used to manipulate the permanent and temporary objects -- Create, Scan, and Probe. The following table lists each object and the access methods that can be performed against that object. The symbols shown in the table are the icons used by Visual Explain. You can find more information about Visual Explain in “View the implementation of your queries with Visual Explain” on page 94.

*Table 1. Permanent Object's Data Access Methods*

Permanent Objects	Scan Operations	Probe Operations
“Table”	“Table Scan” on page 14	“Table Probe” on page 15
“Radix Index” on page 16	“Radix Index Scan” on page 17	“Radix Index Probe” on page 18
“Encoded Vector Index” on page 21	N/A	“Encoded Vector Index Probe” on page 22

### Table

An SQL table or physical file is the base object for a query. It represents the source of the data used to produce the result set for the query. It is created by the user and specified in the FROM clause (or OPNQRYF FILE parameter). The optimizer will determine the most efficient way to extract the data from the table in order to satisfy the query. This may include scanning or probing the table or using an index to extract the data. You can find more information about creating SQL tables in the SQL programming information.

Visual explain icon:



## Table Scan

A table scan is the easiest and simplest operation that can be performed against a table. It sequentially processes all of the rows in the table to determine if they satisfy the selection criteria specified in the query. It does this in a way to maximize the I/O throughput for the table. A table scan operation requests large I/Os to bring as many rows as possible into main memory for processing. It also asynchronously pre-fetches the data to make sure that the table scan operation is never waiting for rows to be paged into memory. Table scan however, has a disadvantage in it has to process all of the rows in order to satisfy the query. The scan operation itself is very efficient if it does not need to perform the I/O synchronously.


Table 2. Table Scan Attributes

Data Access Method	Table Scan
Description	Reads all of the rows from the table and applies the selection criteria to each of the rows within the table. The rows in the table are processed in no guaranteed order, but typically they are processed sequentially.
Advantages	<ul style="list-style-type: none"> <li>Minimizes page I/O operations through asynchronous pre-fetching of the rows since the pages are scanned sequentially</li> <li>Requests a larger I/O to fetch the data efficiently</li> </ul>
Considerations	<ul style="list-style-type: none"> <li>All rows in the table are examined regardless of the selectivity of the query</li> <li>Rows marked as deleted are still paged into memory even though none will be selected. You can reorganize the table to remove deleted rows.</li> </ul>
Likely to be used	<ul style="list-style-type: none"> <li>When expecting a large number of rows returned from the table</li> <li>When the number of large I/Os needed to scan is fewer than the number of small I/Os required to probe the table</li> </ul>
Example SQL statement	<pre>SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01'AND 'E01' OPTIMIZE FOR ALL ROWS</pre>
Messages indicating use	<ul style="list-style-type: none"> <li>Optimizer Debug: CPI4239 – Arrival sequence was used for file EMPLOYEE</li> <li>PRTSQLINF: SQL4010 – Table scan access for table 1.</li> </ul>
SMP parallel enabled	Yes
Also referred to as	Table Scan, Preload
Visual Explain icon	

## | **Table Probe**

| A table probe operation is used to retrieve a specific row from a table based upon its row number. The  
| row number is provided to the table probe access method by some other operation that generates a row  
| number for the table. This can include index operations as well as temporary row number lists or  
| bitmaps. The processing for a table probe is typically random; it requests a small I/O to only retrieve the  
| row in question and does not attempt to bring in any extraneous rows. This leads to very efficient  
| processing for smaller result sets because only the rows needed to satisfy the query are processed rather  
| than the scan method which must process all of the rows. However, since the sequence of the row  
| numbers are not known in advance, very little pre-fetching can be performed to bring the data into main  
| memory. This can result in most of the I/Os associated with this access method to be performed  
| synchronously.

Table 3. Table Probe Attributes

Data Access Method	Table Probe
Description	Reads a single row from the table based upon a specific row number. A random I/O is performed against the table to extract the row.
Advantages	<ul style="list-style-type: none"> <li>• Requests smaller I/Os to prevent paging rows into memory that are not needed</li> <li>• Can be used in conjunction with any access method that generates a row number for the table probe to process</li> </ul>
Considerations	Because of the synchronous random I/O the probe can perform poorly when a large number of rows are selected
Likely to be used	<ul style="list-style-type: none"> <li>• When row numbers (either from indexes or temporary row number lists) are being used, but data from the underlying table rows are required for further processing of the query</li> <li>• When processing any remaining selection or projection of the values</li> </ul>
Example SQL statement	<pre>CREATE INDEX X1 ON Employee (LastName)  SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01' AND 'E01' AND LastName IN ('Smith', 'Jones', 'Peterson') OPTIMIZE FOR ALL ROWS</pre>
Messages indicating use	<p>There is no specific message that indicates the use of a table probe. The messages in this example illustrate the use of a data access method that generates a row number that is used to perform the table probe operation.</p> <ul style="list-style-type: none"> <li>• Optimizer Debug: CPI4328 – Access path of file X1 was used by query</li> <li>• PRSQLINE: SQL4008 – Index X1 used for table 1. SQL4011 – Index scan-key row positioning (probe) used on table 1.</li> </ul>
SMP parallel enabled	Yes
Also referred to as	Table Probe, Preload
Visual Explain icon	

## Radix Index

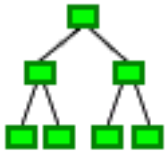
An SQL index (or keyed sequence access path) is a permanent object that is created over a table and used by the optimizer to provide a sequenced view of the data for a scan or probe operation. The rows in the tables are sequenced in the index based upon the key columns specified on the creation of the object.

When the key columns are matched up by the optimizer to a query, it gives the optimizer the ability to use the radix index to help satisfy any selection, ordering, grouping or join requirements. You can find more information about creating SQL indexes in with the CREATE INDEX.



| Typically the use of an index operation will also include a Table Probe operation to provide access to any  
| columns needed to satisfy the query that cannot be found as index keys. If all of the columns necessary  
| to satisfy the query request for a table can be found as keys of an index, then the Table Probe is not  
| required and the query uses Index Only Access. Avoiding the Table Probe can be an important savings  
| for a query. The I/O associated with a Table Probe is typically the more expensive synchronous random  
| I/O.

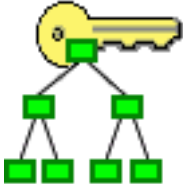
| Visual Explain icon:



### | **Radix Index Scan**

| A radix index scan operation is used to retrieve the rows from a table in a keyed sequence. Like a Table  
| Scan, all of the rows in the index will be sequentially processed, but the resulting row numbers will be  
| sequenced based upon the key columns. The sequenced rows can be used by the optimizer to satisfy a  
| portion of the query request (such as ordering or grouping). They can be also used to provide faster  
| throughput by performing selection against the index keys rather than all the rows in the table. Since the  
| I/Os associated with the index will only contain the index keys, typically more rows can be paged into  
| memory in one I/O against the index than the rows from a table with a large number of columns.

Table 4. Radix Index Scan Attributes

Data Access Method	Radix Index Scan
Description	Sequentially scan and process all of the keys associated with the index. Any selection is applied to every key value of the index before a table row
Advantages	<ul style="list-style-type: none"> <li>• Only those index entries that match any selection continue to be processed</li> <li>• Potential to extract all of the data from the index keys' values, thus eliminating the need for a Table Probe</li> <li>• Returns the rows back in a sequence based upon the keys of the index</li> </ul>
Considerations	Generally requires a Table Probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when a large number of rows are selected because of the random I/O associated with the Table Probe.
Likely to be used	<ul style="list-style-type: none"> <li>• When asking for or expecting only a few rows to be returned from the index</li> <li>• When sequencing the rows is required for the query (for example, ordering or grouping)</li> <li>• When the selection columns cannot be matched against the leading key columns of the index</li> </ul>
Example SQL statement	<pre>CREATE INDEX X1 ON Employee (LastName, WorkDept)  SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01' AND 'E01' ORDER BY LastName OPTIMIZE FOR 30 ROWS</pre>
Messages indicating use	<ul style="list-style-type: none"> <li>• Optimizer Debug: CPI4328 -- Access path of file X1 was used by query.</li> <li>• PRTSQLINF: SQL4008 -- Index X1 used for table 1.</li> </ul>
SMP parallel enabled	Yes
Also referred to as	Index Scan  Index Scan, Preload  Index Scan, Distinct  Index Scan Distinct, Preload  Index Scan, Key Selection
Visual Explain icon	

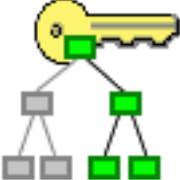
## Radix Index Probe

A radix index probe operation is used to retrieve the rows from a table in a keyed sequence. The main difference between the Radix Index Probe and the Radix Index Scan is that the rows being returned must

| first be identified by a probe operation to subset the rows being retrieved. The optimizer attempts to  
| match the columns used for some or all of the selection against the leading keys of the index. It then  
| rewrites the selection into a series of ranges that can be used to probe directly into the index's key values.  
| Only those keys from the series of ranges are paged into main memory. The resulting row numbers  
| generated by the probe operation can then be further processed by any remaining selection against the  
| index keys or a Table Probe operation. This provides for very quick access to only the rows of the index  
| that satisfy the selection.

| While the main function of a radix index probe is to provide a form of quick selection against the index  
| keys, the sequencing of the rows can still be used by the optimizer to satisfy other portions of the query  
| (such as ordering or grouping). Since the I/Os associated with the index will only be for those index  
| rows that match the selection, no extraneous processing will be performed on those rows that do not  
| match the probe selection. This savings in I/Os against rows that are not a part of the result set for the  
| query, is one of the primary advantages for this operation.  
|

Table 5. Radix Index Probe Attributes

Data Access Method	Radix Index Probe
<b>Description</b>	The index is quickly probed based upon the selection criteria that were rewritten into a series of ranges. Only those keys that satisfy the selection will be used to generate a table row number.
<b>Advantages</b>	<ul style="list-style-type: none"> <li>• Only those index entries that match any selection continue to be processed</li> <li>• Provides very quick access to the selected rows</li> <li>• Potential to extract all of the data from the index keys' values, thus eliminating the need for a Table Probe</li> <li>• Returns the rows back in a sequence based upon the keys of the index</li> </ul>
<b>Considerations</b>	Generally requires a Table Probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when a large number of rows are selected because of the random I/O associated with the Table Probe.
<b>Likely to be used</b>	<ul style="list-style-type: none"> <li>• When asking for or expecting only a few rows to be returned from the index</li> <li>• When sequencing the rows is required the query (for example, ordering or grouping)</li> <li>• When the selection columns match the leading key columns of the index</li> </ul>
<b>Example SQL statement</b>	<pre>CREATE INDEX X1 ON Employee (LastName, WorkDept)  SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01' AND 'E01' AND LastName IN ('Smith', 'Jones', 'Peterson') OPTIMIZE FOR ALL ROWS</pre>
<b>Messages indicating use</b>	<ul style="list-style-type: none"> <li>• Optimizer Debug: CPI4328 -- Access path of file X1 was used by query.</li> <li>• PRTSQLLINE: SQL4008 -- Index X1 used for table 1. SQL4011 -- Index scan-key row positioning used on table 1.</li> </ul>
<b>SMP parallel enabled</b>	Yes
<b>Also referred to as</b>	Index Probe Index Probe, Preload Index Probe, Distinct Index Probe Distinct, Preload Index Probe, Key Positioning Index Scan, Key Row Positioning
<b>Visual Explain icon</b>	

| The following example illustrates a query where the optimizer might choose the radix index probe access method:

```
| CREATE INDEX X1 ON Employee (LastName, WorkDept)
|
| SELECT * FROM Employee
| WHERE WorkDept BETWEEN 'A01' AND 'E01'
| AND LastName IN ('Smith', 'Jones', 'Peterson')
| OPTIMIZE FOR ALL ROWS
```

| In this example, the optimizer uses the index X1 to position (probe) to the first index entry that matches the selection built over both the LastName and WorkDept columns. The selection is rewritten into a series of ranges that match all of the leading key columns used from the index X1. The probe is then based upon the composite concatenated values for all of the leading keys. The pseudo-SQL for this rewritten SQL might look as follows:

```
| SELECT * FROM X1
| WHERE X1.LeadingKeys BETWEEN 'JonesA01' AND 'JonesE01'
| OR X1.LeadingKeys BETWEEN 'PetersonA01' AND 'PetersonE01'
| OR X1.LeadingKeys BETWEEN 'SmithA01' AND 'SmithE01'
```

| All of the key entries that satisfy the probe operation will then be used to generate a row number for the table associated with the index (for example, Employee). The row number will be used by a Table Probe operation to perform random I/O on the table to produce the results for the query. This processing continues until all of the rows that satisfy the index probe operation have been processed. Note that in this example, all of the index entries processed and rows retrieved met the index probe criteria. If additional selection were added that cannot be performed through an index probe operation (such as selection against columns which are not a part of the leading key columns of the index), the optimizer will perform an index scan operation within the range of probed values. This still allows for selection to be performed before the Table Probe operation.

## | Encoded Vector Index

| An encoded vector index is a permanent object that provides access to a table by assigning codes to distinct key values and then representing those values in a vector. The size of the vector will match the number of rows in the underlying table. Each vector entry represents the table row number in the same position. The codes generated to represent the distinct key values can be 1, 2 or 4 bytes in length, depending upon the number of distinct values that need to be represented. Because of their compact size and relative simplicity, the EVI can be used to process large amounts of data very efficiently.


| Even though an encoded vector index is used to represent the values stored in a table, the index itself cannot be used to directly gain access to the table. Instead, the encoded vector index can only be used to generate either a temporary row number list or a temporary row number bitmap. These temporary objects can then be used in conjunction with a Table Probe to specify the rows in the table that the query needs to process. The main difference with the Table Probe associated with an encoded vector index (versus a radix index) is that the paging associated with the table can be asynchronous. The I/O can now be scheduled more efficiently to take advantage of groups of selected rows. Large portions of the table can be skipped over where no rows are selected. For more information about encoded vector indexes, see “Encoded vector indexes” on page 122.

| Visual explain icon:



## Encoded Vector Index Probe

Table 6. Encoded Vector Index Probe Attributes

Data Access Method	Encoded Vector Index Probe
<b>Description</b>	The encoded vector index (EVI) is quickly probed based upon the selection criteria that were rewritten into a series of ranges. It produces either a temporary row number list or bitmap.
<b>Advantages</b>	<ul style="list-style-type: none"> <li>• Only those index entries that match any selection continue to be processed</li> <li>• Provides very quick access to the selected rows</li> <li>• Returns the row numbers in ascending sequence so that the Table Probe can be more aggressive in pre-fetching the rows for its operation</li> </ul>
<b>Considerations</b>	EVI's are generally built over a single key. The more distinct the column is and the higher the overflow percentage, the less advantageous the encoded vector index becomes. EVI's always require a Table Probe to be performed on the result of the EVI probe operation.
<b>Likely to be used</b>	<ul style="list-style-type: none"> <li>• When the selection columns match the leading key columns of the index</li> <li>• When an encoded vector index exists and savings in reduced I/O against the table justifies the extra cost of probing the EVI and fully populating the temporary row number list.</li> </ul>
<b>Example SQL statement</b>	<pre>CREATE ENCODED VECTOR INDEX EVI1 ON   Employee (WorkDept) CREATE ENCODED VECTOR INDEX EVI2 ON   Employee (Salary) CREATE ENCODED VECTOR INDEX EVI3 ON   Employee (Job)  SELECT * FROM Employee WHERE WorkDept = 'E01' AND Job = 'CLERK' AND Salary = 5000 OPTIMIZE FOR 99999 ROWS</pre>
<b>Messages indicating use</b>	<ul style="list-style-type: none"> <li>• Optimizer Debug:           <pre>CPI4239 -- Arrival sequence was used for file EMPLOYEE. CPI4338 -- 3 Access path(s) used for bitmap processing of file EMPLOYEE.</pre> </li> <li>• PRSQLINF:           <pre>SQL4010 -- Table scan access for table 1. SQL4032 -- Index EVI1 used for bitmap processing of table 1. SQL4032 -- Index EVI2 used for bitmap processing of table 1. SQL4032 -- Index EVI3 used for bitmap processing of table 1.</pre> </li> </ul>
<b>SMP parallel enabled</b>	Yes
<b>Also referred to as</b>	Encoded Vector Index Probe, Preload
<b>Visual Explain icon</b>	

Using the example above, the optimizer chooses to create a temporary row number bitmap for each of the encoded vector indexes used by this query. Each bitmap only identifies those rows that match the selection on the key columns for that index. These temporary row number bitmaps are then merged together to determine the intersection of the rows selected from each index. This intersection is used to form a final temporary row number bitmap that will be used to help schedule the I/O paging against the table for the selected rows.

The optimizer might choose to perform an index probe with a binary radix tree index if an index existed over all three columns. The implementation choice is probably decided by the number of rows to be returned and the anticipated cost of the I/O associated with each plan. If very few rows will be returned, the optimizer probably choose to use the binary radix tree index and perform the random I/O against the table. However, selecting more than a few rows will cause the optimizer to use the encoded vector indexes because of the savings associated with the more efficient scheduled I/O against the table.

---

## Temporary objects and access methods

Temporary objects are created by the optimizer in order to process a query. In general, these temporary objects are internal objects and cannot be accessed by a user.

*Table 7. Temporary Object's Data Access Methods*

Temporary Create Objects	Scan Operations	Probe Operations
"Temporary Hash Table"	"Hash Table Scan" on page 24	"Hash Table Probe" on page 25
"Temporary Sorted List" on page 27	"Sorted List Scan" on page 27	"Sorted List Probe" on page 28
"Temporary List" on page 31	"List Scan" on page 31	N/A
"Temporary Row Number List" on page 33	"Row Number List Scan" on page 33	"Row Number List Probe" on page 35
"Temporary Bitmap" on page 37	"Bitmap Scan" on page 37	"Bitmap Probe" on page 39
"Temporary Index" on page 40	"Temporary Index Scan" on page 41	"Temporary Index Probe" on page 43
"Temporary Buffer" on page 44	"Buffer Scan" on page 44	N/A

## Temporary Hash Table

The temporary hash table is a temporary object that allows the optimizer to collate the rows based upon a column or set of columns. The hash table can be either scanned or probed by the optimizer to satisfy different operations of the query.

A temporary hash table is an efficient data structure because the rows are organized for quick and easy retrieval after population has occurred. This is primarily due to the hash table remaining resident within main memory so as to avoid any I/Os associated with either the scan or probe against the temporary object. The optimizer will determine the optimal size for the hash table based upon the number of unique combinations (for example, cardinality) of the columns used as keys for the creation.

Additionally the hash table can be populated with all of the necessary columns to satisfy any further processing, avoiding any random I/Os associated with a Table Probe operation. However, the optimizer does have the ability to selectively include columns in the hash table when the calculated size will exceed the memory pool storage available for this query. In those cases, a Table Probe operation is required to recollect the missing columns from the hash table before the selected rows can be processed.

The optimizer also has the ability to populate the hash table with distinct values. If the query contains grouping or distinct processing, then all of the rows with the same key value are not required to be stored in the temporary object. They are still collated, but the distinct processing is performed during the population of the hash table itself. This allows a simple scan to be performed on the result in order to complete the grouping or distinct operation.

A temporary hash table is an internal data structure and can only be created by the database manager

Visual explain icon:




### | **Hash Table Scan**

| During a Hash Table Scan operation, the entire temporary hash table is scanned and all of the entries  
| contained within the hash table will be processed. The optimizer considers a hash table scan when the  
| data values need to be collated together, but the sequence of the data is not required. The use of a hash  
| table scan will allow the optimizer to generate a plan that can take advantage of any non-join selection  
| while creating the temporary hash table. An additional benefit of using a hash table scan is that the data  
| structure of the temporary hash table will usually cause the table data within the hash table to remain  
| resident within main memory after creation, thus reducing paging on the subsequent hash table scan  
| operation.



Table 8. Hash Table Scan Attributes


Data Access Method	Hash Table Scan
Description	Read all of the entries in a temporary hash table. The hash table may perform distinct processing to eliminate duplicates or takes advantage of the temporary hash table to collate all of the rows with the same value together.
Advantages	<ul style="list-style-type: none"> <li>Reduces the random I/O to the table generally associated with longer running queries that would otherwise use an index to collate the data</li> <li>Selection can be performed before generating the hash table to subset the number of rows in the temporary object</li> </ul>
Considerations	Generally used for distinct or group by processing. Can perform poorly when the entire hash table does not stay resident in memory as it is being processed.
Likely to be used	<ul style="list-style-type: none"> <li>When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA)</li> <li>When the data is required to be collated based upon a column or columns for distinct or grouping</li> </ul>
Example SQL statement	<pre>SELECT COUNT(*), FirstNme FROM Employee WHERE WorkDept BETWEEN 'A01' AND 'E01' GROUP BY FirstNme</pre>
Messages indicating use	<p>There are multiple ways in which a hash scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a hash scan was used.</p> <ul style="list-style-type: none"> <li>Optimizer Debug: CPI4329 -- Arrival sequence was used for file EMPLOYEE.</li> <li>PRTSQLINF: SQL4010 -- Table scan access for table 1. SQL4029 -- Hashing algorithm used to process the grouping.</li> </ul>
SMP parallel enabled	Yes
Also referred to as	<p>Hash Scan, Preload</p> <p>Hash Table Scan Distinct</p> <p>Hash Table Scan Distinct, Preload</p>
Visual Explain icon	

## Hash Table Probe

A hash table probe operation is used to retrieve rows from a temporary hash table based upon a probe lookup operation. The optimizer initially identifies the keys of the temporary hash table from the join criteria specified in the query. This is done so that when the hash table probe is performed, the values used to probe into the temporary hash table will be extracted from the join-from criteria specified in the selection. Those values will be sent through the same hashing algorithm used to populate the temporary

hash table in order to determine if any rows have a matching (equal) value. All of the matching join rows are then returned to be further processed by the query.

Table 9. Hash Table Probe Attributes

Data Access Method	Hash Table Probe
<b>Description</b>	The temporary hash table is quickly probed based upon the join criteria.
<b>Advantages</b>	<ul style="list-style-type: none"> <li>• Provides very quick access to the selected rows that match probe criteria</li> <li>• Reduces the random I/O to the table generally associated with longer running queries that use an index to collate the data</li> <li>• Selection can be performed before generating the hash table to subset the number of rows in the temporary object</li> </ul>
<b>Considerations</b>	Generally used to process equal join criteria. Can perform poorly when the entire hash table does not stay resident in memory as it is being processed.
<b>Likely to be used</b>	<ul style="list-style-type: none"> <li>• When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA)</li> <li>• When the data is required to be collated based upon a column or columns for join processing</li> <li>• The join criteria was specified using an equals (=) operator</li> </ul>
<b>Example SQL statement</b>	<pre>SELET * FROM Employee XXX, Department YYY WHERE XXX.WorkDept = YYY.DeptNbr OPTIMIZE FOR ALL ROWS</pre>
<b>Messages indicating use</b>	<p>There are multiple ways in which a hash probe can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a hash probe was used.</p> <ul style="list-style-type: none"> <li>• Optimizer Debug: <pre>CPI4327 -- File EMPLOYEE processed in join            position 1. CPI4327 -- File DEPARTMENT processed in join            position 2.</pre> </li> <li>• PRTSQLINF: <pre>SQL4007 -- Query implementation for join            position 1 table 1. SQL4010 -- Table scan access for table 1. SQL4007 -- Query implementation for join            position 2 table 2. SQL4010 -- Table scan access for table 2.</pre> </li> </ul>
<b>SMP parallel enabled</b>	Yes
<b>Also referred to as</b>	<p>Hash Table Probe, Preload</p> <p>Hash Table Probe Distinct</p> <p>Hash Table Probe Distinct, Preload</p>
<b>Visual Explain icon</b>	

| The hash table probe access method is generally considered when determining the implementation for a secondary table of a join. The hash table is created with the key columns that match the equal selection or join criteria for the underlying table. The hash table probe allows the optimizer to choose the most efficient implementation to select the rows from the underlying table without regard for any join criteria. This single pass through the underlying table can now choose to perform a Table Scan or use an existing index to select the rows needed for the hash table population.

| Since hash tables are constructed so that the majority of the hash table will remain resident within main memory, the I/O associated with a hash probe is minimal. Additionally, if the hash table was populated with all necessary columns from the underlying table, no additional Table Probe will be required to finish processing this table, once again causing further I/O savings.

## | **Temporary Sorted List**

| The temporary sorted list is a temporary object that allows the optimizer to sequence rows based upon a column or set of columns. The sorted list can be either scanned or probed by the optimizer to satisfy different operations of the query.

| A temporary sorted list is a data structure where the rows are organized for quick and easy retrieval after population has occurred. During population, the rows are copied into the temporary object and then a second pass is made through the temporary object to perform the sort. In order to optimize the creation of this temporary object, minimal data movement is performed while the sort is processed. It is generally not as efficient to probe a temporary sorted list as it is to probe a temporary hash table.

| Additionally, the sorted list can be populated with all of the necessary columns to satisfy any further processing, avoiding any random I/Os associated with a Table Probe operation. However, the optimizer does have the ability to selectively include columns in the sorted list when the calculated size will exceed the memory pool storage available for this query. In those cases, a Table Probe operation is required to recollect the missing columns from the sorted list before the selected rows can be processed.

| A temporary sorted list is an internal data structure and can only be created by the database manager.


| Visual explain icon:



## | **Sorted List Scan**

| During a sorted list scan operation, the entire temporary sorted list is scanned and all of the entries contained within the sorted list will be processed. A sorted list scan is generally considered when the optimizer is considering a plan that requires the data values to be sequenced. The use of a sorted list scan will allow the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary sorted list. An additional benefit of using a sorted list scan is that the data structure of the temporary sorted list will usually cause the table data within the sorted list to remain resident within main memory after creation thus reducing paging on the subsequent sorted list scan operation.

Table 10. Sorted List Scan Attributes


Data Access Method	Sorted List Scan
<b>Description</b>	Read all of the entries in a temporary sorted list. The sorted list may perform distinct processing to eliminate duplicate values or take advantage of the temporary sorted list to sequence all of the rows.
<b>Advantages</b>	<ul style="list-style-type: none"> <li>• Reduces the random I/O to the table generally associated with longer running queries that would otherwise use an index to sequence the data.</li> <li>• Selection can be performed prior to generating the sorted list to subset the number of rows in the temporary object</li> </ul>
<b>Considerations</b>	Generally used to process ordering or distinct processing. Can perform poorly when the entire sorted list does not stay resident in memory as it is being populated and processed.
<b>Likely to be used</b>	<ul style="list-style-type: none"> <li>• When the use of temporary results is allowed by the query environmental parameter (ALWCOPYDTA)</li> <li>• When the data is required to be ordered based upon a column or columns for ordering or distinct processing</li> </ul>
<b>Example SQL statement</b>	<pre>CREATE INDEX X1 ON Employee (LastName, WorkDept)  SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01' AND 'E01' ORDER BY FirstNme OPTIMIZE FOR ALL ROWS</pre>
<b>Messages indicating use</b>	<p>There are multiple ways in which a sorted list scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a sorted list scan was used.</p> <ul style="list-style-type: none"> <li>• Optimizer Debug: <pre>CPI4328 -- Access path of file X1 was used by query. CPI4325 -- Temporary result file built for query.</pre> </li> <li>• PRTSQLINF: <pre>SQL4008 -- Index X1 used for table 1. SQL4002 -- Reusable ODP sort used.</pre> </li> </ul>
<b>SMP parallel enabled</b>	No
<b>Also referred to as</b>	Sorted List Scan, Preload Sorted List Scan Distinct Sorted List Scan Distinct, Preload
<b>Visual Explain icon</b>	

## Sorted List Probe

A sorted list probe operation is used to retrieve rows from a temporary sorted list based upon a probe lookup operation. The optimizer initially identifies the keys of the temporary sorted list from the join criteria specified in the query. This is done so that when the sorted list probe is performed, the values

| used to probe into the temporary sorted list will be extracted from the join-from criteria specified in the  
| selection. Those values will be used to position within the sorted list in order to determine if any rows  
| have a matching value. All of the matching join rows are then returned to be further processed by the  
| query.  
|

Table 11. Sorted List Probe Attributes

Data Access Method	Sorted List Probe
<b>Description</b>	The temporary sorted list is quickly probed based upon the join criteria.
<b>Advantages</b>	<ul style="list-style-type: none"> <li>• Provides very quick access to the selected rows that match probe criteria</li> <li>• Reduces the random I/O to the table generally associated with longer running queries that would otherwise use an index to collate the data</li> <li>• Selection can be performed prior to generating the sorted list to subset the number of rows in the temporary object</li> </ul>
<b>Considerations</b>	Generally used to process non-equal join criteria. Can perform poorly when the entire sorted list does not stay resident in memory as it is being populated and processed.
<b>Likely to be used</b>	<ul style="list-style-type: none"> <li>• When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA)</li> <li>• When the data is required to be collated based upon a column or columns for join processing</li> <li>• The join criteria was specified using a non-equals operator</li> </ul>
<b>Example SQL statement</b>	<pre>SELECT * FROM Employee XXX, Department YYY WHERE XXX.WorkDept &gt; YYY.DeptNbr OPTIMIZE FOR ALL ROWS</pre>
<b>Messages indicating use</b>	<p>There are multiple ways in which a sorted list probe can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a sorted list probe was used.</p> <ul style="list-style-type: none"> <li>• Optimizer Debug: <pre>CPI4327 -- File EMPLOYEE processed in join position 1. CPI4327 -- File DEPARTMENT processed in join            position 2.</pre> </li> <li>• PRTSQLINF: <pre>SQL4007 -- Query implementation for join            position 1 table 1. SQL4010 -- Table scan access for table 1. SQL4007 -- Query implementation for join            position 2 table 2. SQL4010 -- Table scan access for table 2.</pre> </li> </ul>
<b>SMP parallel enabled</b>	Yes
<b>Also referred to as</b>	<p>Sorted List Probe, Preload</p> <p>Sorted List Probe Distinct</p> <p>Sorted List Probe Distinct, Preload</p>
<b>Visual Explain icon</b>	

The sorted list probe access method is generally considered when determining the implementation for a secondary table of a join. The sorted list is created with the key columns that match the non-equal join criteria for the underlying table. The sorted list probe allows the optimizer to choose the most efficient implementation to select the rows from the underlying table without regard for any join criteria. This

| single pass through the underlying table can now choose to perform a Table Scan or use an existing index  
| to select the rows needed for the sorted list population.

| Since sorted lists are constructed so that the majority of the temporary object will remain resident within  
| main memory, the I/O associated with a sorted list is minimal. Additionally, if the sorted list was  
| populated with all necessary columns from the table, no additional Table Probe will be required in order  
| to finish processing this table, once again causing further I/O savings.

## | **Temporary List**

| The temporary list is a temporary object that allows the optimizer to store intermediate results of a query.  
| The list is an unsorted data structure that is used to simplify the operation of the query. Since the list  
| does not have any keys, the rows within the list can only be retrieved by a sequential scan operation.

| The temporary list can be used for a variety of reasons, some of which include an overly complex view  
| or derived table, Symmetric Multiprocessing (SMP) or simply to prevent a portion of the query from  
| being processed multiple times.

| A temporary list is an internal data structure and can only be created by the database manager.


| Visual explain icon:



## | **List Scan**

| The list scan operation is used when a portion of the query will be processed multiple times, but no key  
| columns can be identified. In these cases, that portion of the query is processed once and its results are  
| stored within the temporary list. The list can then be scanned for only those rows that satisfy any  
| selection or processing contained within the temporary object.

Table 12. List Scan Attributes

Data Access Method	List Scan
Description	Sequentially scan and process all of the rows in the temporary list.
Advantages	<ul style="list-style-type: none"> <li>The temporary list and list scan can be used by the optimizer to minimize repetition of an operation or to simplify the optimizer's logic flow</li> <li>Selection can be performed before generating the list to subset the number of rows in the temporary object</li> </ul>
Considerations	Generally used to prevent portions of the query from being processed multiple times when no key columns are required to satisfy the request.
Likely to be used	<ul style="list-style-type: none"> <li>When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA)</li> <li>When Symmetric Multiprocessing will be used for the query</li> </ul>
Example SQL statement	<pre>SELECT * FROM Employee XXX, Department YYY WHERE XXX.LastName IN ('Smith', 'Jones', 'Peterson') AND YYY.DeptNo BETWEEN 'A01' AND 'E01' OPTIMIZE FOR ALL ROWS</pre>
Messages indicating use	<p>There are multiple ways in which a list scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a list scan was used.</p> <ul style="list-style-type: none"> <li>Optimizer Debug: <pre>CPI4325 -- Temporary result file built for query. CPI4327 -- File EMPLOYEE processed in join            position 1. CPI4327 -- File DEPARTMENT processed in join            position 2.</pre> </li> <li>PRTSQLINF: <pre>SQL4007 -- Query implementation for join            position 1 table 1. SQL4010 -- Table scan access for table 1. SQL4007 -- Query implementation for join            position 2 table 2. SQL4001 -- Temporary result created SQL4010 -- Table scan access for table 2.</pre> </li> </ul>
SMP parallel enabled	Yes
Also referred to as	List Scan, Preload
Visual Explain icon	

Using the example above, the optimizer chose to create a temporary list to store the selected rows from the DEPARTMENT table. Since there is no join criteria, a cartesian product join is performed between the two tables. To prevent the join from scanning all of the rows of the DEPARTMENT table for each join possibility, the selection against the DEPARTMENT table is performed once and the results are stored in the temporary list. The temporary list is then scanned for the cartesian product join.



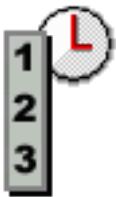
## Temporary Row Number List

The temporary row number list is a temporary object that allows the optimizer to sequence rows based upon their row address (their row number). The row number list can be either scanned or probed by the optimizer to satisfy different operations of the query.

A temporary row number list is a data structure where the rows are organized for quick and efficient retrieval. The temporary only contains the row number for the associated row. Since no table data is present within the temporary, a table probe operation is usually associated with this temporary in order to retrieve the underlying table data. Because the row numbers are sorted, the random I/O associated with the table probe operation can be performed more efficiently. The database manager will perform pre-fetch or look ahead logic to determine if multiple rows are located on adjacent pages. If so, the table probe will request a larger I/O to bring the rows into main memory more efficiently.

A temporary row number list is an internal data structure and can only be created by the database manager.

Visual explain icon:



## Row Number List Scan

During a row number list scan operation, the entire temporary row number list is scanned and all of the row addresses contained within the row number list will be processed. A row number list scan is generally considered when the optimizer is considering a plan that involves an encoded vector index or if the cost of the random I/O associated with an index probe or scan operation can be reduced by first pre-processing and sorting the row numbers associated with the Table Probe operation. The use of a row number list scan will allow the optimizer to generate a plan that can take advantage of multiple indexes to match up to different portions of the query.


An additional benefit of using a row number list scan is that the data structure of the temporary row number list guarantees that the row numbers are sorted, it closely mirrors the row number layout of the table data ensuring that the paging on the table will never revisit the same page of data twice. This results in increased I/O savings for the query.

A row number list scan is identical to a bitmap scan operation. The only difference between the two operations is that a row number list scan is performed over a list of row addresses while the bitmap scan is performed over a bitmap that represents the row addresses. See “Bitmap Scan” on page 37.

Table 13. Row Number List Scan

Data Access Method	Row Number List Scan
Description	Sequentially scan and process all of the row numbers in the temporary row number list. The sorted row numbers can be merged with other temporary row number lists or can be used as input into a Table Probe operation.
Advantages	<ul style="list-style-type: none"><li>• The temporary row number list only contains address, no data, so the temporary can be efficiently scanned within memory</li><li>• The row numbers contained within the temporary object are sorted to provide efficient I/O processing to access the underlying table</li><li>• Selection is performed as the row number list is generated to subset the number of rows in the temporary object</li></ul>

Table 13. Row Number List Scan (continued)

Data Access Method	Row Number List Scan
<b>Considerations</b>	Since the row number list only contains the addresses of the selected row in the table, a separate Table Probe operation must be performed in order to fetch the table rows
<b>Likely to be used</b>	<ul style="list-style-type: none"> <li>• When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA)</li> <li>• When the cost of sorting of the row number is justified by the more efficient I/O that can be performed during the Table Probe operation</li> <li>• When multiple indexes over the same table need to be combined in order to minimize the number of selected rows</li> </ul>
<b>Example SQL statement</b>	<pre>CREATE INDEX X1 ON Employee (WorkDept) CREATE ENCODED VECTOR INDEX EVI2 ON     Employee (Salary) CREATE ENCODED VECTOR INDEX EVI3 ON     Employee (Job)  SELECT * FROM Employee WHERE WorkDept = 'E01' AND Job = 'CLERK' AND Salary = 5000 OPTIMIZE FOR 99999 ROWS</pre>
<b>Messages indicating use</b>	<p>There are multiple ways in which a row number list scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a row number list scan was used.</p> <ul style="list-style-type: none"> <li>• Optimizer Debug: <pre>CPI4239 -- Arrival sequence was used for file     EMPLOYEE. CPI4338 -- 3 Access path(s) used for bitmap     processing of file EMPLOYEE.</pre> </li> <li>• PRSQLINF: <pre>SQL4010 -- Table scan access for table 1. SQL4032 -- Index X1 used for bitmap     processing of table 1. SQL4032 -- Index EVI2 used for bitmap     processing of table 1. SQL4032 -- Index EVI3 used for bitmap     processing of table 1.</pre> </li> </ul>
<b>SMP parallel enabled</b>	Yes
<b>Also referred to as</b>	Row Number List Scan, Preload
<b>Visual Explain icon</b>	

Using the example above, the optimizer created a temporary row number list for each of the indexes used by this query. This query used a combination of a radix index and two encoded vector indexes to create the row number lists. The temporary row number lists for each index was scanned and merged into a final composite row number list that represents the intersection of the rows represented by all of the temporary row number lists. The final row number list is then used by the Table Probe operation to determine what rows are selected and need to be processed for the query results.


## | **Row Number List Probe**

| A row number list probe operation is used to test row numbers generated by a separate operation against the selected rows of a temporary row number list. The row numbers can be generated by any operation that constructs a row number for a table. That row number is then used to probe into a temporary row number list to determine if that row number matches the selection used to generate the temporary row number list. The use of a row number list probe operation allows the optimizer to generate a plan that can take advantage of any sequencing provided by an index, but still use the row number list to perform additional selection prior to any Table Probe operations.

| A row number list probe is identical to a bitmap probe operation. The only difference between the two operations is that a row number list probe is performed over a list of row addresses while the bitmap probe is performed over a bitmap that represents the row addresses. See “Bitmap Probe” on page 39.

|

Table 14. Row Number List Probe

Data Access Method	Row Number List Probe
Description	The temporary row number list is quickly probed based upon the row number generated by a separate operation.
Advantages	<ul style="list-style-type: none"> <li>• The temporary row number list only contains a rows' address, no data, so the temporary can be efficiently probed within memory</li> <li>• The row numbers represented within the row number list are sorted to provide efficient lookup processing to test the underlying table</li> <li>• Selection is performed as the row number list is generated to subset the number of selected rows in the temporary object</li> </ul>
Considerations	Since the row number list only contains the addresses of the selected rows in the table, a separate Table Probe operation must be performed in order to fetch the table rows
Likely to be used	<ul style="list-style-type: none"> <li>• When the use of temporary results is allowed by the query environmental parameter (ALWCOPYDTA)</li> <li>• When the cost of creating and probing the row number list is justified by reducing the number of Table Probe operations that must be performed</li> <li>• When multiple indexes over the same table need to be combined in order to minimize the number of selected rows</li> </ul>
Example SQL statement	<pre>CREATE INDEX X1 ON Employee (WorkDept) CREATE ENCODED VECTOR INDEX EVI2 ON     Employee (Salary) CREATE ENCODED VECTOR INDEX EVI3 ON     Employee (Job)  SELECT * FROM Employee WHERE WorkDept = 'E01' AND Job = 'CLERK' AND Salary = 5000 ORDER BY WorkDept</pre>
Messages indicating use	<p>There are multiple ways in which a row number list probe can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a row number list probe was used.</p> <ul style="list-style-type: none"> <li>• Optimizer Debug: <pre>CPI4328 -- Access path of file X1 was used by query. CPI4338 -- 2 Access path(s) used for bitmap processing of file EMPLOYEE.</pre> </li> <li>• PRTSQLINF: <pre>SQL4008 -- Index X1 used for table 1. SQL4011 -- Index scan-key row positioning used on table 1. SQL4032 -- Index EVI2 used for bitmap processing of table 1. SQL4032 -- Index EVI3 used for bitmap processing of table 1.</pre> </li> </ul>
SMP parallel enabled	Yes
Also referred to as	Row Number List Probe, Preload
Visual Explain icon	

Using the example above, the optimizer created a temporary row number list for each of the encoded vector indexes. Additionally, an index probe operation was performed against the radix index X1 to satisfy the ordering requirement. Since the ORDER BY clause requires that the resulting rows be sequenced by the WorkDept column, we can no longer simply scan the temporary row number list to process the selected rows. However, the temporary row number list can be probed using a row address extracted from the index X1 used to satisfy the ordering. By probing the temporary row number list with the row address extracted from index probe operation we preserve the sequencing of the keys in the index X1 and can still test the row against the selected rows within the row number list.

## Temporary Bitmap

The temporary bitmap is a temporary object that allows the optimizer to sequence rows based upon their row address (their row number). The bitmap can be either scanned or probed by the optimizer to satisfy different operations of the query.

A temporary bitmap is a data structure that uses a bitmap to represent all of the row numbers for a table. Since each row is represented by a separate bit, all of the rows within a table can be represented in a fairly condensed form. When a row is selected by the temporary, the bit within the bitmap that corresponds to the selected row is set on. After the temporary bitmap is populated, all of the selected rows can be retrieved in a sorted manner for quick and efficient retrieval. The temporary only represents the row number for the associated selected rows. No table data is present within the temporary, so a table probe operation is usually associated with this temporary in order to retrieve the underlying table data. Because the bitmap is by definition sorted, the random I/O associated with the table probe operation can be performed more efficiently. The database manager will perform pre-fetch or look ahead logic to determine if multiple rows are located on adjacent pages. If so, the table probe will request a larger I/O to bring the rows into main memory more efficiently.

A temporary bitmap is an internal data structure and can only be created by the database manager.

Visual explain icon:



## Bitmap Scan

During a bitmap scan operation, the entire temporary bitmap is scanned and all of the row addresses contained within the bitmap will be processed. A bitmap scan is generally considered when the optimizer is considering a plan that involves an encoded vector index or if the cost of the random I/O associated with an index probe or scan operation can be reduced by first pre-processing and sorting the row numbers associated with the Table Probe operation. The use of a bitmap scan will allow the optimizer to generate a plan that can take advantage of multiple indexes to match up to different portions of the query.


An additional benefit of using a bitmap scan is that the data structure of the temporary bitmap guarantees that the row numbers are sorted; it closely mirrors the row number layout of the table data ensuring that the paging on the table will never revisit the same page of data twice. This results in increased I/O savings for the query.

A bitmap scan is identical to a row number list scan operation. The only difference between the two operations is that a row number list scan is performed over a list of row addresses while the bitmap scan is performed over a bitmap that represents the row addresses. See “Row Number List Scan” on page 33.

Table 15. Bitmap Scan Attributes

Data Access Method	Bitmap Scan Attributes
<b>Description</b>	Sequentially scan and process all of the row numbers in the temporary bitmap. The sorted row numbers can be merged with other temporary bitmaps or can be used as input into a Table Probe operation.
<b>Advantages</b>	<ul style="list-style-type: none"> <li>• The temporary bitmap only contains a reference to a rows' address, no data, so the temporary can be efficiently scanned within memory</li> <li>• The row numbers represented within the temporary object are sorted to provide efficient I/O processing to access the underlying table</li> <li>• Selection is performed as the bitmap is generated to subset the number of selected rows in the temporary object</li> </ul>
<b>Considerations</b>	Since the bitmap only contains the addresses of the selected row in the table, a separate Table Probe operation must be performed in order to fetch the table rows
<b>Likely to be used</b>	<ul style="list-style-type: none"> <li>• When the use of temporary results is allowed by the query environmental parameter (ALWCOPYDTA)</li> <li>• When the cost of sorting of the row numbers is justified by the more efficient I/O that can be performed during the Table Probe operation</li> <li>• When multiple indexes over the same table need to be combined in order to minimize the number of selected rows</li> </ul>
<b>Example SQL statement</b>	<pre>CREATE INDEX X1 ON Employee (WorkDept) CREATE ENCODED VECTOR INDEX EVI2 ON     Employee (Salary) CREATE ENCODED VECTOR INDEX EVI3 ON     Employee (Job)  SELECT * FROM Employee WHERE WorkDept = 'E01' AND Job = 'CLERK' AND Salary = 5000 OPTIMIZE FOR 99999 ROWS</pre>
<b>Messages indicating use</b>	<p>There are multiple ways in which a bitmap scan can be indicated through the messages. The messages in this example illustrate how the Classic Query Engine will indicate a bitmap scan was used.</p> <ul style="list-style-type: none"> <li>• Optimizer Debug: <pre>CPI4239 -- Arrival sequence was used for file     EMPLOYEE. CPI4338 -- 3 Access path(s) used for bitmap     processing of file EMPLOYEE.</pre> </li> <li>• PRSQLINF: <pre>SQL4010 -- Table scan access for table 1. SQL4032 -- Index X1 used for bitmap     processing of table 1. SQL4032 -- Index EVI2 used for bitmap     processing of table 1. SQL4032 -- Index EVI3 used for bitmap     processing of table 1.</pre> </li> </ul>
<b>SMP parallel enabled</b>	Yes
<b>Also referred to as</b>	<p>Bitmap Scan, Preload</p> <p>Row Number Bitmap Scan</p> <p>Row Number Bitmap Scan, Preload</p> <p>Skip Sequential Scan</p>

Table 15. Bitmap Scan Attributes (continued)

Data Access Method	Bitmap Scan Attributes
Visual Explain icon	

Using the example above, the optimizer created a temporary bitmap for each of the indexes used by his query. This query used a combination of a radix index and two encoded vector indexes to create the row number lists. The temporary bitmaps for each index were scanned and merged into a final composite bitmap that represents the intersection of the rows represented by all of the temporary bitmaps. The final bitmap is then used by the Table Probe operation to determine what rows are selected and need to be processed for the query results.

### Bitmap Probe


A bitmap probe operation is used to test row numbers generated by a separate operation against the selected rows of a temporary bitmap. The row numbers can be generated by any operation that constructs a row number for a table. That row number is then used to probe into a temporary bitmap to determine if that row number matches the selection used to generate the temporary bitmap. The use of a bitmap probe operation allows the optimizer to generate a plan that can take advantage of any sequencing provided by an index, but still use the bitmap to perform additional selection prior to any Table Probe operations.

A bitmap probe is identical to a row number list probe operation. The only difference between the two operations is that a row number list probe is performed over a list of row addresses while the bitmap probe is performed over a bitmap that represents the row addresses. See “Row Number List Probe” on page 35.

Table 16. Bitmap Probe Attributes

Data Access Method	Bitmap Probe Attributes
Description	The temporary bitmap is quickly probed based upon the row number generated by a separate operation.
Advantages	<ul style="list-style-type: none"> <li>• The temporary bitmap only contains a reference to a rows’ address, no data, so the temporary can be efficiently probed within memory</li> <li>• The row numbers represented within the bitmap are sorted to provide efficient lookup processing to test the underlying table</li> <li>• Selection is performed as the bitmap is generated to subset the number of selected rows in the temporary object</li> </ul>
Considerations	Since the bitmap only contains the addresses of the selected rows in the table, a separate Table Probe operation must be performed in order to fetch the table rows
Likely to be used	<ul style="list-style-type: none"> <li>• When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA)</li> <li>• When the cost of creating and probing the bitmap is justified by reducing the number of Table Probe operations that must be performed</li> <li>• When multiple indexes over the same table need to be combined in order to minimize the number of selected rows</li> </ul>

Table 16. Bitmap Probe Attributes (continued)

Data Access Method	Bitmap Probe Attributes
Example SQL statement	<pre>CREATE INDEX X1 ON Employee (WorkDept) CREATE ENCODED VECTOR INDEX EVI2 ON   Employee (Salary) CREATE ENCODED VECTOR INDEX EVI3 ON   Employee (Job)  SELECT * FROM Employee WHERE WorkDept = 'E01' AND Job = 'CLERK' AND Salary = 5000 ORDER BY WorkDept</pre>
Messages indicating use	<p>There are multiple ways in which a bitmap probe can be indicated through the messages. The messages in this example illustrate how the Classic Query Engine will indicate a bitmap probe was used.</p> <ul style="list-style-type: none"> <li>Optimizer Debug:           <pre>CPI4328 -- Access path of file X1 was used by query. CPI4338 -- 2 Access path(s) used for bitmap            processing of file EMPLOYEE.</pre> </li> <li>PRTSQLINF:           <pre>SQL4008 -- Index X1 used for table 1. SQL4011 -- Index scan-key row positioning            used on table 1. SQL4032 -- Index EVI2 used for bitmap            processing of table 1. SQL4032 -- Index EVI3 used for bitmap            processing of table 1.</pre> </li> </ul>
SMP parallel enabled	Yes
Also referred to as	Bitmap Probe, Preload Row Number Bitmap Probe Row Number Bitmap Probe, Preload
Visual Explain icon	

Using the example above, the optimizer created a temporary bitmap for each of the encoded vector indexes. Additionally, an index probe operation was performed against the radix index X1 to satisfy the ordering requirement. Since the ORDER BY clause requires that the resulting rows be sequenced by the WorkDept column, we can no longer simply scan the temporary bitmap to process the selected rows. However, the temporary bitmap can be probed using a row address extracted from the index X1 used to satisfy the ordering. By probing the temporary bitmap with the row address extracted from index probe operation we preserve the sequencing of the keys in the index X1 and can still test the row against the selected rows within the bitmap.

## Temporary Index

A temporary index is a temporary object that allows the optimizer to create and use a radix index for a specific query. The temporary index has all of the same attributes and benefits as a radix index that is created by a user through the CREATE INDEX SQL statement or CRTLF CL command. Additionally, the



| temporary index is optimized for use by the optimizer to satisfy a specific query request. This includes  
| setting the logical page size and applying any selection to the creation to speed up the use of the  
| temporary index after it has been created.

| The temporary index can be used to satisfy a variety of query requests, but it is only considered by the  
| Classic Query Engine when the query contains any of the following:

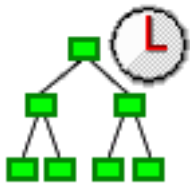
- | • Ordering
- | • Grouping
- | • Joins

| Generally a temporary index is a very expensive temporary object to create. It can be populated by either  
| performing a table scan to fetch the rows to be used for the index or by performing an index scan or  
| probe against an existing radix index to produce the rows. The optimizer considers all of the methods  
| available when determining which method to use to produce the rows for the index creation. This  
| process is similar to the costing and selection of the other temporary objects used by the optimizer.

| One significant difference between the other forms of temporary objects and the temporary index is that  
| the temporary index is the only form of a temporary object that does not require a copy of the rows to be  
| performed. The temporary index is identical to a radix index in that as any inserts or updates are  
| performed against the table, those changes are reflected immediately within the temporary index through  
| the normal index maintenance processing.

| A temporary index is an internal data structure and can only be created by the database manager.

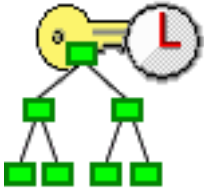
| Visual explain icon:



### | **Temporary Index Scan**

| A temporary index scan operation is identical to the index scan operation that is performed upon the  
| permanent radix index. It is still used to retrieve the rows from a table in a keyed sequence; however, the  
| temporary index object must first be created. All of the rows in the index will be sequentially processed,  
| but the resulting row numbers will be sequenced based upon the key columns. The sequenced rows can  
| be used by the optimizer to satisfy a portion of the query request (such as ordering or grouping). See  
| "Radix Index Scan" on page 17.

Table 17. Temporary Index Scan Attributes

Data Access Method	Temporary Index Scan
Description	Sequentially scan and process all of the keys associated with the temporary index.
Advantages	<ul style="list-style-type: none"> <li>• Potential to extract all of the data from the index keys' values, thus eliminating the need for a Table Probe</li> <li>• Returns the rows back in a sequence based upon the keys of the index</li> </ul>
Considerations	Generally requires a Table Probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when a large number of rows are selected because of the random I/O associated with the Table Probe.
Likely to be used	<ul style="list-style-type: none"> <li>• When sequencing the rows is required for the query (for example, ordering or grouping)</li> <li>• When the selection columns cannot be matched against the leading key columns of the index</li> <li>• When the overhead cost associated with the creation of the temporary index can be justified against other alternative methods to implement this query</li> </ul>
Example SQL statement	<pre>SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01' AND 'E01' ORDER BY LastName OPTIMIZE FOR ALL ROWS</pre>
Messages indicating use	<ul style="list-style-type: none"> <li>• Optimizer Debug: CPI4321 -- Access path built for file EMPLOYEE.</li> <li>• PRSQLINF: SQL4009 -- Index created for table 1.</li> </ul>
SMP parallel enabled	Yes
Also referred to as	Index Scan  Index Scan, Preload  Index Scan, Distinct  Index Scan Distinct, Preload  Index Scan, Key Selection
Visual Explain icon	

Using the example above, the optimizer chose to create a temporary index to sequence the rows based upon the LastName column. A temporary index scan might then be performed to satisfy the ORDER BY clause in this query.

The optimizer will determine where the selection against the WorkDept column best belongs. It can be performed as the temporary index itself is being created or it can be performed as a part of the

temporary index scan. Adding the selection to the temporary index creation has the possibility of making the open data path (ODP) for this query non-reusable. This ODP reuse is taken into consideration when determining how selection will be performed.

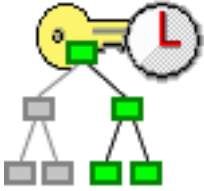
## Temporary Index Probe

A temporary index probe operation is identical to the index probe operation that is performed upon the permanent radix index. Its main function is to provide a form of quick access against the index keys of the temporary index; however it can still be used to retrieve the rows from a table in a keyed sequence. The temporary index is used by the optimizer to satisfy the join portion of the query request. See "Radix Index Probe" on page 18.

Table 18. Temporary Index Probe Attributes

Data Access Method	Temporary Index Probe
<b>Description</b>	The index is quickly probed based upon the selection criteria that were rewritten into a series of ranges. Only those keys that satisfy the selection will be used to generate a table row number.
<b>Advantages</b>	<ul style="list-style-type: none"> <li>• Only those index entries that match any selection continue to be processed. Provides very quick access to the selected rows</li> <li>• Potential to extract all of the data from the index keys' values, thus eliminating the need for a Table Probe</li> <li>• Returns the rows back in a sequence based upon the keys of the index</li> </ul>
<b>Considerations</b>	Generally requires a Table Probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when a large number of rows are selected because of the random I/O associated with the Table Probe.
<b>Likely to be used</b>	<ul style="list-style-type: none"> <li>• When the ability to probe the rows required for the query (for example, joins) exists</li> <li>• When the selection columns cannot be matched against the leading key columns of the index</li> <li>• When the overhead cost associated with the creation of the temporary index can be justified against other alternative methods to implement this query</li> </ul>
<b>Example SQL statement</b>	<pre> SELECT * FROM Employee XXX, Department YYY WHERE XXX.WorkDept = YYY.DeptNo OPTIMIZE FOR ALL ROWS </pre>
<b>Messages indicating use</b>	<p>There are multiple ways in which a temporary index probe can be indicated through the messages. The messages in this example illustrate one example of how the Classic Query Engine will indicate a temporary index probe was used.</p> <ul style="list-style-type: none"> <li>• Optimizer Debug: <pre> CPI4321 -- Access path built for file DEPARTMENT. CPI4327 -- File EMPLOYEE processed in join            position 1. CPI4326 -- File DEPARTMENT processed in join            position 2. </pre> </li> <li>• PRSQLINF: <pre> SQL4007 -- Query implementation for join            position 1 table 1. SQL4010 -- Table scan access for table 1. SQL4007 -- Query implementation for join            position 2 table 2. SQL4009 -- Index created for table 2. </pre> </li> </ul>
<b>SMP parallel enabled</b>	Yes

Table 18. Temporary Index Probe Attributes (continued)

Data Access Method	Temporary Index Probe
Also referred to as	Index Probe Index Probe, Preload Index Probe, Distinct Index Probe Distinct, Preload Index Probe, Key Selection
Visual Explain icon	

Using the example above, the optimizer chose to create a temporary index over the DeptNo column to help satisfy the join requirement against the DEPARTMENT table. A temporary index probe was then performed against the temporary index to process the join criteria between the two tables. In this particular case, there was no additional selection that might be applied against the DEPARTMENT table while the temporary index was being created.

## Temporary Buffer

The temporary buffer is a temporary object that is used to help facilitate operations such as parallelism. It is an unsorted data structure that is used to store intermediate rows of a query. The main difference between a temporary buffer and a temporary list is that the buffer does not have to be fully populated in order to allow its results to be processed. The temporary buffer acts as a serialization point between parallel and non-parallel portions of a query. The operations used to populate the buffer cannot be performed in parallel, whereas the operations that fetch rows from the buffer can be performed in parallel. The temporary buffer is required for the SQL Query Engine because the index scan and index probe operations are not considered to be SMP parallel enabled for this engine. Unlike the Classic Query Engine, which will perform these index operations in parallel, the SQL Query Engine will not subdivide the work necessary within the index operation to take full advantage of parallel processing. The buffer is used to allow a query to be processed under parallelism by serializing access to the index operations, while allowing any remaining work within the query to be processed in parallel.

A temporary buffer is an internal data structure and can only be created by the database manager.

Visual explain icon:




## Buffer Scan

The buffer scan operation is used when a query is processed using DB2 UDB Symmetric Multiprocessing, yet a portion of the query is not enabled to be processed under parallelism. The buffer scan acts as a gateway to control access to rows between the parallel enabled portions of the query and the non-parallel

| portions. Multiple threads can be used to fetch the selected rows from the buffer, allowing the query to  
| perform any remaining processing in parallel. However, the buffer will be populated in a non-parallel  
| manner.

| A buffer scan operation is identical to the list scan operation that is performed upon the temporary list  
| object. The main difference is that a buffer does not have to be fully populated prior to the start of the  
| scan operation. A temporary list requires that the list is fully populated prior to fetching any rows. See  
| "Temporary List" on page 31.  
|

Table 19. Buffer Scan Attributes

Data Access Method	Buffer Scan
Description	Sequentially scan and process all of the rows in the temporary buffer. Enables SMP parallelism to be performed over a non-parallel portion of the query.
Advantages	<ul style="list-style-type: none"> <li>The temporary buffer can be used to enable parallelism over a portion of a query that is non-parallel</li> <li>The temporary buffer does not have to be fully populated in order to start fetching rows</li> </ul>
Considerations	Generally used to prevent portions of the query from being processed multiple times when no key columns are required to satisfy the request.
Likely to be used	<ul style="list-style-type: none"> <li>When the query is attempting to take advantage of DB2 UDB Symmetric Multiprocessing</li> <li>When a portion of the query cannot be performed in parallel (for example, index scan or index probe)</li> </ul>
Example SQL statement	<pre>CHGQRYA DEGREE(*OPTIMIZE) CREATE INDEX X1 ON     Employee (LastName, WorkDept)  SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01' AND 'E01' AND LastName IN ('Smith', 'Jones', 'Peterson') OPTIMIZE FOR ALL ROWS</pre>
Messages indicating use	<ul style="list-style-type: none"> <li>Optimizer Debug: <pre>CPI4328 -- Access path of file X1 was used by query. CPI4330 -- 8 tasks used for parallel index scan of file EMPLOYEE.</pre> </li> <li>PRTSQLINF: <pre>SQL4027 -- Access plan was saved with DB2 UDB SMP installed on the system. SQL4008 -- Index X1 used for table 1. SQL4011 -- Index scan-key row positioning used on table 1. SQL4030 -- 8 tasks specified for parallel scan on table 1.</pre> </li> </ul>
SMP parallel enabled	Yes
Also referred to as	N/A
Visual Explain icon	

Using the example above, the optimizer chose to use the existing index X1 to perform an index probe operation against the table. In order to speed up the remaining processing for this query (for example, the Table Probe operation), DB2 Symmetric Multiprocessing will be used to perform the random probe into the table. Since the index probe operation is not SMP parallel enabled for the SQL Query Engine, that portion of the query is placed within a temporary buffer to control access to the selected index entries.

---

## Objects processed in parallel

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 (CPU and I/O) processors that share memory and disk resource work simultaneously toward 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.

The tables above indicate what data access method are enabled to take advantage of the DB2 UDB Symmetric Multiprocessing feature. An important thing to note, however, is that the parallel implementation differs for both the SQL Query Engine and the Classic Query Engine.

### Processing requirements

| Parallelism requires that SMP parallel processing must be enabled either by the system value  
| QQRVDEGREE, the query option file, or by the DEGREE parameter on the Change Query Attributes  
| (CHGQRYA) command. See “Control parallel processing for queries” on page 111 for information about  
| how to control parallel processing. Once parallelism has been enabled, a set of database system tasks or  
| threads 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.

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

---

## 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 table data space may be spread again by saving, deleting, and then restoring the table.

Maintaining an even distribution of data across all of the disk devices can lead to better throughput on query processing. The number of disk devices used and how the data is spread across these devices is taken into account by the optimizer while costing the different plan permutations.





---

## Chapter 6. Processing queries: 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 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 Chapter 5, “Data access on DB2 UDB for iSeries: data access paths and methods,” on page 13.  
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 can be used to implement the query and selects the most efficient technique.

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

---

### 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 can 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:

- “General query optimization tips”
- “Access plan validation” on page 50
- “Single table optimization” on page 50
- “Join optimization” on page 51
- “Grouping optimization” on page 61
- “Ordering optimization” on page 64
- “View implementation” on page 65
- “Materialized query table optimization” on page 67

### 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). See Chapter 8, “Creating an index strategy,” on page 121.

- 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.

Consider using the following options:

- Specify ALWCPYDTA(\*OPTIMIZE) to allow the query optimizer to create temporary copies of data so better performance can be obtained. The Client Access/400 ODBC driver and Query Management driver always uses this mode. If ALWCPYDTA(\*YES) is specified, the query optimizer will attempt to implement the query without copies of the data, but may create copies if required. If ALWCPYDTA(\*NO) is specified, copies of the data is not allowed. If the query optimizer cannot find a plan that does not use a temporary, then the query cannot be run.
- For SQL, use CLOSQLCSR(\*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.

## 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. Also, see “Plan Cache” on page 11 for additional information.

- 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.

The access plan is validated when the query is opened. Validation includes the following:

- Verifying that the same tables are referenced in the query as in the access plan. For example, the tables were not deleted and recreated or that the tables resolved by using \*LIBL have not changed.
- Verifying that the indexes used to implement the query, still exist.
- Verifying that the table size or predicate selectivity has not changed significantly.
- Verifying that QAQQINI options have not changed.

## Single table optimization

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 uses 2 costs when making decisions: an I/O cost and a CPU cost. The goal of the optimizer is to minimize both I/O and CPU cost.

### Optimizing Access to each table

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

- Determines the default filter factor for each predicate in the selection clause.
- Determines the true filter factor of the predicates by doing a key range estimate when the selection predicates match the left most keys of an index or by using columns statistic when available.
- 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 appropriate.
- Determines the cost of using existing indexes using Index Probe or Index Scan
  - Orders the indexes. For SQE, the indexes are ordered in general such that the indexes that access the smallest number of entries are examined first. For CQE, the indexes are generally ordered from mostly recently created to oldest.
  - For each index available, the optimizer does the following:
    - Determines if the index meets the selection criteria.
    - Determines the cost of using the index by estimating the number of I/Os and the CPU cost that will be needed to perform the Index Probe or the Index Scan and the possible Table Probes. See the Chapter 5, “Data access on DB2 UDB for iSeries: data access paths and methods,” on page 13 for additional information about Index Probe, Index Scan, and Table Probe.
    - Compares the cost of using this index with the previous cost (current best).
    - Picks the cheaper one.
    - Continues to search for best index until the optimizer decides to look at no more indexes.

For SQE, since the indexes are ordered so that the best indexes are examined first, once an index that is more expensive than the previously chosen best index, the search is ended.

For CQE, 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.

- Determines the cost of using a temporary bitmap
  - Orders the indexes that can be used for bitmapping. In general the indexes that select the smallest number of entries are examined first.
  - Determine the cost of using this index for bitmapping and the cost of merging this bitmap with any previously generated bitmaps.
  - If the cost of this bitmap plan is cheaper than the previous bitmap plan, continue searching for bitmap plans.
- After examining the possible methods of access the data for the table, the optimizer chooses the best plan from all the plans examined.

## 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
- Cost estimation and index selection for join secondary tables
- 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, a hash table, or a table scan (arrival sequence) on the secondary tables. In general, the join will be implemented using either an index or a hash table.

**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. Depending on the access to the first secondary table:
  - If using an index to access the secondary table, Radix Index Probe is used to locate the first row that satisfies the join condition for the first secondary table by using an index with keys matching the join condition or local row selection columns of the secondary table.
  - Applies bitmap selection, if applicable.

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.

If an efficient index cannot be found, a temporary index may be 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.

- If using a Hash Table Probe to access the secondary table, a hash temporary result table is created that contains all of the rows selected by local selection against the table on the first probe. 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.

A nested loop join using a Hash Table Probe has several advantages over a nested loop join using an Index Probe:

- 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 typically 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.
- If using a Sorted List Probe to access the secondary table, a sorted list result is created that contains all of the rows selected by local selection against the table on the first probe. The structure of the sorted list table is such that rows with the same join value are sorted together in the list. The location of the rows for any given join value can be found by probing using the join value.

- If using a table scan to access the secondary table, scan 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. The join may be implemented with a table scan when the secondary table is a user-defined table function.
4. 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 on page 52 through 4 are repeated until a row that satisfies both the join condition and any remaining selection is selected from all secondary tables
  5. Returns the result join row.
  6. 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.
  7. Ends processing when all selected rows from the primary table are processed.

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 the optimizer costs the join with that table as the primary and performing the grouping and ordering with an index.
- If ordering and grouping is specified on two or more tables or if temporaries are allowed, 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.

### Queries that cannot use hash join

Hash join cannot be used for queries that:

- 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 cannot be used for a table in a join query where the join condition something other than an equals operator.
- CQE does not support hash join if the query contains any of the following:
  - Subqueries unless all subqueries in the query can be transformed to inner joins.
  - UNION or UNION ALL
  - Perform left outer or exception join.
  - Use a DDS created join logical file.

### Join optimization algorithm

The query optimizer must determine the join columns, join operators, local row selection, dial implementation, 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



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.

**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
```

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
```

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 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 EMP\_ACT, PROJNO, EMP\_ACT or ...) the local row selection can be applied **at the same time** as the join selection (rather than before 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 OPNQRYP 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 and number of selected rows or both.
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 and number of selected rows or both.
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, 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 and in step 5, 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 a plan using a probe must be chosen.

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, unless there is an index available whose key definition is (A,B). Multi key field indexes allows the optimizer to detect when the values between columns are correlated. 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 (OPNQRYP %RANGE)
- 10% for each IN list value (OPNQRYP %VALUES)

For example, when the join operator is less-than, the expected number of matching rows is .33 \* (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.

## 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'
```



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 query optimizer generates a new predicate, whether 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.

### Look Ahead Predicate Generation

A special type of transitive closure called look ahead predicate generation (LPG) may be costed for joins. In this case, the optimizer attempts to minimize the random the I/O costs of a join by pre-applying the results of the query to a large fact table. LPG will typically be used with a class of queries referred to as star join queries, however it can possibly be used with any join query.

Look at the following query:

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

The optimizer will internally modify the query to be:

```
WITH HT AS (SELECT EMPLOYEE.EMPNO
FROM EMPLOYEE
WHERE EMPLOYEE.EMPNO='000010')

SELECT *
FROM HT, EMP_ACT
WHERE HT.EMPNO = EMP_ACT.EMPNO
AND EMP_ACT.EMPNO IN (SELECT DISTINCT EMPNO
FROM HT)
```

The optimizer will place the results of the "subquery" in a temporary hash table and then probe the hash table when selecting rows from EMP\_ACT which will remove any rows that would not join to EMPLOYEE. Since the rows from EMPLOYEE with any local selection applied, were placed in a temporary hash table, the join back to the EMPLOYEE table to get the resulting fan-out is done using the temporary hash table which should reduce I/O.

**Note:** LPG processing is only available in the SQL Query Engine. Classic Query Engine requires use of the STAR\_JOIN and FORCE\_JOIN\_ORDER QAQQINI options. See "Control queries dynamically with the query options file QAQQINI" on page 97 to tune star join queries.

### 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 8, "Creating an index strategy," on page 121 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. The optimizer does not generate transitive closure predicates between 2 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.

```

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.

```

## 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 SQE or 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 about 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 55 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 or column statistics 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 (for example, the data is not uniformly distributed).

### 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 20. 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 row selection columns or both. 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 or hash table.
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.

Table 20. Checklist for Creating an Application that Uses Join Queries (continued)

What to Do	How It Helps
Specify ALWCPYDTA(*OPTIMIZE) or ALWCPYDTA(*YES)	<p>If the query is creating a temporary index or hash table, and you feel that the processing time would be better if the optimizer only used the existing index or hash table, specify ALWCPYDTA(*YES).</p> <p>If the query is not creating a temporary index or hash table, 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 can also set n to a small number before ending the query.</p>
For OPNQRYF, specify OPTIMIZE(*FIRSTIO) or OPTIMIZE(*ALLIO)	Specify the OPTIMIZE(*FIRSTIO) or OPTIMIZE(*ALLIO) option to accurately reflect your application. Use *FIRSTIO, if you want the optimizer to optimize the query to retrieve the first block of rows most efficiently. This will bias the optimizer towards using existing objects. If you want to optimize the retrieval time for the entire answer set, use *ALLIO. This may cause the optimizer to create temporary objects such as temporary indexes or hash tables in order to minimize I/O.
Star join queries	<p>A join in which one table is joined with all secondary tables consecutively is sometimes called a <b>star join</b>. 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. For SQE, the optimizer uses Look Ahead Predicate generation to determine the optimal join order. For CQE, 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. 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 can 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>
Specify ALWCPYDTA(*OPTIMIZE) to allow the query optimizer to use a sort routine.	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.

Table 20. Checklist for Creating an Application that Uses Join Queries (continued)

What to Do	How It Helps
Specify join predicates to prevent all of the rows from one table from being joined to every row in the other table.	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.

## 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 Radix Index Scan or the Radix Index Probe 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 might 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 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 might 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
```

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;
```

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
```

The query optimizer will chose to use Index IX2. The database manager 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;
```

## 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 before 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)).

## 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 might eliminate an ordering column.



```

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

```

## 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 might 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

```

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'

```

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. The view reference in the user's query is then replaced with the temporary, and the query is run against the temporary result.

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
```

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

```

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

```

## Materialized query table optimization

Materialized query tables (MQTs) (also referred to as automatic summary tables or materialized views) can provide performance enhancements for queries. This is done by precomputing and storing results of a query in the materialized query table. The database engine can use these results instead of recomputing them for a user specified query. The query optimizer will look for any applicable MQTs and can choose to implement the query using a given MQT provided this is a faster implementation choice.

Materialized Query Tables are created using the SQL CREATE TABLE statement. Alternatively, the ALTER TABLE statement may be used to convert an existing table into a materialized query table. The REFRESH TABLE statement is used to recompute the results stored in the MQT. For user-maintained MQTs, the MQTs may also be maintained by the user via INSERT, UPDATE, and DELETE statements.

The support for creating and maintaining MQTs was shipped with the base V5R3 release of i5/OS™. The query optimizer support for recognizing and using MQTs is available with V5R3 i5/OS PTF SI17164 and the latest DB group PTF SF99503.

- “MQT supported function”
- “Using MQTs during Query optimization” on page 68
- “MQT examples” on page 68
- “Details on the MQT matching algorithm” on page 70
- “Summary of MQT query recommendations” on page 73
- “Notes on using MQTs” on page 73

### MQT supported function

Although a MQT can contain almost any query, the optimizer only supports a limited set of query functions when matching MQTs to user specified queries. The user specified query and the MQT query must both be supported by the SQE optimizer. (For a description of what is not supported by SQE, see “Query Dispatcher” on page 9.) The optimizer will only use one MQT per query. The supported function in the MQT query by the MQT matching algorithm includes:

- Single table queries and inner join queries
- WHERE clause
- GROUP BY and optional HAVING clauses
- ORDER BY
- FETCH FIRST n ROWS
- Views, common table expressions, and nested table expressions

For details, the MQT matching algorithm, see “Details on the MQT matching algorithm” on page 70.

Among the items not supported in the MQT query:

- subqueries and scalar subselects
- User Defined Functions (UDFs) and user defined table functions

- | • The DBPARTITIONNAME, ATAN2, DIFFERENCE, RADIANS, SOUNDEX, DLVALUE, DLURLPATH, DLURLPATHONLY, DLURLSEVER, DLURLSCHEME, DLURLCOMPLETE, DECRYPT\_BIT, DECRYPT\_BINARY, DECRYPT\_CHAR, DECRYPT\_DB, ENCRYPT\_RC2, GETHINT, DAYNAME, MONTHNAME, INSERT, REPEAT, and REPLACE scalar functions.
- | • Left outer, right outer, left exception, and right exception joins
- | • Unions
- | • Partitioned tables

| In addition, Partitioned MQTs are not supported.

| It is recommended that the MQT only contain references to columns, and column functions. In many environments, queries that contain constants will have the constants converted to parameter markers. This allows a much higher degree of ODP reuse. However the MQT matching algorithm cannot match constants and parameter markers or host variables.

### | **Using MQTs during Query optimization**

| To even consider using MQTs during optimization the following environmental attributes must be true:

- | • The query must specify ALWCOPYDTA(\*OPTIMIZE) or INSENSITIVE cursor.
- | • The query must not be a SENSITIVE cursor.
- | • The table to be replaced with a MQT must not be update or delete capable for this query.
- | • The MQT currently has the ENABLE QUERY OPTIMIZATION attribute active
- | • The MATERIALIZED\_QUERY\_TABLE\_USAGE QAQQINI option must be set to \*ALL or \*USER to enable use of MQTs. The default setting of MATERIALIZED\_QUERY\_TABLE\_USAGE does not allow usage of MQTs.
- | • The timestamp of the last REFRESH TABLE for a MQT is within the duration specified by the MATERIALIZED\_QUERY\_TABLE\_REFRESH\_AGE QAQQINI option or \*ANY is specified which allows MQTs to be considered regardless of the last REFRESH TABLE. The default setting of MATERIALIZED\_QUERY\_TABLE\_REFRESH\_AGE does not allow usage of MQTs.
- | • The query must be capable of being run through SQE.
- | • The following QAQQINI options must match: IGNORE\_LIKE\_REDUNDANT\_SHIFTS, NORMALIZE\_DATA, and VARIABLE\_LENGTH\_OPTIMIZATION. These options are stored at CREATE materialized query table time and must match the options specified at query run time.
- | • The commit level of the MQT must be greater than or equal to the query commit level. The commit level of the MQT is either specified in the MQT query using the WITH clause or it is defaulted to the commit level that the MQT was run under when it was created.

### | **MQT examples**

| The following are examples of using MQTs.

#### | **Example 1**

| The first example is a query that returns information on employees whose job is DESIGNER. The original query looks like this:

```
| Q1: SELECT D.deptname, D.location, E.firstnme, E.lastname, E.salary+E.comm+E.bonus as total_sal
|       FROM Department D, Employee E
|       WHERE D.deptno=E.workdept
|       AND E.job = 'DESIGNER'
```

| Create a table, MQT1, that uses this query:

```
| CREATE TABLE MQT1
|       AS (SELECT D.deptname, D.location, E.firstnme, E.lastname, E.salary, E.comm, E.bonus, E.job
|       FROM Department D, Employee E
```

```

| WHERE D.deptno=E.workdept)
| DATA INITIALLY IMMEDIATE REFRESH DEFERRED
| ENABLE QUERY OPTIMIZATION
| MAINTAINED BY USER

```

| Resulting new query after replacing the specified tables with the MQT.

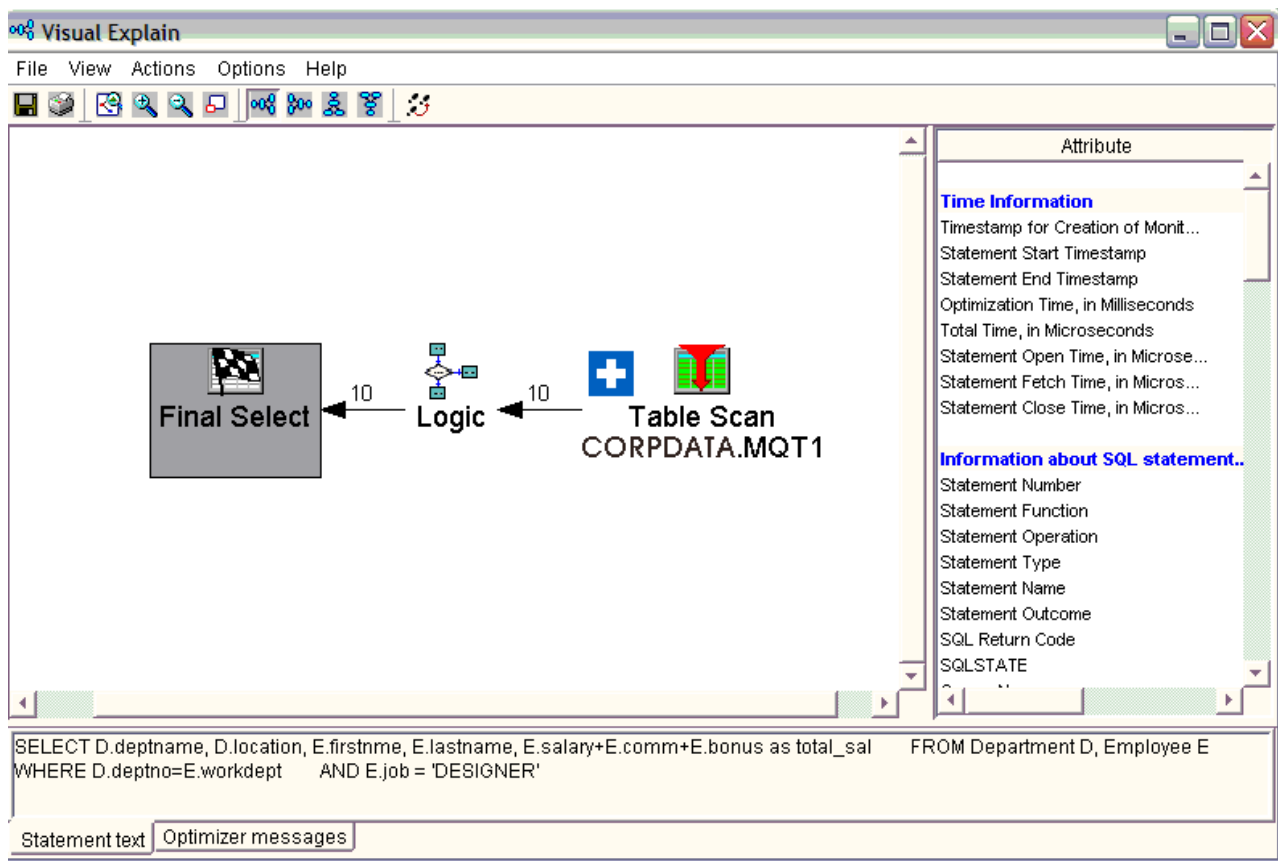
```

| SELECT M.deptname, M.location, M.firstnme, M.lastname, M.salary+M.comm+M.bonus as total_sal
| FROM MQT1 M
| WHERE M.job = 'DESIGNER'

```

| In this query, the MQT matches part of the user's query. The MQT is placed in the FROM clause and replaces tables DEPARTMENT and EMPLOYEE. Any remaining selection not done by the MQT query (M.job='DESIGNER') is done to remove the extra rows and the result expression, M.salary+M.comm+M.bonus, is calculated. Note that JOB must be in the select-list of the MQT so that the additional selection can be performed.

| Visual Explain diagram of the query when using the MQT:



### | Example 2

| Get the total salary for all departments that are located in 'NY'. The original query looks like this:

```

| SELECT D.deptname, sum(E.salary)
| FROM DEPARTMENT D, EMPLOYEE E
| WHERE D.deptno=E.workdept AND D.location = 'NY'
| GROUP BY D.deptname

```

| Create a table, MQT2, that uses this query:

```

| CREATE TABLE MQT2
|   AS (SELECT D.deptname, D.location, sum(E.salary) as sum_sal
| FROM DEPARTMENT D, EMPLOYEE E
| WHERE D.deptno=E.workdept
| GROUP BY D.Deptname, D.location)
| DATA INITIALLY IMMEDIATE REFRESH DEFERRED
| ENABLE QUERY OPTIMIZATION
| MAINTAINED BY USER

```

Resulting new query after replacing the specified tables with the MQT:

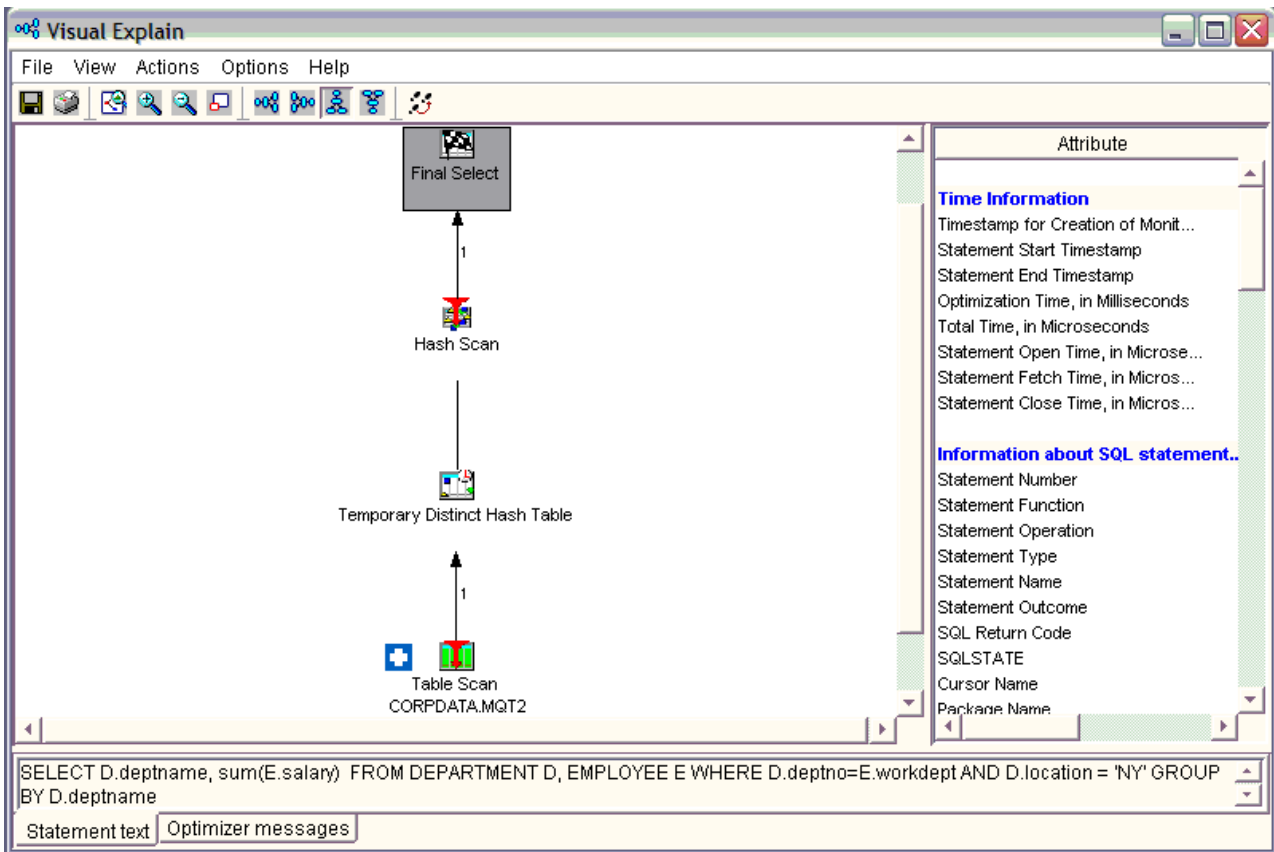
```

| SELECT M.deptname, sum(M.sum_sal)
| FROM MQT2 M
| WHERE M.location = 'NY'
| GROUP BY M.deptname

```

Since the MQT could potentially produce more groups than the original query, the final resulting query must group again and SUM the results to return the correct answer. Also the selection M.location='NY' must be part of the new query.

Visual Explain diagram of the query when using the MQT:



### Details on the MQT matching algorithm

What follows is a generalized discussion of how the MQT matching algorithm works.

- The tables specified in the query and the MQT are examined. If the MQT references tables not referenced in the query, then the MQT cannot be used. If the query references tables not specified in the MQT, then MQT can still potentially be used and the tables not specified in the MQT, remain in the rewritten query.

**Example 3:** The MQT contains less tables than the query:

```

| SELECT D.deptname, p.projname, sum(E.salary)
|   FROM DEPARTMENT D, EMPLOYEE E, EMPPROJACT EP, PROJECT P
|   WHERE D.deptno=E.workdept AND E.Empno=ep.empno
|         AND ep.projno=p.projno
|   GROUP BY D.DEPTNAME, p.projname

```

Create an MQT based on the query above:

```

| CREATE TABLE MQT3
|   AS (SELECT D.deptname, sum(E.salary) as sum_sal, e.workdept, e.empno
|   FROM DEPARTMENT D, EMPLOYEE E
|   WHERE D.deptno=E.workdept
|   GROUP BY D.Deptname, e.workdept, e.empno)
|   DATA INITIALLY IMMEDIATE REFRESH DEFERRED
|   ENABLE QUERY OPTIMIZATION
|   MAINTAINED BY USER

```

The rewritten query looks like this:

```

| SELECT M.deptname, p.projname, SUM(M.sum_sal)
|   FROM MQT3 M, EMPPROJACT EP, PROJECT P
|   WHERE M.Empno=ep.empno AND ep.projno=p.projno
|   GROUP BY M.deptname, p.projname

```

- All predicates specified in the MQT, must also be specified in the query. The query may contain additional predicates. Predicates specified in the MQT must match exactly the predicates in the query. Any additional predicates specified in the query, but not in the MQT must be able to be derived from columns projected from the MQT. See previous example 1 and discussion on “Parameter markers” on page 68.

**Example 4:** Set the total salary for all departments that are located in ‘NY’.

```

| SELECT D.deptname, sum(E.salary)
|   FROM DEPARTMENT D, EMPLOYEE E
|   WHERE D.deptno=E.workdept AND D.location = ?
|   GROUP BY D.Deptname

```

Create an MQT based on the query above:

```

| CREATE TABLE MQT4
|   AS (SELECT D.deptname, D.location, sum(E.salary) as sum_sal
|   FROM DEPARTMENT D, EMPLOYEE E
|   WHERE D.deptno=E.workdept AND D.location = 'NY'
|   GROUP BY D.deptname, D.location)
|   DATA INITIALLY IMMEDIATE REFRESH DEFERRED
|   ENABLE QUERY OPTIMIZATION
|   MAINTAINED BY USER

```

In this example, the constant ‘NY’ was replaced by a parameter marker and the MQT also had the local selection of location=‘NY’ applied to it when the MQT was populated. Since the MQT matching algorithm cannot match constants and parameter markers or host variables, MQT3 is not usable for the query.

Example 5:

```

| SELECT D.deptname, sum(E.salary)
|   FROM DEPARTMENT D, EMPLOYEE E
|   WHERE D.deptno=E.workdept AND D.location = 'NY'
|   GROUP BY D.deptname

```

Create an MQT based on the query above:

```

| CREATE TABLE MQT5
|   AS (SELECT D.deptname, E.salary
|   FROM DEPARTMENT D, EMPLOYEE E
|   WHERE D.deptno=E.workdept)
|   DATA INITIALLY IMMEDIATE REFRESH DEFERRED
|   ENABLE QUERY OPTIMIZATION
|   MAINTAINED BY USER

```

In this example, since D.Location is not a column of the MQT, the user query local selection predicate Location=‘NY’ cannot be determined, so the MQT cannot be used.



- If the MQT contains grouping, then the query must be a grouping query. The simplest case is where the MQT and the query specify the same list of grouping columns and column functions. In some cases if the MQT specifies a list of group by columns that is a superset of query group by columns, the query can be rewritten to do a step called regrouping. This will reaggregate the groups of the MQT, into the groups required by the query. When regrouping is required, the column functions need to be recomputed. The table below shows the supported regroup expressions.

The regroup new expression/aggregation rules are:

Table 21. Expression/aggregation rules for MQTs

Query	MQT	Final query
COUNT(*)	COUNT(*) as cnt	SUM(cnt)
COUNT(*)	COUNT(C2) as cnt2 (where c2 is non-nullable)	SUM(cnt2)
COUNT(c1)	COUNT(c1) as cnt	SUM(cnt)
COUNT(C1) (where C1 is non-nullable)	COUNT(C2) as cnt2 (where C2 is non-nullable)	SUM(cnt2)
COUNT(distinct C1)	C1 as group_c1 (where C1 is a grouping column)	COUNT(group_C1)
COUNT(distinct C1)	where C1 is not a grouping column	MQT not usable
COUNT(C2) where C2 is from a table not in the MQT	COUNT(*) as cnt	cnt*COUNT(C2)
COUNT(distinct C2) where C2 is from a table not in the MQT	N/A	COUNT(distinct C2)
SUM(C1)	SUM(C1) as sm	SUM(sm)
SUM(C1)	C1 as group_c1, COUNT(*) as cnt (where C1 is a grouping column)	SUM(group_c1 * cnt)
SUM(C2) where C2 is from a table not in the MQT	COUNT(*) as cnt	cnt*SUM(C2)
SUM(distinct C1)	C1 as group_c1 (where C1 is a grouping column)	SUM(group_C1)
SUM(distinct C1)	where C1 is not a grouping column	MQT not usable
SUM(distinct C2) where C2 is from a table not in the MQT	N/A	SUM(distinct C2)
MAX(C1)	MAX(C1) as mx	MAX(mx)
MAX(C1)	C1 as group_C1 (where C1 is a grouping column)	MAX(group_c1)
MAX(C2) where C2 is from a table not in the MQT	N/A	MAX(C2)
MIN(C1)	MIN(C1) as mn	MIN(mn)
MIN(C1)	C1 as group_C1 (where C1 is a grouping column)	MIN(group_c1)
MIN(C2) where C2 is from a table not in the MQT	N/A	MIN(C2)

AVG, STDDEV, and VAR\_POP are calculated using combinations of COUNT and SUM. If AVG, STDDEV, or VAR\_POP are included in the MQT and regroup requires recalculation of these functions, the MQT cannot be used. It is recommended that the MQT only use COUNT, SUM, MIN, and MAX. If the query contains AVG, STDDEV, or VAR\_POP, it can be recalculated using COUNT and SUM.



- If the `FETCH FIRST N ROWS` clause is specified in the MQT, then a `FETCH FIRST N ROWS` clause must also be specified in the query and the number of rows specified for the MQT must be greater than or equal to the number of rows specified in the query. It is not recommended that a MQT contain the `FETCH FIRST N ROWS` clause.
- The `ORDER BY` clause on the MQT can be used to order the data in the MQT if a `REFRESH TABLE` is run. It is ignored during MQT matching and if the query contains an `ORDER BY` clause, it will be part of the rewritten query.

### Summary of MQT query recommendations

- Do not include local selection or constants in the MQT because that limits the number of user specified queries that query optimizer can use the MQT in.
- For grouping MQTs, only use the `SUM`, `COUNT`, `MIN`, and `MAX` grouping functions. The query optimizer can recalculate `AVG`, `STDDEV`, and `VAR_POP` in user specified queries.
- Specifying `FETCH FIRST N ROWS` in the MQT limits the number of user specified queries that the query optimizer can use the MQT in and is not recommended.
- If the MQT is created with `DATA INITIALLY DEFERRED`, consider specifying the `DISABLE QUERY OPTIMIZATION` clause to prevent the query optimizer from using the MQT until it has been populated. When the MQT has been populated and is ready for use, the `ALTER TABLE` statement with the `ENABLE QUERY OPTIMIZATION` clause can be used to enable the MQT for the query optimizer.

### Notes on using MQTs

MQT tables need to be optimized just like non-MQT tables. Indexes should be created over the MQT columns that are used for selection, join and grouping as appropriate. Column statistics are collected for MQT tables.

The database monitor shows the list of MQTs considered during optimization. This information is in the 3030 record. If MQT usage has been enabled through the `QAQQINI` file and a MQT exists over at least one of the tables in the query, there will be a 3030 record for the query. Each MQT has a reason code indicating that it was used or if it was not used, why it was not used.



---

## Chapter 7. Optimizing query performance using query optimization tools

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

### Gather performance information about your queries

“Verify the performance of SQL applications”

“Examine query optimizer debug messages in the job log” on page 76

“Gather information about embedded SQL statements with the PRTSQLINF command” on page 77

“Monitoring your queries using Start Database Monitor (STRDBMON)” on page 78

“Monitoring your queries using memory-resident database monitor” on page 92

“View the implementation of your queries with Visual Explain” on page 94

“Collecting statistics with the Statistics Manager” on page 113

### Control the processing of your queries:

“Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command” on page 96

“Control queries dynamically with the query options file QAQQINI” on page 97

“Control long-running queries with the Predictive Query Governor” on page 107

“Control parallel processing for queries” on page 111

### Comparing the different tools:

You may want to check out the “Query optimization tools: Comparison table” on page 119 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

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

---

## 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 about printing information about SQL Statements, see the PRTSQLINF section in “Gather information about embedded SQL statements with the PRTSQLINF command” on page 77.

### **STRDBMON**

You can use the Start Database Monitor (STRDBMON) command to capture to a file information about every SQL statement that runs. See “Monitoring your queries using Start Database Monitor (STRDBMON)” on page 78 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. See “Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command” on page 96.

### **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 using STRDBG command:**

| STRDBG command puts a job into debug mode. It also specifies certain attributes of the debugging session. For example, it can specify whether database files in production schemas can be updated while in debug mode. For example, use the following command:

| STRDBG PGM(Schema/program) UPDPROD(\*YES)

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

### Viewing debug messages using QAQQINI table:

You can also set the QRYOPTLIB parameter on the Change Query Attributes (CHGQRYA) command to a user schema where the QAQQINI table exists. Set the parameter on the QAQQINI table to MESSAGES\_DEBUG, and set the value to \*YES. This option places query optimization information in the job log. Changes made to the QAQQINI table are effective immediately and will affect all users and queries that use this table. Once you change the MESSAGES\_DEBUG parameter, all queries that use this QAQQINI table will write debug messages to their respective joblogs. 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. For more information about using QAQQINI, see “Control queries dynamically with the query options file QAQQINI” on page 97.

### Viewing debug messages in Run SQL Scripts:

To view debug messages in Run SQL Scripts, from the **Options** menu, select **Include Debug Messages in Job Log**. Then from the **View** menu, select **Job Log**. To view detailed messages, double-click a message.

### Viewing debug messages in Visual Explain:

In Visual Explain, debug messages are always available. You do not need to turn them on or off. Debug messages appear in the lower portion of the window. You can view detailed messages by double-clicking on a message. For more information about Visual Explain, see “View the implementation of your queries with Visual Explain” on page 94.

See “Query optimization performance information messages” on page 271 and “Query optimization performance information messages and open data paths” on page 293 for the specific meanings of the debug messages.

---

## Gather information about embedded SQL statements with the PRTSQLINF command

The PRTSQLINF command returns 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. This information is then stored 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.

This output is similar to the information that you can get from debug messages. However, while query debug messages work at runtime, PRTSQLINF works retroactively. You can also see this information in the second level text of the query governor inquiry message CPA4259. To view the messages, see “PRTSQLINF message reference” on page 300.

You can issue PRTSQLINF in a couple of ways. First, you can run the PRTSQLINF command against a saved access plan. 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.

You can also run PRTSQLINF against functions, stored procedures, triggers, SQL packages, and programs from iSeries Navigator. This function is called Explain SQL. To view PRTSQLINF information, right-click an object and select **Explain SQL**.

---

## Monitoring your queries using Start Database Monitor (STRDBMON)

Start Database Monitor (STRDBMON) gathers information about a query in read time and stores this information in an output table. This information can help you determine whether your system and your queries are performing as they should, or whether they need fine tuning. Database monitors can generate significant CPU and disk storage overhead when in use. You can gather performance information for a specific query or for every query on 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. For examples on using the database monitor, see "Database monitor examples" on page 87.

You can start the monitor by using one of the following methods:

- Use the "Start Database Monitor (STRDBMON) command" on page 79 and the "End Database Monitor (ENDDDBMON) command" on page 79
- Use the "Using iSeries Navigator to start STRDBMON" on page 81
- Use the Start Performance Monitor (STRPFRMON) command with the STRDBMON parameter

For information about the rows in the output table, see "Database monitor performance rows" on page 80. This monitor also gives information about creating indexes. For more information, see "Query optimizer index advisor" on page 86. For examples using the STRDBMON, see "Database monitor examples" on page 87.

### 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
- Might perform faster with the creation of a keyed logical file containing keys suggested by the query optimizer.

**Note:** A query that is canceled 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 member or both does not exist, one is created based upon the table and format definition of model table QSYS/QAQQDBMN. If the output table and member or both 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 "Database monitor: DDS" on page 155. 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 wanted. The logical files can be created with the DDS shown in “Optional database monitor logical file DDS” on page 162. 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 might be quite large. The logicals are not required to process the results. They 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 (DYNSLT 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.

## Using iSeries Navigator to start STRDBMON

The iSeries Navigator version of the STDBMON is called a detailed SQL performance monitor. You can start this monitor by right-clicking SQL Performance Monitors under the database portion of the iSeries Navigator tree and selecting **New-> Detailed**. You can monitor a single query or all queries. Once the monitor is started, it appears in the SQL performance monitors list.

SQL Performance monitors provides several predefined reports that you can use to analyze your monitor data. To view these reports, right-click a monitor and select **Analyze**. The monitor does not need to be ended in order to view this information. Additionally, you can import a monitor that has been started by using STRDBMON or some other interface. Right-click SQL Performance monitors and select **Import**. You can then use the same predefined reports to analyze your monitor data.

The following is an overview of the information that you can obtain from the predefined reports.

### General Summary

Contains information that summarizes all SQL activity. This information provides the user with a high level indication of the nature of the SQL statements used. For example, how much SQL is used in the application? Are the SQL statements mainly short-running or long running? Is the number of results returned small or large?

### Job Summary

Contains a row of information for each job. Each row summarizes all SQL activity for that job. This information can be used to tell which jobs on the system are the heaviest users of SQL, and hence which ones are perhaps candidates for performance tuning. The user may then want to start a separate detailed performance monitor on an individual job to get more detailed information without having to monitor the entire system.

### Operation Summary

Contains a row of summary information for each type of SQL operation. Each row summarizes all SQL activity for that type of SQL operation. This information provides the user with a high level indication of the type of SQL statements used. For example, are the applications mainly read-only, or is there a large amount of update, delete, or insert activity. This information can then be used to try specific performance tuning techniques. For example, if a large amount of INSERT activity is occurring, perhaps using an OVRDBF command to increase the blocking factor or perhaps use of the QDBENCWT API is appropriate.

### Program Summary

Contains a row of information for each program that performed SQL operations. Each row summarizes all SQL activity for that program. This information can be used to identify which programs use the most or most expensive SQL statements. Those programs are then potential candidates for performance tuning. Note that a program name is only available if the SQL statements are embedded inside a compiled program. SQL statements that are issued through ODBC, JDBC, or OLE DB have a blank program name unless they result from a procedure, function, or trigger.

### SQL Attributes Summary

Contains a summary of the optimization attributes. This option is only available when you use a detailed SQL performance monitor. This information provides the user with a high level indication of some of the key SQL attributes used. This can help identify attributes that potentially are more

costly than others. For example, in some cases ALWCPYDTA(\*YES) can allow the optimizer to run a query faster if live data is not needed by the application. Also, \*ENDMOD and \*ENDPGM are much more expensive than \*ENDJOB or \*ENDACTGRP.

### **Isolation Level Summary**

Contains a summary of how many statements were run under each isolation level. This option is only available when you use a detailed SQL performance monitor. This information provides the user with a high level indication of the isolation level used. The higher the isolation level, the higher the chance of contention between users. For example, a high level of Repeatable Read or Read Stability use will likely produce a high level of contention. The lowest level isolation level that still satisfies the application design requirement should always be used.

### **Error Summary**

Contains a summary of any SQL statement error messages or warnings that were captured by the monitor

### **Statement Use Summary**

Contains a summary of how many statements are executed and the number of times they are executed during the performance monitor collection period. This option is only available when you use a detailed SQL performance monitor. This information provides the user with a high level indication of how often the same SQL statements are used multiple times. If the same SQL statement is used more than once, it may be cached and subsequent uses of the same statement are less expensive. It is more important to tune an SQL statement that is executed many times than an SQL statement that is only run once.

### **Open Summary**

Contains a summary of how many statements perform an open the number of times they are executed during the performance monitor collection period. This option is only available when you use a detailed SQL performance monitor. This information provides the user with a high level indication of how often an ODP is reused. The first open of a query in a job is a full open. After this, the ODP may be pseudo-closed and then reused. An open of a pseudo-closed ODP is far less expensive than a full open. The user can control when and ODP is pseudo-closed and how many pseudo-closed ODPs are allowed in a job by using the Change Query Attributes action in the Database Folder of iSeries Navigator, the CHGQRYA CL command, or the QQRYDEGREE system value. In rare cases, an ODP is not reusable. High usage of non-reusable ODPs may indicate that the SQL statements causing the non-reusable ODPs should be rewritten.

### **Data Access Summary**

Contains a summary of the number of SQL statements that are read-only versus those that modify data. This option is only available when you use a detailed SQL performance monitor. This information provides the user with a less detailed view of the type of SQL statements used than that available through the Operation Summary. This information can then be used to try specific performance tuning techniques. For example, if a large amount of INSERT activity is occurring, perhaps using an OVRDBF command to increase the blocking factor or perhaps use of the QDBENCWT API is appropriate.

### **Statement Type Summary**

Contains a summary of whether SQL statements are in extended dynamic packages, system-wide statement cache, regular dynamic, or static SQL statements. This option is only available when you use a detailed SQL performance monitor. This information provides the user with a high level indication of how many of the SQL statements were fully parsed and optimized (dynamic) or whether the SQL statements and access plans were stored either statically in a program, procedure, function, package, or trigger. An SQL statement that must be fully parsed and optimized is more expensive than the same statement that is static, extended dynamic, or cached in the system-wide statement cache.

### **Parallel Processing Summary**

Contains a summary of the parallel processing techniques used. This option is only available when you use a detailed SQL performance monitor. This information provides the user with a high level indication of whether one of the many parallel processing techniques were used to execute the SQL statements. Most parallel processing techniques are only available if the Symmetric Processing for iSeries is installed. Once the option is installed, the degree of parallelism must be specified by the user through the Change Query Attributes action in the Database Folder of iSeries Navigator, the CHGQRYA CL command, or the QQRYDEGREE system value.

### **Optimizer Summary**

Contains a summary of the optimizer techniques used. This option is only available when you use a detailed SQL performance monitor. This information provides the user with a high level indication of the types of queries and optimizer attributes used. This information can be used to determine whether the types of queries are complex (use of subqueries or joins) and identify attributes that may deserve further investigation. For example, an access plan rebuild occurs when the prior access plan is no longer valid or if a change has occurred that identified a better access plan. If the number of access plan rebuilds is high, it may indicate that some application redesign may be necessary. Also, if the join order has been forced, this may indicate that the access plan chosen may not be the most efficient. However, it may also indicate that someone has already tuned the SQL statement and explicitly forced the join order because experimentation showed that a specific join order should always provide the best order. Forcing the join order should be used sparingly. It prevents the optimizer from analyzing any join order other than the one specified.

Additionally, you can select more Detailed Results:

#### **Basic statement information**

This information provides the user with basic information about each SQL statement. The most expensive SQL statements are presented first in the list so at a glance the user can see which statements (if any) were long running.

#### **Access plan rebuild information**

Contains a row of information for each SQL statement that required the access plan to be rebuilt. Re-optimization will occasionally be necessary for one of several reasons such as a new index being created or dropped, the apply of a PTF, and so on. However, excessive access plan rebuilds may indicate a problem.

#### **Optimizer information**

Contains a row of optimization information for each subselect in an SQL statement. This information provides the user with basic optimizer information about those SQL statements that involve data manipulation (Selects, opens, updates, and so on) The most expensive SQL statements are presented first in the list.

#### **Index create information**

Contains a row of information for each SQL statement that required an index to be created. Temporary indexes may need to be created for several reasons such as to perform a join, to support scrollable cursors, to implement ORDERBY or GROUP BY, and so on. The created indexes may only contain keys for rows that satisfy the query (such indexes are known as sparse indexes). In many cases, the index create may be perfectly normal and the most efficient way to perform the query. However, if the number of rows is large, or if the same index is repeatedly created, you may be able to create a permanent index to improve performance of this query. This may be true whether an index was advised.

#### **Index used information**

Contains a row of information for each permanent index that an SQL statement used. This can be used to quickly tell if any of the permanent indexes were used to improve the performance of a query. Permanent indexes are typically necessary to achieve optimal query performance. This

information can be used to determine how often a permanent index was used by in the statements that were monitored. Indexes that are never (or very rarely) used should probably be dropped to improve the performance of inserts updates and deletes to a table. Before dropping the index you may also want to look at the last used date in the Description information for the index.

#### **Open information**

Contains a row of information for each open activity for each SQL statement. The first time (or times) a open occurs for a specific statement in a job is a full open. A full open creates an Open Data Path (ODP) that will be then be used to fetch, update, delete, or insert rows. Since there will typically be many fetch, update, delete, or insert operations for an ODP, as much processing of the SQL statement as possible is done during the ODP creation so that same processing does not need to be done on each subsequent I/O operation. An ODP may be cached at close time so that if the SQL statement is run again during the job, the ODP will be reused. Such an open is called a pseudo open and is much less expensive than a full open. You can control the number of ODPs that are cached in the job and then number of times the same ODP for a statement should be created before caching it.

#### **Index advised information**

Provides information about advised indexes. This option is only available when you use a detailed SQL performance monitor. This can be used to quickly tell if the optimizer recommends creating a specific permanent index to improve performance. While creating an index that is advised will typically improve performance, this is not a guarantee. Once the index is created, much more accurate estimates of the actual costs are available. The optimizer may decide based on this new information that the cost of using the index is too high. Even if the optimizer does not use the index to implement the query, the new estimates available from the new index provides more detailed information to the optimizer that may produce better performance

#### **Governor time out information**

Provides information about all optimizer time outs. This option is only available when you use a detailed SQL performance monitor. This can be used to determine how often users are attempting to run queries that exceed the governor time out value. A large number of governor time outs may indicate that the time out value is set too low.

#### **Optimizer time out information**

Provides information about any optimizer time outs. This option is only available when you use a detailed SQL performance monitor. Choosing the best access plan for a very complex query can be time consuming. As the optimizer evaluates different possible access plans a better estimate of how long a query takes shape. At some point, for dynamic SQL statements, the optimizer may decide that further time spent optimizing is no longer reasonable and use the best access plan up to that point. This may or may not be the best plan available. If the SQL statement is only run a few times or if the performance of the query is good, an optimizer time out is not a concern. However, if the SQL statement is long running or if it is run many times and the optimizer times out, a better plan may be possible by enabling extended dynamic package support or by using static SQL in a procedure or program. Since many dynamic SQL statements can be cached in the system-wide statement cache, optimizer timeouts are not common.

#### **Procedure call information**

Provides information about Procedure call usage. This option is only available when you use a detailed SQL performance monitor. Performance of client/server or web-based applications is best when the number of round trips between the client and the server is minimized, because the total communications cost is minimized). One common way to accomplish this is to call a procedure that will perform a number of operations on the server before returning results, rather than sending each individual SQL statement to the server.

#### **Hash table information**

Provides information about any temporary has tables that were used. This option is only available

when you use a detailed SQL performance monitor. Hash join and hash group may be chosen by the optimizer to perform an SQL statement because it will result in the best performance. However, hashing can use a significant amount of temporary storage. If the hash tables are very large, and several users are performing hash joins or group by at the same time, the total resources necessary for the hash tables may become a problem.

#### **Distinct processing information**

Provides information about any DISTINCT processing. This option is only available when you use a detailed SQL performance monitor. SELECT DISTINCT in an SQL statement may be a time consuming operation because a final sort may be necessary of the result set to eliminate duplicate rows. DISTINCT in long running SQL statements should only be used if it is absolutely necessary to eliminate the duplicate resulting rows.

#### **Table scan**

Contains a row of information for each subselect that required records to be processed in arrival sequence order. Table scans of large tables can be time-consuming. If the SQL statement is long running, it may indicate that an index might be necessary to improve performance.

#### **Sort information**

Contains a row of information for each sort that an SQL statement performed. Sorts of large result sets in an SQL statement may be a time consuming operation. In some cases, an index can be created that will eliminate the need for a sort.

#### **Temporary file information**

Contains a row of information for each SQL statement that required a temporary result. Temporary results are sometimes necessary based on the SQL statement. If the result set inserted into a temporary result is large, you may want to investigate why the temporary result is necessary. In some cases, the SQL statement can be modified to eliminate the need for the temporary result. For example, if a cursor has an attribute of INSENSITIVE, a temporary result will be created. Eliminating the keyword INSENSITIVE will typically remove the need for the temporary result, but your application will then see changes as they are occur in the database tables.

#### **Data conversion information**

Contains a row of information for each SQL statement that required data conversion. For example, if a result column has an attribute of INTEGER, but the variable the result is being returned to is DECIMAL, the data must be converted from integer to decimal. A single data conversion operation is very inexpensive, but repeated thousands or millions of times can add up. In some cases, it is a simple task to change one of the attributes so a faster direct map can be performed. In other cases, the conversion is necessary because there is no exact matching data type available.

#### **Subquery information**

Contains a row of subquery information. This information can indicate which subquery in a complex SQL statement is the most expensive.

#### **Row access information**

Contains information about the rows returned and I/Os performed. This option is only available when you use a detailed SQL performance monitor. This information can indicate the amount of I/Os that occur for the SQL statement. A large number of physical I/Os can indicate that perhaps a larger pool is necessary or perhaps SETOBJACC may be used to pre-bring some of the data into main memory.

#### **Lock escalation information**

Provides information about any lock escalation. This option is only available when you use a detailed SQL performance monitor. In a few rare cases, a lock must be escalated to the table level



instead of the row level. This can cause much more contention or lock wait time outs between a user that is modifying the table and the reader of the table. A large number of lock escalation entries may indicate a contention performance problem.

#### **Bitmap information**

Provides information about any bitmap creates or merges. This option is only available when you use a detailed SQL performance monitor. Bitmap generation is typically used when performing indexing or joining. This typically is a very efficient mechanism.

#### **Union merge information**

Provides information about any union operations. This option is only available when you use a detailed SQL performance monitor. UNION is a more expensive operation than UNION ALL because duplicate rows must be eliminated. If the SQL statement is long running, make sure it is necessary that the duplicate rows be eliminated.

#### **Group by information**

Provides information about any GROUP BY operations. This option is only available when you use a detailed SQL performance monitor.

#### **Error information**

Provides information about any SQL statement error messages and warnings that were captured by the monitor.

#### **Start and end monitor information**

Provides information about any start and end database monitor operations. This option is only available when you use a detailed SQL performance monitor.

There is also an Extended Detailed Results reports that provide essentially the same information as above, but with added levels of details.

## **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 improves performance. If the optimizer determines that a permanent index might 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 QQIDXD. 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 QQIDXD. The intention of creating this index is to improve the performance of the query.

You can also view the Query optimizer index advisor recommendations from Visual Explain. See "Overview of information available from Visual Explain" on page 95 for details.

In the list of key columns contained in column QQIDXD 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 QQIDX. 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 QQIDX 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 162. 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 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 can 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.QQUCNT = B.QQUCNT
AND    A.QQJFLD = C.QQJFLD AND A.QQUCNT = C.QQUCNT
```

Sample output of this query is shown in Table 22 on page 88. Key to this example are the join criteria:

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

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 22. 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.QQR0WR,
       (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 23.

**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 23. 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
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 can be used for this:

```

SELECT A.QQTLN, A.QQTFN, A.QQIDXA, A.QQIDXD,
       A.QQIDXK, 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 24 shows what the results might look like.

Table 24. 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 satisfies 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 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 need 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.QQIDXD, 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.QQIDXD,
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 can 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, QQPID
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.QQPID,
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.

---

## Monitoring your queries using memory-resident database monitor

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 set of APIs, takes database monitoring information 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)
- schema 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 schema 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 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 about the database monitor APIs:

- | • “Memory-resident database monitor external API description”
- | • “Memory-resident database monitor external table description”
- | • “Sample SQL queries” on page 94
- | • “Memory-resident database monitor row identification” on page 94

| You can also start a DBMon monitor from iSeries Navigator. For information using iSeries Navigator with monitors, see “Using iSeries Navigator to start STRDBMON” on page 81.

## 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 25. 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 26. External table Description*

QAQQQRYI	Query (SQL) information
----------	-------------------------

Table 26. External table Description (continued)

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).

---

## View the implementation 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. To launch Visual Explain, see "Starting Visual Explain" on page 95.



Queries are displayed using a graph with a series of icons that represent different operations that occur during implementation. This graph is displayed in the main window. In the lower portion of the pane, the SQL statement that the graph is based on is displayed. If Visual explain is started from Run SQL Scripts, you can view the debug messages issued by the optimizer by clicking the **Optimizer messages** tab. The Query attributes are displayed in the right pane. For more detail about the information available from Visual Explain, see "Overview of information available from Visual Explain."

## Starting Visual Explain

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, QQQVEXPL. You can start Visual Explain from any of the following windows in iSeries Navigator:

- 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.
- Expand the list of available SQL Performance Monitors. Right-click a detailed 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 want to explain and click **Run Visual Explain**. You can also start an SQL Performance Monitor from Run SQL Scripts. Select **Start SQL Performance monitor** from the **Monitor** menu.
- Start the Current<sup>®</sup> SQL for a Job function by right-clicking **Databases** and select **Current SQL for a Job**. Select a job from the list and click **SQL Statement**. When the SQL is displayed in the lower pane, you can start Visual Explain by clicking **Run Visual Explain**.

In addition, a database monitor table that was not created as a result of using iSeries Navigator can be explained through iSeries Navigator. First you must import the database monitor table into iSeries Navigator. To do this, right-click 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 represents the file-based (STRDBMON) monitor while Summary represents the memory-resident monitor (which is not supported by Visual Explain). Once the monitor has been imported, follow the steps to start Visual Explain from within iSeries Navigator.

**Note:** When using the Explain only option of Visual Explain from Run SQL Scripts in iSeries Navigator, some queries receive an error code 93 stating that they are too complex for displaying in Visual Explain. You can circumvent this by selecting the "Run and Explain" option.

You can save your Visual Explain information as an SQL Performance monitor, which can be useful if you started the query from Run SQL Scripts and want to save the information for later comparison. Select **Save as Performance monitor** from the **File** menu.

## Overview of information available from Visual Explain

You can use Visual Explain to view many types of information:

- Information about each operation (icon) in the query graph
- Highlight expensive icons
- The statistics and index advisor
- The predicate implementation of the query
- Basic and detailed information in the graph

**Information about each operation (icon) in the query graph:** As stated before, the icons in the graph represent operations that occur during the implementation of the query. The order of operations is shown by the arrows connecting the icons. If parallelism was used to process an operation, the arrows are doubled. Occasionally, the optimizer "shares" hash tables with different operations in a query, causing the lines of the query to cross.

| You can view information about an operation by selecting the icon. Information is displayed in the  
| **Attributes** table in the right pane. To view information about the environment, click an icon and then  
| select **Display query environment** from the **Action** menu. Finally, you can view more information about  
| the icon by right-clicking the icon and selecting **Help**.

| **Highlight expensive icons:** You can highlight problem areas (expensive icons) in your query using Visual  
| Explain. Visual Explain offers you two types of expensive icons to highlight: by processing time or  
| number of rows. You can highlight icons by selecting **Highlight expensive icons** from the **View** menu.

| **The statistics and index advisor:** During the implementation of a query, the optimizer can determine if  
| statistics need to be created or refreshed, or if an index could make the query run faster. You can view  
| these recommendations using the Statistics and Index Advisor from Visual Explain. Start the advisor by  
| selecting **Advisor** from the **Action** menu. Additionally, you can begin collecting statistics or create an  
| index directly from the advisor. See “Query optimizer index advisor” on page 86 for more information  
| about the index advisor.

| **The predicate implementation of the query:** Visual explain allows you to view the implementation of  
| query predicates. Predicate implementation is represented by a blue plus sign next to an icon. You can  
| expand this view by right-clicking the icon and selecting **Expand**. or open it into another window. Click  
| an icon to view attributes about the operation. To collapse the view, right-click anywhere in the window  
| and select **Collapse**. This function is only available on V5R3 or later systems.

| The optimizer can also use the Look Ahead Predicate Generation to minimize the random the I/O costs  
| of a join. To highlight predicates that used this method, select **Highlight LPG** from the **View** menu. See  
| “Look Ahead Predicate Generation” on page 57 for details.

| **Basic and full information in the graph:** Visual Explain also presents information in two different views:  
| basic and full. The basic view only shows those icons that are necessary to understand how the SQL  
| statement was executed, thus excluding some preliminary or intermediate operations that are not  
| essential for understanding the main flow of query implementation. The full view may show more icons  
| that further depict the flow of the execution tree. You can change the graph detail by select **Graph Detail**  
| from the **Options** menu and selecting either **Basic** or **Full**. The default view is basic. Note that in order to  
| see all of the detail for a **Full** view, you will need to change the Graph Detail to **Full**, close out Visual  
| Explain, and run the query again. The setting for Graph Detail will persist.

| For more information about Visual Explain and the different options that are available, see the Visual  
| Explain online help.

---

## 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 Change Query Attributes (CHGQRYA) CL 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 canceled, 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.

- | You can find more details about modifying your query attributes in the following topics:
  - | • “Control queries dynamically with the query options file QAQQINI”
  - | • “Control long-running queries with the Predictive Query Governor” on page 107
  - | • “Control parallel processing for queries” on page 111
- | Asynchronous job and Apply CHGQRYA to remote is discussed in the Changes to the change query attributes (CHGQRYA) command with DB2 Multisystem topic in the *DB2 Multisystem* information
- | Finally, you can find reference information for the Change Query Attributes (CHGQRYA) CL command in the *Programming* topic of the Information Center.

---

## 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 database manager. For each query that is run the query option values are retrieved from the QAQQINI file in the schema 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\_DERIVED\_INDEX
- | • IGNORE\_LIKE\_REDUNDANT\_SHIFTS
- | • LOB\_LOCATOR\_THRESHOLD
- | • MATERIALIZED\_QUERY\_TABLE\_REFRESH\_AGE
- | • MATERIALIZED\_QUERY\_TABLE\_USAGE
- | • MESSAGES\_DEBUG
- | • NORMALIZE\_DATA
- | • 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
- | • VARIABLE\_LENGTH\_OPTIMIZATION

To specify the schema that currently holds or will contain the query options file QAQQINI, see “Specifying the QAQQINI file.”

To create your own QAQQINI file, see “Creating the QAQQINI query options file.”

For a complete listing of the QAQQINI file, see “QAQQINI query options” on page 100.

## Specifying the QAQQINI file

Use the CHGQRYA command with the QRYOPTLIB (query options library) parameter to specify which schema currently contains or will contain the query options file QAQQINI. The query options file will be retrieved from the schema specified on the QRYOPTLIB parameter for each query and remains in effect for the duration of the job or user session, or until the QRYOPTLIB parameter is changed by the CHGQRYA command.

If the CHGQRYA command is not issued or is issued but the QRYOPTLIB parameter is not specified, the schema 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 an indication that a QAQQINI file that contains all default values is being used. The initial value of the QRYOPTLIB parameter for a job is QUSRSYS.

## Creating the QAQQINI query options file

Each server is shipped with a QAQQINI template file in schema 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 schema that will be specified on the CHGQRYA QRYOPTLIB 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
A          R QAQQINI
A          QQPARM      256A
                                     UNIQUE
                                     TEXT('Query options + file')
                                     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 schema QSYS has been pre-populated with the following rows:

Table 27. QAQQINI File Records. Description

QQPARAM	QQVAL
APPLY_REMOTE	*DEFAULT
ASYNC_JOB_USAGE	*DEFAULT
COMMITMENT_CONTROL_LOCK_LIMIT	*DEFAULT
FORCE_JOIN_ORDER	*DEFAULT
IGNORE_DERIVED_INDEX	*DEFAULT
IGNORE_LIKE_REDUNDANT_SHIFTS	*DEFAULT
LOB_LOCATOR_THRESHOLD	*DEFAULT
MESSAGES_DEBUG	*DEFAULT
NORMALIZE_DATA	*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
VARIABLE_LENGTH_OPTIMIZATION	*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 QQVAL='*YES'  
WHERE QPPARM='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 QPPARM='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 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.
```

## QAQQINI query options

The following table summarizes the query options that can be specified on the QAQQINI command:

Table 28. Query Options Specified on QAQQINI Command

Parameter	Value	Description
APPLY_REMOTE	*DEFAULT	The default value is set to *YES.
	*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.
COMMITMENT_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.
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 before 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.

Table 28. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
IGNORE_DERIVED_INDEX	*DEFAULT	The default value is the same as *NO.
	*YES	<p>Allow the SQE optimizer to ignore the derived index and process the query. The resulting query plan will be created without any regard to the existence of the derived index(s). The index types that are ignored include:</p> <ul style="list-style-type: none"> <li>• Keyed logical files defined with select or omit criteria and with the DYNSTL keyword omitted</li> <li>• Keyed logical files built over multiple physical file members (V5R2 restriction, not a restriction for V5R3)</li> <li>• Keyed logical files where one or more keys reference an intermediate derivation in the DDS. Exceptions to this are: 1. when the intermediate definition is simply defining the field in the DDS so that shows up in the logical's format and 2. RENAME of a field (these two exceptions do not make the key derived)</li> <li>• Keyed logical files with K *NONE specified.</li> <li>• Keyed logical files with Alternate Collating Sequence (ACS) specified</li> <li>• SQL indexes created when the sort sequence active at the time of creation requires a weighting (translation) of the key to occur. This is true when any of several non-US language IDs are specified. It also occurs if language ID shared weight is specified, even for language US.</li> </ul>
	*NO	Do not ignore the derived index. If a derived index exists, have CQE process the query.
IGNORE_LIKE_REDUNDANT_SHIFTS	*DEFAULT	The default value is set to *OPTIMIZE.
	*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.
LOB_LOCATOR_THRESHOLD	*DEFAULT	The default value is set to 0. This indicates that the database will take no action to free locators.
	Integer Value	If the value is 0, then the database will take no action to free locators. For values 1 through 250,000, on a FETCH request, the database will compare the active LOB locator count for the job against the threshold value. If the locator count is greater than or equal to the threshold, the database will free host server created locators that have been retrieved. This option applies to all host server jobs (QZDASOINIT) and has no impact to other jobs.

Table 28. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
MATERIALIZED_QUERY_TABLE_REFRESH_AGE:	*DEFAULT	The default value is *NONE.
	*NONE	Materialized Query Tables may not be used in query optimization and implementation.
	*ALL	User-maintained refresh-deferred query tables may be used.
	*USER	User-maintained materialized query tables may be used.
MATERIALIZED_QUERY_TABLE_USAGE:	*DEFAULT	The default value is 0.
	0	No materialized query tables may be used.
	*ANY	Any tables indicated by the MATERIALIZED_QUERY_TABLE_USAGE QAQQINI parameter may be used. Equivalent to specifying 9999 99 99 99 99 99 (which is 9999 years, 99 months, 99 days, 99 hours, 99 minutes, 99 seconds). If the materialized query table has never been refreshed by the REFRESH TABLE SQL statement, but the table should be considered, then the MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option must be set to *ANY.
	Timestamp_duration	Only tables indicated by the MATERIALIZED_QUERY_TABLE_USAGE QAQQINI option which have a REFRESH TABLE performed within the specified timestamp duration will be used. This is a DECIMAL(20,6) number which indicates a timestamp duration since the last REFRESH TABLE was done.
MESSAGES_DEBUG	*DEFAULT	The default is set to *NO.
	*NO	No debug messages are to be displayed.
	*YES	Issue all debug messages that would be generated for STRDBG.
NORMALIZE_DATA	*DEFAULT	The default is set to *NO.
	*NO	Unicode constants, host variables, parameter markers, and expressions that combine strings will not be normalized.
	*YES	Unicode constants, host variables, parameter markers, and expressions that combine strings will be normalized
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 28. 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) indicates 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 cancel 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 (See note)	*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 28. 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 28. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
REOPTIMIZE_ACCESS_PLAN	*DEFAULT	The default value is set to *NO.
	*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, and so on. Non-subjective reasons include, deletion of an index used by existing access plan, query file being deleted and recreated, and so on.
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 is + 30, 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.
STAR_JOIN (see note)	*DEFAULT	The default value is set to *NO
	*NO	The EVI Star Join optimization support is not enabled.
	*COST	Allow query optimization to consider (cost) the usage of EVI Star Join support.  The determination of whether 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.

Table 28. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
UDF_TIME_OUT (see note)	*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.
VARIABLE_LENGTH_OPTIMIZATION	*DEFAULT	The default value is set to *YES.
	*YES	Allow aggressive optimization of variable length columns. Allows index only access for the column(s). It also allows constant value substitution when an equal predicate is present against the column(s). As a consequence, the length of the data returned for the variable length column may not include any trailing blanks that existed in the original data.
	*NO	Do not allow aggressive optimization of variable length columns.

**Note:** Only modifies the environment for the Classic Query Engine.

## Control long-running queries with the 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 “Using the Query Governor” for details on how the query governor works in conjunction with query optimizer.

See “Canceling a query with the Query Governor” on page 109 to see how to cancel a query that is predicted to run beyond its time limit.

Before using the predictive query governor, you should see the following topics:

- “Query governor implementation considerations” on page 109
- “Controlling the default reply to the query governor inquiry message” on page 109

You can also test the performance of your queries using the predictive query governor. See “Testing performance with the query governor” on page 109.

And finally, see “Examples of setting query time limits” on page 110 for examples of using the query governor.

## Using the Query Governor

The governor works in conjunction with the query optimizer. When a user issues a request to the server 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.

### 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 might 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.

## Canceling a query with the Query Governor

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 might 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 typically 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.

## 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 about the \*SYSRPLY parameter, see the Change Job (CHGJOB) CL command 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 can find by running the PRSQLINF (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 syntax or both 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 QRYOPLIB 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 about setting the query options file, see “Control queries dynamically with the query options file QAQQINI” on page 97.

To set the query time limit for 45 seconds you can 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 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 is 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

- | There are two types of parallel processing available. The first is a parallel I/O that is available at no charge. The second is DB2 UDB Symmetric Multiprocessing, a feature that you can purchase. You can turn parallel processing 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 “Objects processed in parallel” on page 47 for a description of the performance characteristics and restrictions of each of the parallel access methods.

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
- RTVSYVAL - 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 QQRYDEGREE system value is \*NONE, so you must change the value if you want parallel query processing 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 QAQQINI 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.

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 QQRYDEGREE system value.

The initial value of the DEGREE attribute for a job is \*SYSVAL.



---

## Collecting statistics with the Statistics Manager

As stated earlier, the collection of statistics is handled by a separate component called the Statistics Manager. Statistical information can be used by the query optimizer to determine the best access plan for a query. 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 only rarely is it necessary to update statistics manually.

The Statistics Manager does not actually run or optimize the query. It controls the access to the metadata and other information that is required to optimize the query. It uses this information to answer questions posed by the query optimizer. The answers can either be derived from table header information, from existing indexes, or from single-column statistics.

The Statistics Manager typically gathers and keeps track of the following information:

### **Cardinality of values**

This is the number of unique or distinct occurrences of a specific value in a single column or multiple columns of a table

### **Selectivity**

Also known as a histogram, this information is an indication of how many rows will be selected by any given selection predicate or combination of predicates. Using sampling techniques, it describes the selectivity and distribution of values in a given column of the table.

### **Frequent values**

This is the top *m* most frequent values of a column together with account of how frequently each value occurs. Currently, the top value is 100. This information is obtained by making use of statistical sampling techniques. Built-in algorithms eliminate the possibility of data skewing; for example, NULL values and default values that might influence the statistical values are not taken into account.

### **Metadata information**

This includes the total number of rows in the table, indexes that exist over the table, and which indexes would be useful for implementing the particular query.

### **Estimate of IO operation**

This is an estimate of the amount of IO operations that are required to process the table or the identified index.

While some of these values may have been previously available through an index, statistics have the advantage of being precalculated and stored with table for faster access. Column statistics stored with a table do not dramatically increase the size of the table object—statistics per column average only 8 to 12 KB in size.

The Statistics Manager must always provide an answer to the questions from the Optimizer. It uses the best method available to provide the answers. For example, it may use a single-column statistic or perform a key range estimate over an index. Along with the answer, the Statistics Manager returns a confidence level to the optimizer that the optimizer may use to provide greater latitude for sizing algorithms. If the Statistics Manager provides a low confidence in the number of groups that are estimated for a grouping request, then the optimizer may increase the size of the temporary hash table allocated.

For more information about statistics and the Statistics Manager, see the following topics:

- “Automatic statistics collection” on page 114

- | • “Automatic statistics refresh”
- | • “Viewing statistics requests” on page 115
- | • “Indexes versus column statistics” on page 115
- | • “Monitoring background statistics collection” on page 116
- | • “Replication of column statistics with CRTDUPOBJ versus CPYF” on page 116
- | • “Determining what column statistics exist” on page 116
- | • “Manually collecting and refreshing statistics” on page 117
- | • “Statistics Manager APIs” on page 118

## | **Automatic statistics collection**

| When the Statistics Manager prepares its responses to the Optimizer, it keeps track of the responses that are generated by using default filter factors (because column statistics or indexes were not available). It uses this information during the time that the access plan is being written to the Plan Cache to automatically generate a statistic collection request for the columns. As system resources become available, the requested column statistics will be collected in the background. That way, the next time that the query is executed, the missing column statistics will be available to the Statistics Manager, thus allowing it to provide more accurate information to the Optimizer. More statistics make it easier for the Optimizer to generate a good performing access plan.

| If a query is canceled before or during execution, the requests for column statistics are still processed, as long as the execution reaches the point where the generated access plan is written to the Plan Cache.

| To minimize the number of passes through a table during statistics collection, the Statistics Manager groups multiple requests for the same table together. For example, two queries are executed against table T1. The first query has selection criteria on column C1 and the second over column C2. If no statistics are available for the table, the Statistics Manager identifies both of these columns as good candidates for column statistics. When the Statistics Manager reviews requests, it looks for multiple requests for the same table and groups them together into one request. This allows both column statistics to be created with only one pass through table T1.

| One thing to note is that column statistics normally are automatically created when the Statistics Manager must answer questions from the optimizer using default filter factors. However, when an index is available that might be used to generate the answer, then column statistics are not automatically generated. There may be cases where optimization time would benefit from column statistics in this scenario because using column statistics to answer questions from the optimizer is generally more efficient than using the index data. So if you have cases where the query performance seems extended, you might want to verify that there are indexes over the relevant columns in your query. If this is the case, try manually generating columns statistics for these columns.

| As stated before, statistics collection occurs as system resources become available. If you have schedule a low priority job that is permanently active on your system and that is supposed to use all spare CPU cycles for processing, your statistics collection will never become active.

## | **Automatic statistics refresh**

| Column statistics are not maintained when the underlying table data changes. This means that there has to be some way for the Statistics Manager to determine if columns statistics are still valid or if they no longer represent the column accurately (stale). This validation is done each time one of the following occurs:

- | • A full open occurs for a query where column statistics were used to create the access plan
- | • A new plan is added to the plan cache, either because a completely new query was optimized or because an existing plan was re-optimized.

| To validate the statistics, the Statistics Manager checks to see if any of the following apply:

- | • Number of rows in the table has changed by more than 15% of the total table row count
  - | • Number of rows changed in the table is more than 15% of the total table row count
- | If the statistics is determined to be stale, the Statistics Manager would then still use the stale column statistics to answer the questions from the optimizer, but it also marks the column statistics as stale in the Plan Cache and generate a request to refresh the statistics.

## | **Viewing statistics requests**

| You can view the current statistics requests by using iSeries Navigator or by using Statistics APIs. To view requests in iSeries Navigator, right-click **Database** and select **Statistic Requests**. This window shows all user requested statistics collections that are pending or active, as well as all system requested statistics collections that are active or have failed. You can change the status of the request, order the request to process immediately, or cancel the request.

| You can find information about Statistics APIs at “Statistics Manager APIs” on page 118.

## | **Indexes versus column statistics**

| If you are trying to decide whether to use statistics or indexes to provide information to the Statistics Manager, keep the following differences in mind.

| One major difference between indexes and column statistics is that indexes are permanent objects that are updated when changes to the underlying table occur, while column statistics are not. If your data is constantly changing, the Statistics Manager may need to rely on stale column statistics. However, maintaining an index after each change to the table might take up more system resources than refreshing the stale column statistics after a group of changes to the table have occurred.

| Another difference is the effect that the existence of new indexes or column statistics has on the Optimizer. When new indexes become available, the Optimizer will consider them for implementation. If they are candidates, the Optimizer will re-optimize the query and try to find a better implementation. However, this is not true for column statistics. When new or refreshed column statistics are available, the Statistics Manager will interrogate immediately. Reoptimization will occur only if the answers are significantly different from the ones that were given before these refreshed statistics. This means that it is possible to use statistics that are refreshed without causing a reoptimization of an access plan.

| When trying to determine the selectivity of predicates, the Statistics Manager considers column statistics and indexes as resources for its answers in the following order:

- | 1. Try to use a multi-column keyed index when ANDed or ORed predicates reference multiple columns
- | 2. If there is no perfect index that contains all of the columns in the predicates, it will try to find a combination of indexes that can be used.
- | 3. For single column questions, it will use available column statistics
- | 4. If the answer derived from the column statistics shows a selectivity of less than 2%, indexes are used to verify this answer

| Accessing column statistics to answer questions is faster than trying to obtain these answers from indexes.

| Column statistics can only be used by SQE. For CQE, all statistics are retrieved from indexes.

| Finally, column statistics can be used only for query optimization. They cannot be used for the actual implementation of a query, whereas indexes can be used for both.

## Monitoring background statistics collection

The system value QDBFSTCCOL controls who is allowed to create statistics in the background. The following list provides the possible values:

### \*ALL

Allows all statistics to be collected in the background. This is the default setting.

### \*NONE

Restricts everyone from creating statistics in the background. This does not prevent immediate user-requested statistics from being collected, however.

### \*USER

Allows only user-requested statistics to be collected in the background.

### \*SYSTEM

Allows only system-requested statistics to be collected in the background.

When you switch the system value to something other than \*ALL or \*SYSTEM, the Statistics Manager continues to place statistics requests in the Plan Cache. When the system value is switched back to \*ALL, for example, background processing analyzes the entire Plan Cache and looks for any column statistics requests that are there. This background task also identifies column statistics that have been used by a plan in the Plan Cache and determines if these column statistics have become stale. Requests for the new column statistics as well as requests for refresh of the stale columns statistics are then executed.

All background statistic collections initiated by the system or submitted to the background by a user are performed by the system job QDBFSTCCOL (user-initiated immediate requests are run within the user's job). This job uses multiple threads to create the statistics. The number of threads is determined by the number of processors that the system has. Each thread is then associated with a request queue.

There are four types of request queues based on who submitted the request and how long the collection is estimated to take. The default priority assigned to each thread can determine to which queue the thread belongs:

- Priority 90 — short user requests
- Priority 93 — long user requests
- Priority 96 — short system requests
- Priority 99 — long system requests

Background statistics collections attempt to use as much parallelism as possible. This parallelism is independent of the SMP feature installed on the iSeries. However, parallel processing is allowed only for immediate statistics collection if SMP is installed on the system and the job requesting the column statistics is set to allow parallelism.

## Replication of column statistics with CRTDUPOBJ versus CPYF

Statistics are not copied to new tables when using the CPYF command. If statistics are needed immediately after using this command, then you must manually generate the statistics using iSeries Navigator or the statistics APIs. If statistics are not needed immediately, then the creation of column statistics may be performed automatically by the system after the first touch of a column by a query.

Statistics are copied when using CRTDUPOBJ command with DATA(\*YES). You can use this as an alternative to creating statistics automatically after using a CPYF command.

## Determining what column statistics exist

You can determine what column statistics exist in a couple of ways. The first is to view statistics by using iSeries Navigator. Right-click a table or alias and select **Statistic Data**. Another way is to create a user-defined table function and call that function from an SQL statement or stored procedure.

## Manually collecting and refreshing statistics

You can manually collect and refresh statistics through iSeries Navigator or by using Statistics APIs. To collect statistics using iSeries Navigator, right-click a table or alias and select **Statistic Data**. On the **Statistic Data** dialog, click **New**. Then select the columns that you want to collect statistics for. Once you have selected the columns, you can collect the statistics immediately or collect them in the background.

To refresh a statistic using iSeries Navigator, right-click a table or alias and select **Statistic Data**. Click **Update**. Select the statistic that you want to refresh. You can collect the statistics immediately or collect them in the background.

For information about using Statistic APIs to collect and refresh statistics, see “Statistics Manager APIs” on page 118.

There are several scenarios in which the manual management (create, remove, refresh, and so on) of column statistics may be beneficial and recommended.

### High Availability (HA) solutions

When considering the design of high availability solutions where data is replicated to a secondary system by using journal entries, it is important to know that column statistics information is not journaled. That means that, on your backup system, no column statistics are available when you first start using that system. To prevent the “warm up” effect that this may cause, you may want to propagate the column statistics were gathered on your production system and re-create them on your backup system manually.

### ISV (Independent Solution Provider) preparation

An ISV may want to deliver a solution to a customer that already includes column statistics frequently used in the application instead of waiting for the automatic statistics collection to create them. A way to accomplish this is to run the application on the development system for some time and examine which column statistics were created automatically. You can then generate a script file to be shipped as part of the application that should be executed on the customer system after the initial data load took place.

### Business Intelligence environments

In a large Business Intelligence environment, it is quite common for large data load and update operations to occur overnight. As column statistics are marked as stale only when they are touched by the Statistics Manager, and then refreshed after first touch, you may want to consider refreshing them manually after loading the data.

You can do this easily by toggling the system value QDBFSTCCOL to \*NONE and then back to \*ALL. This causes all stale column statistics to be refreshed and starts collection of any column statistics previously requested by the system but not yet available. Since this process relies on the access plans stored in the Plan Cache, avoid performing a system initial program load (IPL) before toggling QDBFSTCCOL since an IPL clears the Plan Cache.

You should be aware that this procedure works only if you do not delete (drop) the tables and re-create them in the process of loading your data. When deleting a table, access plans in the Plan Cache that refer to this table are deleted. Information about column statistics on that table is also lost. The process in this environment is either to add data to your tables or to clear the tables instead of deleting them.

### Massive data updates

Updating rows in a column statistics-enabled table that significantly change the cardinality, add new ranges of values, or change the distribution of data values can affect the performance for queries

when they are first run against the new data. This may happen because, on the first run of such a query, the optimizer uses stale column statistics to make decisions on the access plan. At that point, it starts a request to refresh the column statistics.

If you know that you are doing this kind of update to your data, you may want to toggle the system value QDBFSTCCOL to \*NONE and back to \*ALL or \*SYSTEM. This causes an analysis of the Plan Cache. The analysis includes searching for column statistics that were used in the generation of an access plan, analyzing them for staleness, and requesting updates for the stale statistics.

If you massively update or load data and run queries against these tables at the same time, then the automatic collection of column statistics tries to refresh every time 15% of the data is changed. This can be redundant processing since you are still in the process of updating or loading the data. In this case, you may want to block automatic statistics collection for the tables in question and unblock it again after the data update or load finishes. An alternative is to turn off automatic statistics collection for the whole system before updating or loading the data and switching it back on after the updating or loading has finished.

### **Backup and recovery**

When thinking about backup and recovery strategies, keep in mind that creation of column statistics is not journaled. Column statistics that exist at the time a save operation occurs are saved as part of the table and restored with the table. Any column statistics created after the save took place are lost and cannot be re-created by using techniques such as applying journal entries. If you have a rather long interval between save operations and rely heavily on journaling for restoring your environment to a current state, consider keeping track of column statistics that are generated after the latest save operation.

## **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, QdbstListDetailStatistics) 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



## Query optimization tools: Comparison table

<b>PRTSQLINF</b>	<b>STRDBG or CHGQRYA</b>	<b>File-based monitor</b>	<b>Memory -Based Monitor</b>	<b>Visual Explain</b>
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 about host variable implementation	Limited information about the implementation of host variables	All information about host variables, implementation, and values	All information about host variables, implementation, and values	All information about 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
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 8. Creating an index strategy

DB2 Universal Database for iSeries provides two basic means for accessing tables: a table scan and an index-based retrieval. Index-based retrieval is typically more efficient than table scan when less than 20% of the table rows are selected. See the following sections for more information about using index:

- “Index basics”
- “Indexes and the optimizer” on page 125
- “Indexing strategy” on page 127
- “Coding for effective indexes” on page 129
- “Using indexes with sort sequence” on page 131
- “Examples of indexes” on page 132

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

---

### Index basics

There are two kinds of persistent indexes: binary radix tree indexes, which have been available since 1988, and encoded vector indexes (EVIs), which became available in 1998 with V4R2. Both types of indexes are useful in improving performance for certain kinds of queries.

- “Binary radix indexes”
- “Encoded vector indexes” on page 122
- “Comparing Binary radix indexes and Encoded vector indexes” on page 125

### Binary radix indexes

A radix index is a multilevel, hybrid tree structure that allows a large number of key values to be stored efficiently while minimizing access times. A key compression algorithm assists in this process. The lowest level of the tree contains the leaf nodes, which contain the address of the rows in the base table that are associated with the key value. The key value is used to quickly navigator to the leaf node with a few simple binary search tests.

The binary radix tree structure is very good for finding a small number of rows because it is able to find a given row with a minimal amount of processing. For example, using a binary radix index over a customer number column for a typical OLTP request like “find the outstanding orders for a single customer: will result in fast performance. An index created over the customer number column is considered to be the perfect index for this type of query because it allows the database to zero in on the rows it needs and perform a minimal number of I/Os.

In some situations, however, you do not always have the same level of predictability. Increasingly, users want ad hoc access to the detail data. They might for example, run a report every week to look at sales data, then “drill down” for more information related to a particular problem areas that they found in the report. In this scenario, you cannot write all of the queries in advance on behalf of the end users. Without knowing what queries will be run, it is impossible to build the perfect index.

### 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) 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.

## 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.

### Creating EVIs

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

### EVI maintenance

There are unique challenges to maintaining EVIs. 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 characteristics.

Table 29. EVI Maintenance Considerations

Most Effective	Condition	Characteristics
Least Effective	When inserting an existing distinct key value	<ul style="list-style-type: none"> <li>• Minimum overhead</li> <li>• Symbol table key value looked up and statistics updated</li> <li>• Vector element added for new row, with existing byte code</li> </ul>

Table 29. EVI Maintenance Considerations (continued)

	When inserting a <i>new</i> distinct key value - in order, within byte code range	<ul style="list-style-type: none"> <li>• Minimum overhead</li> <li>• Symbol table key value added, byte code assigned, statistics assigned</li> <li>• Vector element added for new row, with new byte code</li> </ul>
	When inserting a new distinct key value - out of order, within byte code range	<ul style="list-style-type: none"> <li>• Minimum overhead if contained within overflow area threshold</li> <li>• Symbol table key value added to overflow area, byte code assigned, statistics assigned</li> <li>• Vector element added for new row, with new byte code</li> <li>• Considerable overhead if overflow area threshold reached</li> <li>• Access path validated - not available</li> <li>• EVI refreshed, overflow area keys incorporated, new byte codes assigned (symbol table and vector elements updated)</li> </ul>
	When inserting a new distinct key value - out of byte code range	<ul style="list-style-type: none"> <li>• Considerable overhead</li> <li>• Access plan invalidated - not available</li> <li>• EVI refreshed, next byte code size used, new byte codes assigned (symbol table and vector elements updated)</li> </ul>

### 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 File Description (DSPFD) command (or iSeries 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.

**Comparing Binary radix indexes and Encoded vector indexes**

DB2 UDB for iSeries makes indexes a powerful tool. The following table summarizes some of the concepts discussed in this section:

	Binary Radix Indexes	Encoded Vector Indexes
Basic data structure	A wide, flat tree	A Symbol Table and a vector
Interface for creating	Command, SQL, iSeries Navigator	SQL, iSeries Navigator
Can be created in parallel	Yes	Yes
Can be maintained in parallel	Yes	Yes
Used for statistics	Yes	Yes
Used for selection	Yes	Yes, via dynamic bitmaps or RRN list
Used for joining	Yes	No
Used for grouping	Yes	No
Used for ordering	Yes	No
Used to enforce unique Referential Integrity constraints	Yes	No

**Indexes and the optimizer**

Since the iSeries optimizer uses cost based optimization, the more information that the optimizer is given about the rows and columns in the database, the better able the optimizer is to create the best possible (least costly/fastest) access plan for the query. With the information from the indexes, the optimizer can make better choices about how to process the request (local selection, joins, grouping, and ordering).

| The CQE optimizer attempts to examine most, if not all, indexes built over at able unless or until it times out. However, the SQE optimizer only considers those indexes that are returned by the Statistics Manager. These include only indexes that the Statistics Manager decides are useful in performing local selection based on the "where" clause predicates. Consequently, the SQE optimizer does not time out.

| The primary goal of the optimizer is to choose an implementation that quickly and efficiently eliminates the rows that are not interesting or required to satisfy the request. Normally, query optimization is thought of as trying to find the rows of interest. A proper indexing strategy will assist the optimizer and database engine with this task.

- | • "Instances where an index is not used"
- | • "Determining unnecessary indexes" on page 127

## | Instances where an index is not used

| 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, the system cannot use an index with a key of WORKDEPT.

| The system 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, the system uses an index with a key of EMPNO.

| The system 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 OPNQRYF command, for example:

```
| OPNQRYF 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.

## Determining unnecessary indexes

Prior to V5R3, it was difficult to determine unnecessary indexes. Using the Last Used Date was not dependable, as it was only updated when the logical file was opened using a native database application (for example, in an RPG application). Furthermore, it was difficult to find all the indexes over a physical file. Indexes are created as part of a keyed physical file, a keyed logical file, a join logical file, an SQL index, a primary key or unique constraint, or a referential constraint. However, you can now easily find all indexes and retrieve statistics on index usage as a result of new V5R3 iSeries Navigator and OS/400 functionality. To assist you in tuning your performance, this function now produces statistics on index usage as well as index usage in a query. To take advantage of this new feature, you must have the program temporary fixes (PTFs) SF99503 version 4 applied.

You can access this new feature through the iSeries Navigator. This feature requires that your iSeries Access for Windows is at V5R3 with iSeries Access PTF number SI15176 installed. To access this through the iSeries Navigator, navigate to:

1. **Database**
2. **Schemas**
3. **Tables**
4. Right-click your table and select **Indexes**

**Note:** You can also view the statistics through an application programming interface (API). See Retrieve Member Description (QUSRMBRD) for more information.

In addition to all existing attributes of an index, four new fields have been added to the iSeries Navigator. Those four new fields are:

### Last Query Use

States the timestamp when the index was last used to access tables in a query.

### Last Query Statistic Use

States the timestamp when the index was last used to gather statistical information.

### Query Use Count

Lists the number of instances the index was used in a query.

### Query Statistics Use

Lists the number of instances the index was used for statistical information.

**Note:** If a query statistics use is large, it does not mean it cannot be deleted. This should be used as an indication that it is less likely that you want to delete the index.

The fields start and stop counting based on your situation, or the actions you are currently performing on your system. The following list describes what might affect one or both of your counters:

- The SQE and CQE query engines increment both counters. As a result, the statistics field will be updated regardless of what query interface is used.
- A save and restore procedure does not reset the statistics counter if the index is restored over an existing index. If an index is restored that does not exist on the server, the statistics are reset.
- The statistics counter begins after the PTFs are applied.

---

## Indexing strategy

There are two approaches to index creation: proactive and reactive. As the name implies proactive index creation involves anticipating which columns will be most often used for selection, joining, grouping and ordering; and then building indexes over those columns. In the reactive approach, indexes are created based on optimizer feedback, query implementation plan, and system performance measurements.



| It is useful to initially build indexes based on the database model and application(s) and not any particular query. As a starting point, consider designing basic indexes based on the following criteria:

- | • Primary and foreign key columns based on the database model
- | • Commonly used local selection columns, including columns that are dependent, such as an automobile's make and model
- | • Commonly used join columns not considered primary or foreign key columns
- | • Commonly used grouping columns

| For more information about an index strategy, see the following sections:

- | • "Reactive approach to tuning"
- | • "Proactive approach to tuning"

| For additional information about developing an index strategy, see Indexing Strategies for DB2 UDB for iSeries .

## | **Reactive approach to tuning**

| To perform reactive tuning, build a prototype of the proposed application without any indexes and start running some queries or build an initial set of indexes and start running the application to see what gets used and what does not. Even with a smaller database, the slow running queries will become obvious very quickly. The reactive tuning method is also used when trying to understand and tune an existing application that is not performing up to expectations. Using the appropriate debugging and monitoring tools, which are described in the next section, the database feedback messages that will tell basically three things can be viewed:

- | • Any indexes the optimizer recommends for local selection
- | • Any temporary indexes used for a query
- | • The implementation method(s) that the optimizer has chosen to run the queries

| If the database engine is building temporary indexes to process joins or to perform grouping and selection over permanent tables, permanent indexes should be built over the same columns to try to eliminate the temporary index creation. In some cases, a temporary index is built over a temporary table, so a permanent index will not be able to be built for those tables. You can use the optimization tools listed in the previous section to note the creation of the temporary index, the reason the temporary index was created, and the key columns in the temporary index.

## | **Proactive approach to tuning**

| Typically you will create an index for the most selective columns and create statistics for the least selective columns in a query. By creating an index, the optimizer knows that the column is selective and it also gives the optimizer the ability to chose that index to implement the query.

| In a perfect radix index, the order of the columns is important. In fact, it can make a difference as to whether the optimizer uses it for data retrieval at all. As a general rule, order the columns in an index in the following way:

- | • Equal predicates first. That is, any predicate that uses the "=" operator may narrow down the range of rows the fastest and should therefore be first in the index.
- | • If all predicates have an equal operator, then order the columns as follows:
  - | – Selection predicates + join predicates
  - | – Join predicates + selection predicates
  - | – Selection predicates + group by columns
  - | – Selection predicates + order by columns

| In addition to the guidelines above, in general, the most selective key columns should be placed first in the index.



| Consider the following SQL statement:

```
| SELECT b.col1, b.col2, a.col1
| FROM table1 a, table2 b
| WHERE b.col1='some_value',
| b.col2=some_number,
| a.join_col=b.join_col
| GROUP BY b.col1, b.col2, a.col1
| ORDER BY b.col1
```

| With a query like this, the proactive index creation process can begin. The basic rules are:

| • Custom-build a radix index for the largest or most commonly used queries. Example using the query above:

```
| radix index over join column(s) - a.join_col and b.join_col
| radix index over most commonly used local selection column(s) - b.col2
```

| • For ad hoc online analytical processing (OLAP) environments or less frequently used queries, build single-key EVIs over the local selection column(s) used in the queries. Example using the query above:

```
| EVI over non-unique local selection columns - b.col1 and b.col2
```

---

## Coding for effective indexes

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:

- “Avoid numeric conversions”
- “Avoid arithmetic expressions” on page 130
- “Avoid character string padding” on page 130
- “Avoid the use of like patterns beginning with % or \_” on page 130
- “Instances where an index is not used” on page 126

### 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 need 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.

## 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
```

## 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"')
```

## 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 can be used, but index scan-key positioning cannot.

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.

---

## Using indexes with sort sequence

The following sections provide useful information about how indexes work with sort sequence tables.

- "Using indexes and sort sequence with selection, joins, or grouping"
- "Using indexes and sort sequence with ordering"

For more information about how sort sequence tables work, see the topic "Sort Sequence" in the *SQL Reference* topic.

## 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 (OPNQRYF %WLDCRD and \*CT)
- IN predicate (OPNQRYF %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

### Notes:

1. The table does not need to match the unique-weight sort sequence table associated with the query.
2. 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.

## Using indexes and sort sequence with 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

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

### 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```

The system can 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)
```



The system can 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)
```

The system can 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)
```

The system can use UNQIX2 to satisfy both the grouping and ordering requirements. If index UNQIX2 did not exist, the system 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:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(JOB SALARY)
  SRTSEQ(*LANGIDUNQ) LANGID(ENU)
  ALWCPYDTA(*OPTIMIZE)
```

The system can use UNQIX2 to satisfy both the grouping and ordering requirements. If index UNQIX2 did not exist, the system 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 OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(JOB SALARY)
  SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

The system can use SHRIX2 to satisfy both the grouping and ordering requirements. If index SHRIX2 did not exist, the system 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 OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF)) FORMAT(FORMAT3)
  GRPFLD(JOB SALARY)
  KEYFLD(JOB SALARY)
  SRTSEQ(*LANGIDSHR) LANGID(ENU)
  ALWCPYDTA(*OPTIMIZE)
```

The system can use SHRIX2 to satisfy both the grouping and ordering requirements. If index SHRIX2 did not exist, the system 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)
```

The system can use index HEXIX2 or index UNQIX2 to satisfy the grouping requirements. A temporary result is created containing the grouping results. A temporary index is then 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)
```

The system can use index HEXIX2 or index UNQIX2 to satisfy the grouping requirements. A sort is 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)
```

The system can use index SHRIX2 to satisfy the grouping requirements. A sort is performed to satisfy the ordering requirements.

---

## Chapter 9. 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:

- “Use live data”
- “Reduce the number of open operations” on page 140
- “Retain cursor positions” on page 142

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

---

### 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 (ALWCPYDTA) parameter on the precompiler commands or on the Start SQL (STRSQL) command.

Specifying ALWCPYDTA(\*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 need 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 can result in creating an additional ODP that increases the processing time on the iSeries server. In place of the above example, the programmer can design the application specifying ALWCPYDTA(\*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 might perform better if the FOR 20 ROWS clause was used on the multiple-row FETCH statement. Then, the 20 rows are retrieved in one operation.

---

## 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 affects 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 changes 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 that was not reusable 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.
- Open data paths left open by DB2 Universal Database when the application has requested a close can be forced to close for a specific file by using the ALCOBJ CL command. This will not force the ODP to be closed if the application has not requested the cursor be closed. The syntax for the command is: ALCOBJ OBJ((library/file \*FILE \*EXCL)) CONFLICT(\*RQSRLS).

You can control whether the system 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)

The system 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 the system 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 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).

---

## 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.

### 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.

## **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.

## **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 can 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 might also control this by sending a special parameter on the first call only. This special parameter value indicates 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 you wanted.

---

## Chapter 10. Programming techniques for database performance

The following coding tips can help you improve the performance of your SQL queries:

- “Use the OPTIMIZE clause”
- “Use FETCH FOR n ROWS” on page 146
- “Use INSERT n ROWS” on page 147
- “Control database manager blocking” on page 147
- “Optimize the number of columns that are selected with SELECT statements” on page 148
- “Eliminate redundant validation with SQL PREPARE statements” on page 149
- “Page interactively displayed data with REFRESH(\*FORWARD)” on page 149

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

---

### 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.

## 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 about declaring arrays of host structures or row storage areas, see the SQL Reference book or the individual programming chapters in the Embedded SQL Programming 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 "Control database manager blocking" on page 147.

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 30. 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

## 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 is 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.

---

## 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 31. 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

---

## 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 you want. 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 need 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.

---

## 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 wanted 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.

---

## 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.

---

## Page interactively displayed data with REFRESH(\*FORWARD)

In large tables, paging performance is typically 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 11. 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 152
- “Tips for using VARCHAR and VARGRAPHIC data types in databases” on page 153

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

---

### 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 CHGCMDDFT command to customize the optimal values for the precompile parameters. The DSPPGM, DSPSRVPGM, DSPMOD, or PRTSQLINF 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 152.
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 “Control database manager blocking” on page 147.

Precompile Option	Optimal Value	Improvements	Considerations	Related Topics
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 "Retaining cursor positions for non-ILE program calls" on page 142.
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 "Eliminate redundant validation with SQL PREPARE statements" on page 149.
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 can 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 before 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 "Use the OPTIMIZE clause" on page 145 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 telephone 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 can cause more rows to spill into the overflow space. This increases 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 allocate 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 Embedded SQL Programming information.

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. For more information about Reorganizing a table or file, see Reorganizing a physical file topic in *Database programming*.

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.

---

## Chapter 12. Database Monitor DDS

See the following for reference information about database monitor DDS:

- “Database monitor: DDS”
- “Memory Resident Database Monitor: DDS” on page 252

**Note:** Read the “Code disclaimer” on page 2 for important legal information.

---

### Database monitor: DDS

This section 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 162

### 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          QQSYS              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 + COLHDG('NLSS' 'Library')
A	QQSTIM	Z	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') + ALWNULL + COLHDG('Data' 'Space' + Selection')
A	QQIDXA	1A	TEXT('Index advised') + ALWNULL + COLHDG('Index' 'Advised')
A	QQORDG	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 +
A	QQRCDS	1A	COLHDG('Host' 'Variables') TEXT('Row Selection') + ALWNULL +
A	QQRCOD	2A	COLHDG('Row' 'Selection') TEXT('Reason Code') + ALWNULL +
A	QQRSS	15P	COLHDG('Reason' 'Code') TEXT('Number of rows + selected or sorted') + ALWNULL +
A	QQREST	15P	COLHDG('Number of' + 'Rows' + 'Selected') TEXT('Estimated number + of rows selected') + ALWNULL +
A	QQRIDX	15P	COLHDG('Estimated' + 'Rows' + 'Selected') TEXT('Number of entries + in index created') + ALWNULL +
A	QQFKEY	15P	COLHDG('Entries in' + 'Index' + 'Created') TEXT('Estimated keys for + index scan-key positioning') + ALWNULL +
A	QQKSEL	15P	COLHDG('Estimated' + 'Entries for' + 'index scan-key positioning') TEXT('Estimated keys for + key selection') + ALWNULL +
A	QQAJN	15P	COLHDG('Estimated' + 'Entries for' + 'Key Selection') TEXT('Estimated number + of joined rows') + ALWNULL +
A	QQIDX	1000A	COLHDG('Estimated' + 'Joined' + 'Rows') VARLEN(48) + TEXT('Columns + for the index advised') + ALWNULL +
A	QQC11	1A	COLHDG('Advised' 'Key' + 'Columns')
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' +

A	QVILIB	128A	'Used' + 'Long Name') 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') TEXT('Decomposed' + 'Subselect + Reason code') + ALWNULL + COLHDG('Decomposed' + 'Reason' + 'Code')
A	QVP15C	15P	
			TEXT('Number of first' + 'Decomposed + Subselect') + ALWNULL + COLHDG('Starting' + 'Decomposed' + 'Subselect')
A	QVP15D	15P	
			TEXT('Materialized Union' + 'Level') ALWNULL + COLHDG('Materialized' + 'Union' + 'Level')
A	QVP15E	15P	
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	QVC1000	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 163

- “Database monitor logical table 3000 - Summary Row for Table Scan” on page 179
- “Database monitor logical table 3001 - Summary Row for Index Used” on page 183
- “Database monitor logical table 3002 - Summary Row for Index Created” on page 190
- “Database monitor logical table 3003 - Summary Row for Query Sort” on page 197
- “Database monitor logical table 3004 - Summary Row for Temp Table” on page 201
- “Database monitor logical table 3005 - Summary Row for Table Locked” on page 206
- “Database monitor logical table 3006 - Summary Row for Access Plan Rebuilt” on page 209
- “Database monitor logical table 3007 - Summary Row for Optimizer Timed Out” on page 212
- “Database monitor logical table 3008 - Summary Row for Subquery Processing” on page 215
- “Database monitor logical table 3010 - Summary for HostVar & ODP Implementation” on page 217
- “Database monitor logical table 3014 - Summary Row for Generic QQ Information” on page 218
- “Database monitor logical table 3015 - Summary Row for Statistics Information” on page 226
- “Database monitor logical table 3018 - Summary Row for STRDBMON/ENDDDBMON” on page 228
- “Database monitor logical table 3019 - Detail Row for Rows Retrieved” on page 229
- “Database monitor logical table 3021 - Summary Row for Bitmap Created” on page 231
- “Database monitor logical table 3022 - Summary Row for Bitmap Merge” on page 234
- “Database monitor logical table 3023 - Summary for Temp Hash Table Created” on page 237
- “Database monitor logical table 3025 - Summary Row for Distinct Processing” on page 240
- “Database monitor logical table 3027 - Summary Row for Subquery Merge” on page 242
- “Database monitor logical table 3028 - Summary Row for Grouping” on page 246
- “Database monitor logical table 3030 - Summary Row for Materialized query tables” on page 250

## 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          QQCNT
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          QQSNAM          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('CLOSQLCSR' +
                                   '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	QVNAMEC	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      QVSCHM      RENAME(QVC1284) +
                        COLHDG('SQL' +
                        'Schema')
A      QBINDTYPE    RENAME(QQC18) +
                        COLHDG('Binding' +
                        'Type')
A      QCSRTPY     RENAME(QQC61) +
                        COLHDG('Cursor' +
                        'Type')
A      QSQLSTMTO   RENAME(QVC1D) +
                        COLHDG('SQL' +
                        'Statement' +
                        'Originator')
A      QCLSRC      RENAME(QQC15) +
                        COLHDG('SQL Cursor' +
                        'Hard Close' +
                        'Reason Code')
A      QCLSSUBRC   RENAME(QQC23) +
                        COLHDG('SQL Cursor' +
                        'Hard Close' +
                        'Reason Subcode')

```

```

A*
A* Environmental information about the SQL statement executed
A*

```

```

A      QVDFMT      RENAME(QVC42) +
                        COLHDG('Date' +
                        'Format')
A      QVDSEP      RENAME(QWC11) +
                        COLHDG('Date' +
                        'Separator')
A      QVTFMT      RENAME(QVC43) +
                        COLHDG('Time' +
                        'Format')
A      QVTSEP      RENAME(QWC12) +
                        COLHDG('Time' +
                        'Separator')
A      QVDPNT      RENAME(QWC13) +
                        COLHDG('Decimal' +
                        'Point')
A      QVSRYSQ     RENAME(QVC104) +
                        COLHDG('Sort' +
                        'Sequence' +
                        'Table')
A      QVSRYSL     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' +

```

A	QVNOSV	'Reason Code') 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' + 'Constants')
A	QVSQLSUPP	RENAME(QWC16) + COLHDG('SQL' + 'Suppress' + 'Warnings')
A	QVSQLASCII	RENAME(QWC17) + COLHDG('SQL' + 'Translate' + 'ASCII')
A	QVSQLCACHE	RENAME(QWC18) + COLHDG('SQL' + 'Statement' + 'Cache')
A	QLOBTHRHD	RENAME(QVP159) + COLHDG('LOB' + 'Locator' + 'Threshold')
A	QMAXPREC	RENAME(QVP156) + COLHDG('Maximum' + 'Decimal' + 'Precision')
A	QMAXSCLE	RENAME(QVP157) + COLHDG('Maximum' + 'Decimal' + 'Scale')
A	QMINDIV	RENAME(QVP158 + COLHDG('Maximum' + 'Decimal' + 'Divide Scale')
A	NORM_DATA	RENAME(QWC19 + COLHDG('Unicode' + 'Data' + 'Normalization')
A	QSTMTCMP	RENAME(QQF2 + COLHDG('Statement' + 'Compressions' + 'Allowed')
A	QAPLENO	RENAME(QVP15B + COLHDG('Access Plan' + 'Length' +

```

          'Before Rebuild')
A          QAPLENN          RENAME(QVP15C +
          COLHDG('Access Plan' +
          'Length' +
          'After Rebuild')
A          QFSTDCNT          RENAME(QVP151 +
          COLHDG('Minimum' +
          'Row Count' +
          'Fast Delete')
A          QREALUSR          RENAME(QVC102 +
          COLHDG('Real' +
          'User' +
          'Name')
A          QNTSSPID          RENAME(QQC301 +
          COLHDG('NTSLock' +
          'Space' +
          'ID')
A          K QQJFLD
A          S QQRID          CMP(EQ 1000)

```

| Table 32. 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
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
QQSTN	QQSTN	Statement number (unique per statement)
QQSTF	QQC11	Statement function: <ul style="list-style-type: none"> <li>• S - Select</li> <li>• U - Update</li> <li>• I - Insert</li> <li>• D - Delete</li> <li>• L - Data definition language</li> <li>• O - Other</li> </ul>

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QQSTOP	QQC21	Statement operation: <ul style="list-style-type: none"> <li>• AL - Alter table</li> <li>• AQ - Alter sequence</li> <li>• CA - Call</li> <li>• CC - Create collection</li> <li>• CD - Create type</li> <li>• CF - Create function</li> <li>• CG - Create trigger</li> <li>• CI - Create index</li> <li>• CL - Close</li> <li>• CM - Commit</li> <li>• CN - Connect</li> <li>• CO - Comment on</li> <li>• CP - Create procedure</li> <li>• CQ - Create sequence</li> <li>• CS - Create alias/synonym</li> <li>• CT - Create table</li> <li>• CV - Create view</li> <li>• DE - Describe</li> <li>• DI - Disconnect</li> <li>• DL - Delete</li> <li>• DM - Describe parameter marker</li> <li>• DP - Declare procedure</li> <li>• DR - Drop</li> <li>• DT - Describe table</li> <li>• EI - Execute immediate</li> <li>• EX - Execute</li> <li>• FE - Fetch</li> <li>• FL - Free locator</li> <li>• GR - Grant</li> <li>• HC - Hard close</li> <li>• HL - Hold locator</li> <li>• IN - Insert</li> <li>• JR - Prestart job reused</li> <li>• LK - Lock</li> <li>• LO - Label on</li> <li>• MT - More text</li> <li>• OP - Open</li> <li>• PD - Prepare and describe</li> <li>• PR - Prepare</li> <li>• RB - Rollback Savepoint</li> <li>• RE - Release</li> </ul>

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QOSTOP (continued)	QQC21	<ul style="list-style-type: none"> <li>• RF - Refresh Table</li> <li>• RO - Rollback</li> <li>• RS - Release Savepoint</li> <li>• RT - Rename table</li> <li>• RV - Revoke</li> <li>• SA - Savepoint</li> <li>• SC - Set connection</li> <li>• SE - Set encryption password</li> <li>• SI - Select into</li> <li>• SP - Set path</li> <li>• SR - Set result set</li> <li>• SS - Set current schema</li> <li>• ST - Set transaction</li> <li>• SV - Set variable</li> <li>• UP - Update</li> <li>• VI - Values into</li> <li>• X0 - Unknown statement</li> <li>• X1 - Unknown statement</li> <li>• X2 - DRDA Application Server (AS) - unknown statement</li> <li>• X3 - Unknown statement</li> <li>• X9 - Error occurred while processing statement for Database Monitor</li> </ul>
QOSTTY	QQC12	Statement type: <ul style="list-style-type: none"> <li>• D - Dynamic statement</li> <li>• S - Static statement</li> </ul>
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
QOSTIM	QOSTIM	Time this statement entered
QOSTTX	QQ1000	Statement text
QOSTOC	QQC14	Statement outcome <ul style="list-style-type: none"> <li>• S - Successful</li> <li>• U - Unsuccessful</li> </ul>
QQROWR	QQI2	Number of result rows returned

Table 32. 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"> <li>• NA - No replan.</li> <li>• NR - SQL QDT rebuilt for new release.</li> <li>• 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"> <li>– Object was deleted and recreated.</li> <li>– Object was saved and restored.</li> <li>– Library list was changed.</li> <li>– Object was renamed.</li> <li>– Object was moved.</li> <li>– Object was overridden to a different object.</li> <li>– This is the first run of this query after the object containing the query has been restored.</li> </ul> </li> <li>• 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.</li> <li>• 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.</li> <li>• A4 - The number of rows in the table member has changed by more than 10% since the access plan was last built.</li> <li>• A5 - A new index exists over one of the tables in the query.</li> <li>• A6 - An index that was used for this access plan no longer exists or is no longer valid.</li> <li>• A7 - OS/400 Query requires the access plan to be rebuilt because of system programming changes.</li> <li>• A8 - The CCSID of the current job is different than the CCSID of the job that last created the access plan.</li> <li>• 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"> <li>– date format</li> <li>– date separator</li> <li>– time format</li> <li>– time separator</li> </ul> </li> <li>• AA - The sort sequence table specified is different than the sort sequence table that was used when this access plan was created.</li> <li>• AB - Storage pool changed or DEGREE parameter of CHGQRYA command changed.</li> <li>• AC - The system feature DB2 multisystem has been installed or removed.</li> <li>• AD - The value of the degree query attribute has changed.</li> <li>• AE - A view is either being opened by a high level language or a view is being materialized.</li> <li>• 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.</li> </ul>



Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QQDYNR (continued)	QQC22	<ul style="list-style-type: none"> <li>• B0 - The options specified have changed as a result of the query options file.</li> <li>• B1 - The access plan was generated with a commitment control level that is different in the current job.</li> <li>• B2 - The access plan was generated with a static cursor answer set size that is different than the previous access plan.</li> <li>• B3 - The query was reoptimized because this is the first run of the query after a prepare. That is, it is the first run with real actual parameter marker values.</li> <li>• B4 - The query was reoptimized because referential or check constraints have changed.</li> <li>• B5 - The query was reoptimized because the MQT was no longer eligible to be used. Possible reasons are: <ul style="list-style-type: none"> <li>– The MQT no longer exists</li> <li>– A new MQT was found,</li> <li>– The enable/disable query optimization changed</li> <li>– Time since the last REFRESH TABLE exceeds the MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option</li> <li>– Other QAQQINI options no longer match.</li> </ul> </li> </ul>
QQDACV	QQC16	<p>Data conversion</p> <ul style="list-style-type: none"> <li>• N - No.</li> <li>• 0 - Not applicable.</li> <li>• 1 - Lengths do not match.</li> <li>• 2 - Numeric types do not match.</li> <li>• 3 - C host variable is NUL-terminated.</li> <li>• 4 - Host variable or column is variable length and the other is not variable length.</li> <li>• 5 - CCSID conversion.</li> <li>• 6 - DRDA<sup>®</sup> and NULL capable, variable length, contained in a partial row, derived expression, or blocked fetch with not enough host variables.</li> <li>• 7 - Data, time, or timestamp column.</li> <li>• 8 - Too many host variables.</li> <li>• 9 - Target table of an insert is not an SQL table.</li> </ul>
QQTTIM	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
QQTTIMM	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
QQUCNT	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

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVSQST	QQC81	SQLSTATE
QVCLSCR	QVC101	Close Cursor. Possible values are: <ul style="list-style-type: none"> <li>• *ENDJOB - SQL cursors are closed when the job ends.</li> <li>• *ENDMOD - SQL cursors are closed when the module ends</li> <li>• *ENDPGM - SQL cursors are closed when the program ends.</li> <li>• *ENDSQL - SQL cursors are closed when the first SQL program on the call stack ends.</li> <li>• *ENDACTGRP - SQL cursors are closed when the activation group ends.</li> </ul>
QVALWCY	QVC11	ALWCPYDTA setting (Y/N/O) <ul style="list-style-type: none"> <li>• Y - A copy of the data may be used.</li> <li>• N - Cannot use a copy of the data.</li> <li>• O - The optimizer can choose to use a copy of the data for performance.</li> </ul>
QVPSUDO	QVC12	Pseudo Open (Y/N) for SQL operations that can trigger opens. <ul style="list-style-type: none"> <li>• OP - Open</li> <li>• IN - Insert</li> <li>• UP - Update</li> <li>• DL - Delete</li> <li>• SI - Select Into</li> <li>• SV - Set</li> <li>• VI - Values into</li> </ul> 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"> <li>• CL - Close</li> <li>• IN - Insert</li> <li>• UP - Update</li> <li>• DL - Delete</li> <li>• SI - Select Into</li> <li>• SV - Set</li> <li>• VI - Values into</li> </ul> For all operations it can be blank.
QVODPI	QVC14	ODP implementation <ul style="list-style-type: none"> <li>• R - Reusable ODP</li> <li>• N - Nonreusable ODP</li> <li>• ' ' - Column not used</li> </ul>
QQDYNSC	QVC21	Dynamic replan, subtype reason code
QVCMMT	QVC41	Commitment control level. Possible values are: <ul style="list-style-type: none"> <li>• NC</li> <li>• UR</li> <li>• CS</li> <li>• CSKL</li> <li>• RS</li> <li>• RR</li> </ul>

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVBLKE	QVC15	Type of blocking . Possible value are: <ul style="list-style-type: none"> <li>• S - Single row, ALWBLK(*READ)</li> <li>• F - Force one row, ALWBLK(*NONE)</li> <li>• L - Limited block, ALWBLK(*ALLREAD)</li> </ul>
QVDLYPR	QVC16	Delay Prep (Y/N)
QVEXPLF	QVC1C	The SQL statement is explainable (Y/N).
QVNAMC	QVC17	Naming convention. Possibles values: <ul style="list-style-type: none"> <li>• N - System naming convention</li> <li>• S - SQL naming convention</li> </ul>
QVDYNTY	QVC18	Type of dynamic processing. <ul style="list-style-type: none"> <li>• E - Extended dynamic</li> <li>• S - System wide cache</li> <li>• L - Local prepared statement</li> </ul>
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"> <li>• N = User Profile is determined by naming conventions. For *SQL, USRPRF(*OWNER) is used. For *SYS, USRPRF(*USER) is used.</li> <li>• U = USRPRF(*USER) is used.</li> <li>• O = USRPRF(*OWNER) is used.</li> </ul>
QVDUSRP	QVC1B	User profile used for dynamic SQL statements. <ul style="list-style-type: none"> <li>• U = USRPRF(*USER) is used.</li> <li>• O = USRPRF(*OWNER) is used.</li> </ul>
QVDFTCL	QVC1281	Name of the default collection.
QVPROCN	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
QBINDTYPE	QQC18	Binding type: <ul style="list-style-type: none"> <li>• C - Column-wise binding</li> <li>• R - Row-wise binding</li> </ul>

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QCSRTYPE	QQC61	Cursor Type: <ul style="list-style-type: none"> <li>• NSA - Non-scrollable, asensitive, forward only</li> <li>• NSI - Non-scrollable, sensitive, forward only</li> <li>• NSS - Non-scrollable, insensitive, forward only</li> <li>• SCA - scrollable, asensitive</li> <li>• SCI - scrollable, sensitive</li> <li>• SCS - scrollable, insensitive</li> </ul>
QSQLSTMTO	QVC1D	SQL statement originator: <ul style="list-style-type: none"> <li>• U - User</li> <li>• S - System</li> </ul>
QCLSRC	QQC15	SQL cursor hardclose reason. Possible reasons are: <ul style="list-style-type: none"> <li>• 1 - Internal Error</li> <li>• 2 - Exclusive Lock</li> <li>• 3 - Interactive SQL Reuse Restriction</li> <li>• 4 - Host variable Reuse Restriction</li> <li>• 5 - Temporary Result Restriction</li> <li>• 6 - Cursor Restriction</li> <li>• 7 - Cursor Hard Close Requested</li> <li>• 8 - Internal Error</li> <li>• 9 - Cursor Threshold</li> <li>• A - Refresh Error</li> <li>• B - Reuse Cursor Error</li> <li>• C - DRDA AS Cursor Closed</li> <li>• D - DRDA AR Not WITH HOLD</li> <li>• E - Repeatable Read</li> <li>• F - Lock Conflict Or QSQPRCED Threshold - Library</li> <li>• G - Lock Conflict Or QSQPRCED Threshold - File</li> <li>• H - Execute Immediate Access Plan Space</li> <li>• I - QSQCSRTH Dummy Cursor Threshold</li> <li>• J - File Override Change</li> <li>• K - Program Invocation Change</li> <li>• L - File Open Options Change</li> <li>• M - Statement Reuse Restriction</li> <li>• N - Internal Error</li> <li>• O - Library List Changed</li> <li>• P - Exit Processing</li> </ul>
QCLSSUBRC	QQC23	SQL cursor hardclose reason subcode

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVDFMT	QVC42	Date Format. Possible values are: <ul style="list-style-type: none"> <li>• ISO</li> <li>• USA</li> <li>• EUR</li> <li>• JIS</li> <li>• MDY</li> <li>• DMY</li> <li>• YMD</li> </ul>
QVDSEP	QWC11	Date Separator. Possible values are: <ul style="list-style-type: none"> <li>• "/"</li> <li>• "."</li> <li>• ","</li> <li>• "-"</li> <li>• " "</li> </ul>
QVTFMT	QVC43	Time Format. Possible values are: <ul style="list-style-type: none"> <li>• ISO</li> <li>• USA</li> <li>• EUR</li> <li>• JIS</li> <li>• HMS</li> </ul>
QVTSEP	QWC12	Time Separator. Possible values are: <ul style="list-style-type: none"> <li>• ":"</li> <li>• "."</li> <li>• ","</li> <li>• " "</li> </ul>
QVDPNT	QWC13	Decimal Point. Possible values are: <ul style="list-style-type: none"> <li>• "."</li> <li>• ","</li> </ul>
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.

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVRAPRC	QVC22	<p>SQL access plan rebuild reason code. Possible reasons are:</p> <ul style="list-style-type: none"> <li>• 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"> <li>– Object was deleted and recreated.</li> <li>– Object was saved and restored.</li> <li>– Library list was changed.</li> <li>– Object was renamed.</li> <li>– Object was moved.</li> <li>– Object was overridden to a different object.</li> <li>– This is the first run of this query after the object containing the query has been restored.</li> </ul> </li> <li>• 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.</li> <li>• 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.</li> <li>• A4 - The number of rows in the table has changed by more than 10% since the access plan was last built.</li> <li>• A5 - A new index exists over one of the tables in the query</li> <li>• A6 - An index that was used for this access plan no longer exists or is no longer valid.</li> <li>• A7 - OS/400 Query requires the access plan to be rebuilt because of system programming changes.</li> <li>• A8 - The CCSID of the current job is different than the CCSID of the job that last created the access plan.</li> <li>• 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"> <li>– date format</li> <li>– date separator</li> <li>– time format</li> <li>– time separator.</li> </ul> </li> <li>• AA - The sort sequence table specified is different than the sort sequence table that was used when this access plan was created.</li> <li>• AB - Storage pool changed or DEGREE parameter of CHGQRYA command changed.</li> <li>• AC - The system feature DB2 multisystem has been installed or removed.</li> <li>• AD - The value of the degree query attribute has changed.</li> <li>• AE - A view is either being opened by a high level language or a view is being materialized.</li> </ul>

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVRAPRC (continued)	QVC22	<ul style="list-style-type: none"> <li>• 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.</li> <li>• B0 - The options specified have changed as a result of the query options file.</li> <li>• B1 - The access plan was generated with a commitment control level that is different in the current job.</li> <li>• B2 - The access plan was generated with a static cursor answer set size that is different than the previous access plan.</li> <li>• B3 - The query was reoptimized because this is the first run of the query after a prepare. That is, it is the first run with real actual parameter marker values.</li> <li>• B4 - The query was reoptimized because referential or check constraints have changed.</li> </ul>
QVNOSV	QVC24	<p>Access plan not saved reason code. Possible reasons are:</p> <ul style="list-style-type: none"> <li>• A1 - Failed to get a LSUP lock on associated space of program or package.</li> <li>• A2 - Failed to get an immediate LEAR space location lock on first byte of associated space of program.</li> <li>• A3 - Failed to get an immediate LENR space location lock on first byte of associated space of program.</li> <li>• A5 - Failed to get an immediate LEAR space location lock on first byte of ILE associated space of a program.</li> <li>• A6 - Error trying to extend space of an ILE program.</li> <li>• A7 - No room in program.</li> <li>• A8 - No room in program associated space.</li> <li>• A9 - No room in program associated space.</li> <li>• AA - No need to save. Save already done in another job.</li> <li>• AB - Query optimizer could not lock the QDT.</li> <li>• B1 - Saved at the end of the program associated space.</li> <li>• B2 - Saved at the end of the program associated space.</li> <li>• B3 - Saved in place.</li> <li>• B4 - Saved in place.</li> <li>• B5 - Saved at the end of the program associated space.</li> <li>• B6 - Saved in place.</li> <li>• B7 - Saved at the end of the program associated space.</li> <li>• B8 - Saved at the end of the program associated space.</li> </ul>
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)

Table 32. QQQ1000 - Summary row for SQL Information (continued)

Logical Column Name	Physical Column Name	Description
QVSQLCACHE	QWC18	Using system-wide SQL statement cache (Y/N)
QLOBTHRHD	QVP159	LOB locator threshold
QMAXPREC	QVP156	Maximum decimal precision (63/31)
QMAXSCLE	QVP157	Maximum decimal scale
QMINDIV	QVP158	Maximum decimal divide scale
NORM_DATA	QWC19	Unicode data normalization requested (Y/N)
QSTMTCMP	QQF2	Number of statement compressions which can occur before the access plan is removed.
QAPLENO	QVP15B	Access plan length prior to rebuild
QAPLENN	QVP15C	Access plan length after rebuild
QFSTDCNT	QVP151	Minimum row count needed to attempt fast delete
QREALUSR	QVC102	Real user name
QNTSSPID	QQC301	NTS lock space identifier

### 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      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      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	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	QQIDX	
A	QVQTBL	
A	QVQLIB	
A	QVPTBL	
A	QVPLIB	
A	QVBNDY	
A	QVRCNT	
A	QVJFANO	
A	QVFILES	
A	QVPPARPF	
A	QVPPARPL	
A	QVPPARD	
A	QVPPARU	
A	QVPPARRC	
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	QVRDTRG	RENAME(QQC18) + COLHDG('Read' + 'Trigger')
A	QVCARD	RENAME(QVP157) +

```

|          A          QVUTSP          COLHDG('Cardinality')
|                                     RENAME(QVC1281) +
|                                     COLHDG('Specific' +
|                                     'Name')
|          A          QVULSP          RENAME(QQC1282) +
|                                     COLHDG('Specific' +
|                                     'Schema')
|          A          PLSIZE_P        RENAME(QVP154) +
|                                     COLHDG('Pool' +
|                                     'Size')
|          A          POOLID          RENAME(QVP155) +
|                                     COLHDG('Pool' +
|                                     'ID')
|          A          QQMQT          RENAME(QQC13) +
|                                     COLHDG('Materialized' +
|                                     'Query Table')
|          A          K QQJFLD
|          A          S QQRID          CMP(EQ 3000)

```

Table 33. 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
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
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

Table 33. QQQ3000 - Summary Row for Table Scan (continued)

Logical Column Name	Physical Column Name	Description
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"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
QQIDXK	QQI2	Number of advised columns that use index scan-key positioning
QQDSS	QQDSS	dataspace selection <ul style="list-style-type: none"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
QQIDXA	QQIDXA	Index advised <ul style="list-style-type: none"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> <li>• T1 - No indexes exist.</li> <li>• T2 - Indexes exist, but none could be used.</li> <li>• T3 - Optimizer chose table scan over available indexes.</li> </ul>
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"> <li>• I - I/O bound</li> <li>• C - CPU bound</li> </ul>
QVRCNT	QVRCNT	Unique refresh counter

Table 33. QQQ3000 - Summary Row for Table Scan (continued)

Logical Column Name	Physical Column Name	Description
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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
PLSIZE_P	QVP154	Pool size
POOLID	QVP155	Pool id
QQMQT	QQC13	MQT table replaced query table (Y/N)

### 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      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	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' + 'Size')
	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	QVRDTRG	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	QQMQT	RENAME(QQC13) + COLHDG('Materialized' + 'Query Table')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3001)

Table 34. 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
QQSYS	QQSYS	System name
QQJOB	QQJOB	Job name
QQUSER	QQUSER	Job user
QQJNUM	QQJNUM	Job number

Table 34. QQQ3001 - Summary Row for Index Used (continued)

Logical Column Name	Physical Column Name	Description
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
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
QQREST	QQREST	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
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI1	dataspace number
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>



Table 34. QQQ3001 - Summary Row for Index Used (continued)

Logical Column Name	Physical Column Name	Description
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
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"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
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"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
QQDSS	QQDSS	dataspace selection <ul style="list-style-type: none"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
QQIDXA	QQIDXA	Index advised <ul style="list-style-type: none"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> <li>• I1 - Row selection</li> <li>• I2 - Ordering/Grouping</li> <li>• I3 - Row selection and Ordering/Grouping</li> <li>• I4 - Nested loop join</li> <li>• I5 - Row selection using bitmap processing</li> </ul>
QQIDXD	QQIDXD	Columns for index advised
QQCST	QQC11	Index is a constraint (Y/N)
QQCSTN	QQ1000	Constraint name
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"> <li>• I - I/O bound</li> <li>• C - CPU bound</li> </ul>
QVRCNT	QVRCNT	Unique refresh counter

Table 34. QQQ3001 - Summary Row for Index Used (continued)

Logical Column Name	Physical Column Name	Description
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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"> <li>• B - Binary Radix Index</li> <li>• C - Constraint (Binary Radix)</li> <li>• E - Encoded Vector Index (EVI)</li> <li>• X - Query created temporary index</li> </ul>
QVIDXUS	QVC12	Index Usage. Possible values are: <ul style="list-style-type: none"> <li>• P - Primary Index</li> <li>• T - Tertiary (AND/OR) Index</li> </ul>
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
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

Table 34. QQQ3001 - Summary Row for Index Used (continued)

Logical Column Name	Physical Column Name	Description
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
QQMQT	QQC13	MQT table replaced query table (Y/N)

### 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      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      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	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' +

A	QVTIXRZ	'of Index' + 'Created') 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' + 'Degree' + 'Limited')
A	QVKOA	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	QQMQT	RENAME(QQC13) + COLHDG('Materialized' + 'Query Table')
	A	K QQJFLD	
	A	S QQRID	CMP(EQ 3002)

| Table 35. 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
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 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
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

Table 35. QQQ3002 - Summary row for Index Created (continued)

Logical Column Name	Physical Column Name	Description
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"> <li>NL - Nested loop</li> <li>MF - Nested loop with selection</li> <li>HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>IN - Inner join</li> <li>PO - Left partial outer join</li> <li>EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>EQ - Equal</li> <li>NE - Not equal</li> <li>GT - Greater than</li> <li>GE - Greater than or equal</li> <li>LT - Less than</li> <li>LE - Less than or equal</li> <li>CP - Cartesian product</li> </ul>
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"> <li>Y - Yes</li> <li>N - No</li> </ul>
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"> <li>Y - Yes</li> <li>N - No</li> </ul>
QQDSS	QQDSS	dataspace selection <ul style="list-style-type: none"> <li>Y - Yes</li> <li>N - No</li> </ul>
QQIDXA	QQIDXA	Index advised <ul style="list-style-type: none"> <li>Y - Yes</li> <li>N - No</li> </ul>
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> <li>I1 - Row selection</li> <li>I2 - Ordering/Grouping</li> <li>I3 - Row selection and Ordering/Grouping</li> <li>I4 - Nested loop join</li> </ul>
QQIDXN	QQIDXN	Key columns for index advised
QQCRTK	QQ1000	Key columns for index created



| Table 35. QQQ3002 - Summary row for Index Created (continued)

Logical Column Name	Physical Column Name	Description
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"> <li>• I - I/O bound</li> <li>• C - CPU bound</li> </ul>
QVRCNT	QVRCNT	Unique refresh counter
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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)
QVTIXTY	QVC1F	Type of index created. Possible values: <ul style="list-style-type: none"> <li>• B - Binary Radix Index</li> <li>• E - Encoded Vector Index (EVI)</li> </ul>
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)

| Table 35. QQQ3002 - Summary row for Index Created (continued)

Logical Column Name	Physical Column Name	Description
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: <ul style="list-style-type: none"> <li>• B - Binary Radix Index</li> <li>• C - Constraint (Binary Radix)</li> <li>• E - Encoded Vector Index (EVI)</li> <li>• T - Tertiary (AND/OR) Index</li> </ul>
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)
QQMQT	QQC13	MQT table replaced query table (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                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      QQDEF
A      QQDTN
A      QQDTL

```

A	QQMATN	
A	QQMATL	
A	QQMATULVL	RENAME(QVP15E) + COLHDG('Materialized' + 'Union' + 'Level')
A	QQQDTN	RENAME(QVP15A) + COLHDG('Decomposed' + 'Subselect' + 'Number')
A	QQQDTR	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	QQRCOD	
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 36. 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
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
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 36. QQQ3003 - Summary Row for Query Sort (continued)

Logical Column Name	Physical Column Name	Description
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> <li>• 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.</li> <li>• 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.</li> <li>• F3 - The grouping and ordering columns are not compatible.</li> <li>• F4 - DISTINCT was specified for the query.</li> <li>• F5 - UNION was specified for the query.</li> <li>• 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.</li> <li>• F7 - Query optimizer chose to use a sort rather than an index to order the results of the query.</li> <li>• F8 - Perform specified row selection to minimize I/O wait time.</li> <li>• FC - The query contains grouping fields and there is a read trigger on at least one of the physical files in the query.</li> </ul>
QQRCSUB	QQI7	Reason subcode for Union: <ul style="list-style-type: none"> <li>• 51 - Query contains UNION and ORDER BY</li> <li>• 52 - Query contains UNION ALL</li> </ul>
QVBNDY	QVBNDY	I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> <li>• I - I/O bound</li> <li>• C - CPU bound</li> </ul>
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"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>

Table 36. 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"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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      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	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	QVTRSZ	RENAME(QQI2) + COLHDG('Row Size' + 'Temporary' + 'Table')
A	QVTSIZ	RENAME(QQI3) + COLHDG('Table Size' + 'Temporary' + 'Table')
A	QVTRST	RENAME(QQC12) + COLHDG('Temporary' + 'Result')
A	QVTDST	RENAME(QQC13) + COLHDG('Distributed' + 'Table')
A	QVTTNOD	RENAME(QVC3001) + COLHDG('Data' +

A	QMATDLVL	'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 37. 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
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 base table for table queried
QQPTMN	QQPTMN	Member name of base table
QQSTIM	QQSTIM	Start timestamp
QQETIM	QQETIM	End timestamp



Table 37. 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"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
QQTMPR	QQTMPR	Number of rows in the temporary
QQRCOD	QQRCOD	Reason code. Possible values are: <ul style="list-style-type: none"> <li>• 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.</li> <li>• 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.</li> <li>• F3 - The grouping and ordering columns are not compatible.</li> <li>• F4 - DISTINCT was specified for the query.</li> <li>• F5 - UNION was specified for the query.</li> <li>• 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.</li> <li>• F7 - Query optimizer chose to use a sort rather than an index to order the results of the query.</li> <li>• F8 - Perform specified row selection to minimize I/O wait time.</li> <li>• F9 - The query optimizer chose to use a hashing algorithm rather than an index to perform the grouping.</li> <li>• FA - The query contains a join condition that requires a temporary table</li> <li>• FB - The query optimizer creates a run-time temporary file in order to implement certain correlated group by queries.</li> <li>• FC - The query contains grouping fields and there is a read trigger on at least one of the physical files in the query.</li> <li>• FD - The query optimizer creates a runtime temporary file for a static-cursor request.</li> <li>• H1 - Table is a join logical file and its join type does not match the join type specified in the query.</li> <li>• H2 - Format specified for the logical table references more than one base table.</li> <li>• H3 - Table is a complex SQL view requiring a temporary table to contain the results of the SQL view.</li> <li>• H4 - For an update-capable query, a subselect references a column in this table which matches one of the columns being updated.</li> <li>• H5 - For an update-capable query, a subselect references an SQL view which is based on the table being updated.</li> <li>• 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</li> <li>• H7 - A user-defined table function was materialized.</li> </ul>
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

Table 37. QQQ3004 - Summary Row for Temp Table (continued)

Logical Column Name	Physical Column Name	Description
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"> <li>• I - I/O bound</li> <li>• C - CPU bound</li> </ul>
QVRCNT	QVRCNT	Unique refresh counter
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
QVTRSZ	QQI2	Row size of temporary table, in bytes
QVTSIZ	QQI3	Size of temporary table, in bytes

Table 37. QQQ3004 - Summary Row for Temp Table (continued)

Logical Column Name	Physical Column Name	Description
QVTRST	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      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      QQLCKF          RENAME(QQC11) +
                        COLHDG('Lock' +

```

A	QQULCK	'Indicator') 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 38. 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
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
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

Table 38. QQQ3005 - Summary Row for Table Locked (continued)

Logical Column Name	Physical Column Name	Description
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"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
QQULCK	QQC12	Unlock request <ul style="list-style-type: none"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
QQRCOD	QQRCOD	Reason code <ul style="list-style-type: none"> <li>• L1 - UNION with *ALL or *CS with Keep Locks</li> <li>• L2 - DISTINCT with *ALL or *CS with Keep Locks</li> <li>• L3 - No duplicate keys with *ALL or *CS with Keep Locks</li> <li>• L4 - Temporary needed with *ALL or *CS with Keep Locks</li> <li>• L5 - System Table with *ALL or *CS with Keep Locks</li> <li>• L6 - Orderby &gt; 2000 bytes with *ALL or *CS with Keep Locks</li> <li>• L9 - Unknown</li> <li>• LA - User-defined table function with *ALL or *CS with Keep Locks</li> </ul>
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"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>

Table 38. 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"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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      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      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      QQINLN
A      QQINFN
A      QQRCD
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 39. 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
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

Table 39. 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"> <li>• 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"> <li>– Object was deleted and recreated.</li> <li>– Object was saved and restored.</li> <li>– Library list was changed.</li> <li>– Object was renamed.</li> <li>– Object was moved.</li> <li>– Object was overridden to a different object.</li> <li>– This is the first run of this query after the object containing the query has been restored.</li> </ul> </li> <li>• 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.</li> <li>• 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.</li> <li>• A4 - The number of rows in the table has changed by more than 10% since the access plan was last built.</li> <li>• A5 - A new index exists over one of the tables in the query</li> <li>• A6 - An index that was used for this access plan no longer exists or is no longer valid.</li> <li>• A7 - OS/400 Query requires the access plan to be rebuilt because of system programming changes.</li> <li>• A8 - The CCSID of the current job is different than the CCSID of the job that last created the access plan.</li> <li>• 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"> <li>– date format</li> <li>– date separator</li> <li>– time format</li> <li>– time separator.</li> </ul> </li> <li>• AA - The sort sequence table specified is different than the sort sequence table that was used when this access plan was created.</li> <li>• AB - Storage pool changed or DEGREE parameter of CHGQRYA command changed.</li> <li>• AC - The system feature DB2 multisystem has been installed or removed.</li> <li>• AD - The value of the degree query attribute has changed.</li> <li>• AE - A view is either being opened by a high level language or a view is being materialized.</li> <li>• AF - A sequence object or user-defined type or function is not the same object as the one referred to in the access plan; or, the SQL path used to generate the access plan is different than the current SQL path.</li> </ul>



Table 39. QQQ3006 - Summary Row for Access Plan Rebuilt (continued)

Logical Column Name	Physical Column Name	Description
QQRCOD (continued)	QQRCOD	<ul style="list-style-type: none"> <li>• B0 - The options specified have changed as a result of the query options file.</li> <li>• B1 - The access plan was generated with a commitment control level that is different in the current job.</li> <li>• B2 - The access plan was generated with a static cursor answer set size that is different than the previous access plan.</li> <li>• B3 - The query was reoptimized because this is the first run of the query after a prepare. That is, it is the first run with real actual parameter marker values.</li> <li>• B4 - The query was reoptimized because referential or check constraints have changed.</li> <li>• B5 - The query was reoptimized because the MQT was no longer eligible to be used. Possible reasons are: <ul style="list-style-type: none"> <li>– The MQT no longer exists</li> <li>– A new MQT was found,</li> <li>– The enable/disable query optimization changed</li> <li>– Time since the last REFRESH TABLE exceeds the MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option</li> <li>– Other QAQQINI options no longer match.</li> </ul> </li> </ul>
QVSUBRC	QQC21	If the access plan rebuild reason code was A7 this two-byte hex value identifies which specific reason for A7 forced a rebuild.
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"> <li>• Y - Yes, plan was really optimized.</li> <li>• N - No, the plan was not reoptimized because of the QAQQINI option for the REOPTIMIZE_ACCESS_PLAN parameter value</li> </ul>
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      R QQQ3007          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	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 40. 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
QQSYS	QQSYS	System name

Table 40. QQQ3007 - Summary Row for Optimizer Timed Out (continued)

Logical Column Name	Physical Column Name	Description
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
QQIDXN	QQ1000	Index names
QQTOUT	QQC11	Optimizer timed out <ul style="list-style-type: none"> <li>• Y - Yes</li> <li>• N - No</li> </ul>
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"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>

Table 40. QQQ3007 - Summary Row for Optimizer Timed Out (continued)

Logical Column Name	Physical Column Name	Description
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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      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	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 41. 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
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

Table 41. QQQ3008 - Summary Row for Subquery Processing (continued)

Logical Column Name	Physical Column Name	Description
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      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      QQODPI          RENAME(QQC11) +
                        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 42. 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)

Table 42. QQQ3010 - Summary for HostVar & ODP Implementation (continued)

Logical Column Name	Physical Column Name	Description
QQRcnt	QQI5	Unique refresh counter
QQUDEF	QQUDEF	User defined column
QQODPI	QQC11	ODP implementation <ul style="list-style-type: none"> <li>• R - Reusable ODP</li> <li>• N - Nonreusable ODP</li> <li>• ' ' - Column not used</li> </ul>
QQHVI	QQC12	Host variable implementation <ul style="list-style-type: none"> <li>• I - Interface supplied values (ISV)</li> <li>• V - Host variables treated as constants (V2)</li> <li>• U - Table management row positioning (UP)</li> </ul>
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 QQQ3014          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      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      QQREST
  
```

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	QQRCDS	
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	QVNLSSST	RENAME(QVC105) +



A	QVNLSSL	COLHDG('Sort' + 'Sequence' + 'Table') 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(QVC16) + 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*		
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) +

A	QVPMCNV	COLHDG('Display' + 'DEBUG' + 'Messages') 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' + 'QDT')
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	NORM_DATA	RENAME(QQC21) + COLHDG('Unicode' + 'Data'+ 'Normalization')
A	PL_SIZE_FS	RENAME(QVP153) + COLHDG('Pool size' + 'Fair Share')
A	FRCJORD	RENAME(QQC28) + COLHDG('Force' + 'Join' + 'Order')
A	FRCJORDDS	RENAME(QVP152) + COLHDG('Force' + 'Primary' + 'File')
A	PMCONVRC	RENAME(QQ16) + COLHDG('Parameter Mark' + 'Conversion' + 'Reason Code')
A	POOLID	

```

|      A          QQMQTR          RENAME(QQ17) +
|      |          |          COLHDG('Materialized' +
|      |          |          'Query Table' +
|      |          |          'Refresh')
|      A          QQMQTU          RENAME(QVC42) +
|      |          |          COLHDG('Materialized' +
|      |          |          'Query Table' +
|      |          |          'Usage')
|      A          K QQJFLD
|      A          S QQRID          CMP(EQ 3014)

```

Table 43. 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
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
QQREST	QQREST	Estimated number of rows selected
QQEPT	QQEPT	Estimated processing time, in seconds
QQQTIM	QQI1	Time spent to open cursor, in milliseconds
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"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQUNIN	QQUNIN	Union Query (Y/N)

Table 43. QQQ3014 - Summary Row for Generic QQ Information (continued)

Logical Column Name	Physical Column Name	Description
QQSUBQ	QQSUBQ	Subquery (Y/N)
QQSSUB	QWC1F	Scalar Subselects (Y/N)
QQHSTV	QQHSTV	Host variables (Y/N)
QQRCDSD	QQRCDSD	Row selection (Y/N)
QQGVNE	QQC11	Query governor enabled (Y/N)
QQGVNS	QQC12	Query governor stopped the query (Y/N)
QQOPIID	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"> <li>• I - Index</li> <li>• S - Sort</li> </ul>
QVGBYIM	QVC12	Grouping implementation. Possible values are: <ul style="list-style-type: none"> <li>• I - Index</li> <li>• H - Hash grouping</li> </ul>
QVJONIM	QVC13	Join Implementation. Possible values are: <ul style="list-style-type: none"> <li>• N - Nested Loop join</li> <li>• H - Hash join</li> <li>• C - Combination of Nested Loop and Hash</li> </ul>
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.
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

Table 43. QQQ3014 - Summary Row for Generic QQ Information (continued)

Logical Column Name	Physical Column Name	Description
QVPARA	QVC81	Parallel Degree <ul style="list-style-type: none"> <li>• *SAME - Don't change current setting</li> <li>• *NONE - No parallel processing is allowed</li> <li>• *I/O - Any number of tasks may be used for I/O processing. SMP parallel processing is not allowed.</li> <li>• *OPTIMIZE - The optimizer chooses the number of tasks to use for either I/O or SMP parallel processing.</li> <li>• *MAX - The optimizer chooses to use either I/O or SMP parallel processing.</li> <li>• *SYSVAL - Use the current system value to process the query.</li> <li>• *ANY - Has the same meaning as *I/O.</li> <li>• *NBRTASKS - The number of tasks for SMP parallel processing is specified in column QVTASKN.</li> </ul>
QVTASKN	QQI3	Max number of tasks
QVAPLYR	QVC17	Apply CHGQRYA remotely (Y/N)
QVASYNC	QVC82	Asynchronous job usage <ul style="list-style-type: none"> <li>• *SAME - Don't change current setting</li> <li>• *DIST - Asynchronous jobs may be used for queries with distributed tables</li> <li>• *LOCAL - Asynchronous jobs may be used for queries with local tables only</li> <li>• *ANY - Asynchronous jobs may be used for any database query</li> <li>• *NONE - No asynchronous jobs are allowed</li> </ul>
QVFRCJO	QVC18	Force join order (Y/N)
QVDMGS	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"> <li>• *PERCENT followed by 2 byte integer containing the percent value</li> <li>• *MAX_NUMBER_OF_RECORDS followed by an integer value that represents the maximum number of rows</li> </ul>
QVREOPT	QVC1E	Reoptimize access plan requested. Possible values are: <ul style="list-style-type: none"> <li>• 'O' - Only reoptimize the access plan when absolutely required. Do not reoptimize for subjective reasons.</li> <li>• 'Y' - Yes, force the access plan to be reoptimized.</li> <li>• 'N' - No, do not reoptimize the access plan, unless optimizer determines that it is necessary. May reoptimize for subjective reasons.</li> </ul>
QVOPALL	QVC87	Optimize all indexes requested <ul style="list-style-type: none"> <li>• *SAME - Don't change current setting</li> <li>• *YES - Examine all indexes</li> <li>• *NO - Allow optimizer to time-out</li> <li>• *TIMEOUT - Force optimizer to time-out</li> </ul>
QVDFQDTF	QQC14	Final decomposed QDT built indicator (Y/N)

Table 43. QQQ3014 - Summary Row for Generic QQ Information (continued)

Logical Column Name	Physical Column Name	Description
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"> <li>*NO - Star join optimization will not be performed.</li> <li>*COST - The optimizer will determine if any EVIs can be used for star join optimization.</li> <li>*FORCE - The optimizer will add any EVIs that can be used for star join optimization.</li> </ul>
OPTGOAL	QVC23	Byte 1 = Optimization goal. Possible values are: <ul style="list-style-type: none"> <li>'F' - First I/O, optimize the query to return the first screen full of rows as quickly as possible.</li> <li>'A' - All I/O, optimize the query to return all rows as quickly as possible.</li> </ul>
DIAGLIKE	QVC24	Byte 1 = Type of Visual Explain diagram. Possible values are: <ul style="list-style-type: none"> <li>'D' - Detail</li> <li>'B' - Basic</li> </ul> Byte 2 - Ignore LIKE redundant shifts. Possible values are: <ul style="list-style-type: none"> <li>'O' - Optimize, the query optimizer determines which redundant shifts to ignore.</li> <li>'A' - All, all redundant shifts will be ignored.</li> </ul>
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)
Norm_Data	QQC21	Unicode data normalization requested (Y/N)
PLSize_FS	QVP153	Fair share of the pool size as determined by the optimizer
FrcJOrd	QQC82	Force Join Order requested. Possible values are: <ul style="list-style-type: none"> <li>*NO - The optimizer was allowed to reorder join files</li> <li>*YES - The optimizer was not allowed to reorder join files as part of its optimization process</li> <li>*SQL - The optimizer only forced the join order for those queries that used the SQL JOIN syntax</li> <li>*PRIMARY - The optimizer was only required to force the primary dial for the join.</li> </ul>

Table 43. QQQ3014 - Summary Row for Generic QQ Information (continued)

Logical Column Name	Physical Column Name	Description
PMConvRC	QQI6	Reason code for why Parameter Marker Conversion was not performed: <ul style="list-style-type: none"> <li>• Argument of function must be a literal</li> <li>• LOCALTIME or LOCALTIMESTAMP</li> <li>• Duration literal in arithmetic expression</li> <li>• UPDATE query with no WHERE clause</li> <li>• BLOB literal</li> <li>• Special register in UPDATE or INSERT with values</li> <li>• Result expression for CASE</li> <li>• GROUP BY expression</li> <li>• ESCAPE character</li> <li>• Double Negative value -(-1)</li> <li>• INSERT or UPDATE with a mix of literals, parameter markers, and NULLs</li> <li>• UPDATE with a mix of literals and parameter markers</li> <li>• INSERT with VALUES containing NULL value and expressions</li> <li>• UPDATE with list of expressions</li> </ul>
QQMQTR	QQI7	Value of the MATERIALIZED_QUERY_TABLE_REFRESH_AGE duration. If the QAQQINI parameter value is set to *ANY, the timestamp duration will be 99999999999999.
QQMQTU	QVC42	Byte 1 - Contains the MATERIALIZED_QUERY_TABLE_USAGE. Possible values are: <ul style="list-style-type: none"> <li>• 'N' - *NONE - no materialized query tables used in query optimization and implementation</li> <li>• 'A' - User-maintained. Refresh-deferred query tables can be used.</li> <li>• 'U' - Only user-maintained materialized query tables can be used.</li> </ul>

### 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      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' +

```

A	QQQDTN	'Union' + 'Level') 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	QVQTB	
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 44. QQQ3015 - Summary Row for Statistic 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
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



Table 44. QQQ3015 - Summary Row for Statistic Information (continued)

Logical Column Name	Physical Column Name	Description
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"> <li>• 'N' - No statistic</li> <li>• 'S' - Stale statistic</li> <li>• '' - Unknown</li> </ul>
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 45. QQQ3018 - Summary Row for STRDBMON/ENDDBMON

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"> <li>• C - Current</li> <li>• J - Job name</li> <li>• A - All</li> </ul>
QQCMDT	QQC12	Command type <ul style="list-style-type: none"> <li>• S - STRDBMON</li> <li>• E - ENDDDBMON</li> </ul>
QQJOBI	QQC301	Monitored job information <ul style="list-style-type: none"> <li>• * - Current job</li> <li>• Job number/User/Job name</li> <li>• *ALL - All jobs</li> </ul>

### 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(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	QQCPUT	RENAME(QQI1) + COLHDG('Row' + 'Retrieval' + 'CPU Time')
A	QQCLKT	RENAME(QQI2) + 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 46. 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
QQSYS	QQSYS	System name

Table 46. QQQ3019 - Detail Row for Rows Retrieved (continued)

Logical Column Name	Physical Column Name	Description
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
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	QVBMCNT	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 47. 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)

Table 47. QQQ3021 - Summary Row for Bitmap Created (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
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
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"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>

Table 47. QQQ3021 - Summary Row for Bitmap Created (continued)

Logical Column Name	Physical Column Name	Description
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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      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      QQQDTN                RENAME(QVP15A) +
                              COLHDG('Decomposed' +
                              'Subselect' +
                              'Number')

A      QQQDTT                RENAME(QVP15B) +

```

		COLHDG('Number of' + 'Decomposed' + 'Subselects')
A	QQDQTR	RENAME(QVP15C) + COLHDG('Decomposed' + 'Reason' + 'Code')
A	QQDQTS	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 48. 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)
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



Table 48. QQQ3022 - Summary Row for Bitmap Merge (continued)

Logical Column Name	Physical Column Name	Description
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"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>

Table 48. QQQ3022 - Summary Row for Bitmap Merge (continued)

Logical Column Name	Physical Column Name	Description
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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          QQDEF
A          QQDTN
A          QQDTL
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	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 49. 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
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
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"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>

Table 49. QQQ3023 - Summary for Temp Hash Table Created (continued)

Logical Column Name	Physical Column Name	Description
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
QVFILES	QVFILES	Number of tables joined
QVHTRC	QVC1F	Hash table reason code <ul style="list-style-type: none"> <li>• J - Created for hash join</li> <li>• G - Created for hash grouping</li> </ul>
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	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	K QQJFLD	
A	S QQRID	CMP(EQ 3025)

Table 50. QQQ3025 - Summary Row for Distinct 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

Table 50. QQQ3025 - Summary Row for Distinct Processing (continued)

Logical Column Name	Physical Column Name	Description
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      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      QQMATULVL      RENAME(QVP15E) +
                       COLHDG('Materialized' +
                               'Union' +
                               'Level')

A      QDQDTN          RENAME(QVP15A) +

```

A	QDQDTT	COLHDG('Decomposed' + 'Subselect' + 'Number') 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(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) + COLHDG('View' + 'Level' + 'Inner' + 'Subselect')
A	QVIMATUL	RENAME(QSP155) + COLHDG('Materialized' + 'Union' + 'of Inner')
A	QVSUBOP	RENAME(QQC101) + COLHDG('Subquery' + 'Operator')
A	QVSUBTYP	RENAME(QVC21) + COLHDG('Subquery' + 'Type')
A	QVCORRI	RENAME(QQC11) + COLHDG('Correlated' + 'Columns' +



A	QVCORRC	'Exist') RENAME(QVC3001) + COLHDG('Correlated' + 'Columns')
A	K QQJFLD	
A	S QQRID	CMP(EQ 3027)

Table 51. QQQ3027 - Summary Row for Subquery 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)
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	Subselect number for outer subquery
QQQDTL	QQQDTL	Subselect level for outer subquery
QQMATN	QQMATN	Materialized view subselect number for outer subquery
QQMATL	QQMATL	Materialized view subselect level for outer subquery
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
QQAJN	QQAJN	Estimated number of joined rows
QQJNP	QQJNP	Join position - when available
QQJNDS	QQI6	dataspace number/Original table position

Table 51. QQQ3027 - Summary Row for Subquery Merge (continued)

Logical Column Name	Physical Column Name	Description
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
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
QVSUBOP	QQC101	Subquery operator. Possible values are: <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not Equal</li> <li>• LT - Less Than or Equal</li> <li>• LT - Less Than</li> <li>• GE - Greater Than or Equal</li> <li>• GT - Greater Than</li> <li>• IN</li> <li>• LIKE</li> <li>• EXISTS</li> <li>• NOT - Can precede IN, LIKE or EXISTS</li> </ul>
QVSSUBTYP	QVC21	Subquery type. Possible values are: <ul style="list-style-type: none"> <li>• SQ - Subquery</li> <li>• SS - Scalar subselect</li> <li>• SU - Set Update</li> </ul>

Table 51. QQQ3027 - Summary Row for Subquery Merge (continued)

Logical Column Name	Physical Column Name	Description
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      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      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	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' +

```

A          QVGBYVAR          'Columns')
          RENAME(QVC3008) +
          COLHDG('VAR' +
          'Columns')
A          K QQJFLD
A          S QQRID          CMP(EQ 3028)

```

Table 52. 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
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

Table 52. QQQ3028 - Summary Row for Grouping (continued)

Logical Column Name	Physical Column Name	Description
QQJNMT	QQC21	Join method - when available <ul style="list-style-type: none"> <li>• NL - Nested loop</li> <li>• MF - Nested loop with selection</li> <li>• HJ - Hash join</li> </ul>
QQJNTY	QQC22	Join type - when available <ul style="list-style-type: none"> <li>• IN - Inner join</li> <li>• PO - Left partial outer join</li> <li>• EX - Exception join</li> </ul>
QQJNOP	QQC23	Join operator - when available <ul style="list-style-type: none"> <li>• EQ - Equal</li> <li>• NE - Not equal</li> <li>• GT - Greater than</li> <li>• GE - Greater than or equal</li> <li>• LT - Less than</li> <li>• LE - Less than or equal</li> <li>• CP - Cartesian product</li> </ul>
QVJFANO	QVJFANO	Join fan out. Possible values are: <ul style="list-style-type: none"> <li>• N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned.</li> <li>• D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned.</li> <li>• U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs.</li> </ul>
QVFILES	QVFILES	Number of tables joined
QVGBYI	QQC11	Groupby implementation <ul style="list-style-type: none"> <li>• ' ' - No grouping</li> <li>• I - Index</li> <li>• H - Hash</li> </ul>
QVGBYIT	QQC15	Type of Index. Possible values are: <ul style="list-style-type: none"> <li>• B - Binary Radix Index</li> <li>• C - Constraint (Binary Radix)</li> <li>• E - Encoded Vector Index (EVI)</li> <li>• X - Query created temporary index</li> </ul>
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

Table 52. QQQ3028 - Summary Row for Grouping (continued)

Logical Column Name	Physical Column Name	Description
QVGBYMAX	QVC3003	MAX columns
QVGBYSUM	QVC3004	SUM columns
QVGBYCNT	QVC3005	COUNT columns
QVGBYAVG	QVC3006	AVG columns
QVGBYSTD	QVC3007	STDDEV columns
QVGBYVAR	QVC3008	VAR columns

**Database monitor logical table 3030 - Summary Row for Materialized query tables**

```
| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
| A*
| A* DB Monitor logical table 3030 - Summary Row for Materialized query table
| A*
| A* New row added for Visual Explain
| A*
| A          R QQQ3030          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          QDQDTS          RENAME(QVP15D) +
|                               COLHDG('Starting' +
|                                   'Decomposed' +
|                                   'Subselect')
|
| A          QQMQTN          RENAME(QQ1000) +
|                               COLHDG('Materialized' +
|                                   'Query Table' +
|                                   'Usage')
|
| A          QQISRN          RENAME(QQC301) +
|                               COLHDG('Materialized' +
```

```

|                                     'Query Table' +
|                                     'Reason code')
|
|      A      QVRCNT
|      A      K QQJFLD
|      A      S QQRID
|                                     CMP(EQ 3030)

```

Table 53. QQQ3030 - Summary Row for Materialized query tables

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
QQMATUL	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 53. QQQ3030 - Summary Row for Materialized query tables (continued)

Logical Column Name	Physical Column Name	Description
QQMQTN	QQ1000	<p>Materialized query tables examined and reason why used or not used:</p> <ul style="list-style-type: none"> <li>• 0 - The materialized query table was used</li> <li>• 1 - The cost to use the materialized query table, as determined by the optimizer, was higher than the cost associated with the chosen implementation.</li> <li>• 2 - The join specified in the materialized query was not compatible with the query.</li> <li>• 3 - The materialized query table had predicates that were not matched in the query.</li> <li>• 4 - The grouping specified in the materialized query table is not compatible with the grouping specified in the query.</li> <li>• 5 - The query specified columns that were not in the select-list of the materialized query table.</li> <li>• 6 - The materialized query table query contains functionality that is not supported by the query optimizer.</li> <li>• 7 - The materialized query table specified the DISABLE QUERY OPTIMIZATION clause.</li> <li>• 8 - The ordering specified in the materialized query table is not compatible with the ordering specified in the query.</li> <li>• 9 - The query contains functionality that is not supported by the materialized query table matching algorithm.</li> <li>• 10 - Materialized query tables may not be used for this query.</li> <li>• 11 - The refresh age of this materialized query table exceeds the duration specified by the MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option.</li> <li>• 12 - The commit level of the materialized query table is lower than the commit level specified for the query.</li> <li>• 13 - The distinct specified in the materialized query table is not compatible with the distinct specified in the query.</li> <li>• 14 - The FETCH FOR FIRST n ROWS clause of the materialized query table is not compatible with the query.</li> <li>• 15 - The QAQQINI options used to create the materialized query table are not compatible with the QAQQINI options used to run this query.</li> <li>• 16 - The materialized query table is not usable.</li> </ul>
QQISRN	QQC301	List of unique reason codes used by the materialized query tables (each materialized query table has a corresponding reason code associated with it)
QVRCNT	QVRCNT	Unique refresh counter

## 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” on page 253
- “External table description (QAQQTEXT) - Summary Row for SQL Statement” on page 259
- “External table description (QAQQ3000) - Summary Row for Arrival Sequence” on page 259

- “External table description (QAQQ3001) - Summary row for Using Existing Index” on page 261
- “External table description (QAQQ3002) - Summary Row for Index Created” on page 263
- “External table description (QAQQ3003) - Summary Row for Query Sort” on page 265
- “External table description (QAQQ3004) - Summary Row for Temporary Table” on page 266
- “External table description (QAQQ3007) - Summary Row for Optimizer Information” on page 268
- “External table description (QAQQ3008) - Summary Row for Subquery Processing” on page 269
- “External table description (QAQQ3010) - Summary Row for Host Variable and ODP Implementation” on page 269

## External table description (QAQQRYI) - Summary Row for SQL Information

Table 54. QAQQRYI - 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
QQAUGT	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)
QQOTHT	Other time for most expensive execution (ms)
QQLTU	Time statement last used
QQMETU	Most expensive time used
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

| *Table 54. QAQQRYI - Summary Row for SQL Information (continued)*

Column Name	Description
<b>QQRROW</b>	<b>Statement function</b>
	S - Select - Update
	I - Insert
	D - Delete
	L - Data definition language
	O - Other

Table 54. QAQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQSTOP	<b>Statement operation</b>
	• AL - Alter table
	• AQ - Alter sequence
	• 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
	• CQ - Create sequence
	• 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

| Table 54. QAQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQSTOP (continued)	<ul style="list-style-type: none"> <li>• RF - Refresh Table</li> <li>• RO - Rollback</li> <li>• RS - Release Savepoint</li> <li>• RT - Rename table</li> <li>• RV - Revoke</li> <li>• SA - Savepoint</li> <li>• SC - Set connection</li> <li>• SE - Set encryption password</li> <li>• SI - Select into</li> <li>• SP - Set path</li> <li>• SR - Set result set</li> <li>• SS - Set current schema</li> <li>• ST - Set transaction</li> <li>• SV - Set variable</li> <li>• UP - Update</li> <li>• VI - Values into</li> </ul>
QQODPI	<p><b>ODP implementation</b></p> <ul style="list-style-type: none"> <li>R - Reusable ODP (ISV)</li> <li>N - Non-reusable ODP (V2)</li> </ul>
QQHVI	<p><b>Host variable implementation</b></p> <ul style="list-style-type: none"> <li>I - Interface supplied values (ISV)</li> <li>V - Host variables treated as constants (V2)</li> <li>U - Table management row positioning (UP)</li> </ul>

Table 54. QAQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQAPR	<b>Access plan rebuilt</b>
	<p><b>A1</b> 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"> <li>• Object was deleted and recreated.</li> <li>• Object was saved and restored.</li> <li>• Library list was changed.</li> <li>• Object was renamed.</li> <li>• Object was moved.</li> <li>• Object was overridden to a different object.</li> <li>• This is the first run of this query after the object containing the query has been restored.</li> </ul>
	<p><b>A2</b> 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><b>A3</b> 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><b>A4</b> The number of rows in the table has changed by more than 10% since the access plan was last built.</p>
	<p><b>A5</b> A new index exists over one of the tables in the query.</p>
	<p><b>A6</b> An index that was used for this access plan no longer exists or is no longer valid.</p>
	<p><b>A7</b> OS/400 Query requires the access plan to be rebuilt because of system programming changes.</p>
	<p><b>A8</b> The CCSID of the current job is different than the CCSID of the job that last created the access plan.</p>
	<p><b>A9</b> 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"> <li>• date format</li> <li>• date separator</li> <li>• time format</li> <li>• time separator</li> </ul>

Table 54. 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 sequence object or user-defined type or function is not the same object as the one referred to in the access plan; or, the SQL path used to generate the access plan is different than the current SQL path.
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.
B3	The query was reoptimized because this is the first run of the query after a prepare. That is, it is the first run with real actual parameter marker values
B4	The query was reoptimized because referential or check constraints have changed
<b>QQDACV</b>	<b>Data conversion</b>
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.
<b>QQCTS</b>	<b>Statement table scan usage count</b>
<b>QQCIU</b>	<b>Statement index usage count</b>
<b>QQCIC</b>	<b>Statement index creation count</b>
<b>QQCSO</b>	<b>Statement sort usage count</b>
<b>QQCTF</b>	<b>Statement temporary table count</b>
<b>QQCIA</b>	<b>Statement index advised count</b>
<b>QQCAPR</b>	<b>Statement access plan rebuild count</b>

Table 54. QAQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQARSS	Average result set size
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQC1000	Reserved

## External table description (QAQQTEXT) - Summary Row for SQL Statement

Table 55. 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
QQSTX	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 56. 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



Table 56. QAQQ3000 - Summary Row for Arrival Sequence (continued)

Column Name	Description
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQPTLN	Physical library
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
QQC22	Reserved

Table 56. QAQQ3000 - Summary Row for Arrival Sequence (continued)

Column Name	Description
QQI1	Number of advised key columns that use index scan-key positioning.
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

## External table description (QAQQ3001) - Summary row for Using Existing Index

Table 57. 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 57. QQQ3001 - Summary Row for Using Existing Index (continued)

Column Name	Description
QQJNOP	<b>Join operator - when available</b>  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	<b>Number of advised key columns that use index scan-key positioning</b>
QQKP	<b>Index scan-key positioning</b>  Y - Yes N - No
QQKPN	<b>Number of key positioning columns</b>
QQKS	<b>Index scan-key selection</b>  Y - Yes N - No
QQDSS	<b>Dataspace selection</b>  Y - Yes N - No
QQIDXA	<b>Index advised</b>  Y - Yes N - No
QQRCOD	<b>Reason code</b>  I1 - Row selection I2 - Ordering/Grouping I3 - Row selection and Ordering/Grouping I4 - Nested loop join I5 - Row selection using bitmap processing
QQCST	<b>Constraint indicator</b>  Y - Yes N - No
QQCSTN	<b>Constraint name</b>
QQLTLN	<b>Library-long</b>
QQLTFN	<b>Table-long</b>
QQLPTL	<b>Physical library-long</b>
QQLPTF	<b>Table-long</b>
QQLILN	<b>Index library - long</b>
QQLIFN	<b>Index - long</b>
QQIDXK	<b>Key columns for the index advised</b>
QQC11	<b>Reserved</b>

Table 57. 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 (QAAQ3002) - Summary Row for Index Created

Table 58. 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 58. QQQ3002 - Summary Row for Index Created (continued)

Column Name	Description
QQJNTY	<b>Join type - when available</b>  IN - Inner join PO - Left partial outer join EX - Exception join
QQJNOP	<b>Join operator - when available</b>  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	<b>Number of advised key columns that use index scan-key positioning</b>
QQEPT	<b>Estimated processing time, in seconds</b>
QQKP	<b>Index scan-key positioning</b>  Y - Yes N - No
QQKPN	<b>Number of index scan-key positioning columns</b>
QQKS	<b>Index scan-key selection</b>  Y - Yes N - No
QQDSS	<b>Dataspace selection</b>  Y - Yes N - No
QQIDXA	<b>Index advised</b>  Y - Yes N - No
QQCST	<b>Constraint indicator</b>  Y - Yes N - No
QQCSTN	<b>Constraint name</b>
QQRCOD	<b>Reason code</b>  I1 - Row selection I2 - Ordering/Grouping I3 - Row selection and Ordering/Grouping I4 - Nested loop join I5 - Row selection using bitmap processing
QQTTIM	<b>Index create time</b>
QQTLN	<b>Library-long</b>
QQLTFN	<b>Table-long</b>

Table 58. 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 59. QAAA3003 - 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 59. QQQ3003 - Summary Row for Query Sort (continued)

Column Name	Description
<b>QQRCD</b>	<b>Reason code</b>
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 60. 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
QQTMPR	Number of rows in temporary

Table 60. QQQ3004 - Summary Row for Temporary Table (continued)

Column Name	Description
<b>QQRCD</b>	<b>Reason code</b>
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.
<b>QQDFVL</b>	<b>Default values may be present in temporary</b> Y - Yes N - No
<b>QQLTLN</b>	<b>Library-long</b>
<b>QQLTFN</b>	<b>Table-long</b>
<b>QQC11</b>	<b>Reserved</b>
<b>QQC12</b>	<b>Reserved</b>
<b>QQC21</b>	<b>Reserved</b>



Table 60. QQQ3004 - Summary Row for Temporary Table (continued)

Column Name	Description
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

## External table description (QAQQ3007) - Summary Row for Optimizer Information

Table 61. 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
QQC302	Reserved
QQ1000	Reserved

## External table description (QAQQ3008) - Summary Row for Subquery Processing

Table 62. 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 (QAQQ3010) - Summary Row for Host Variable and ODP Implementation

Table 63. 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
QQC301	Reserved
QQC302	Reserved



---

## Chapter 13. Query optimizer messages reference

See one of the following for query optimizer message reference:

- “Query optimization performance information messages”
- “Query optimization performance information messages and open data paths” on page 293
- “PRTSQLINF message reference” on page 300

---

### 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 &18 &19” on page 272
- “CPI4322 - Access path built from keyed file &1” on page 273
- “CPI4323 - The OS/400 query access plan has been rebuilt” on page 274
- “CPI4324 - Temporary file built for file &1” on page 276
- “CPI4325 - Temporary result file built for query” on page 277
- “CPI4326 - &12 &13 processed in join position &10” on page 277
- “CPI4327 - File &12 &13 processed in join position &10” on page 278
- “CPI4328 - Access path of file &3 was used by query” on page 279
- “CPI4329 - Arrival sequence access was used for &12 &13” on page 279
- “CPI432A - Query optimizer timed out for file &1” on page 280
- “CPI432B - Subselects processed as join query” on page 282
- “CPI432C - All access paths were considered for file &1” on page 282
- “CPI432D - Additional access path reason codes were used” on page 283
- “CPI432F - Access path suggestion for file &1” on page 284
- “CPI4330 - &6 tasks used for parallel &10 scan of file &1” on page 284
- “CPI4331 - &6 tasks used for parallel index created over file” on page 285
- “CPI4332 - &1 host variables used in query” on page 286
- “CPI4333 - Hashing algorithm used to process join” on page 287
- “CPI4334 - Query implemented as reusable ODP” on page 287
- “CPI4335 - Optimizer debug messages for hash join step &1 foil” on page 287
- “CPI4336 - Group processing generated” on page 288
- “CPI4337 - Temporary hash table build for hash join step &1” on page 288
- “CPI4338 - &1 Access path(s) used for bitmap processing of file &2” on page 288
- “CPI433D - Query options used to build the OS/400 query access plan” on page 289
- “CPI433F - Multiple join classes used to process join” on page 289
- “CPI4340 - Optimizer debug messages for join class step &1 foil” on page 289
- “CPI4341 - Performing distributed query” on page 290
- “CPI4342 - Performing distributed join for query” on page 290
- “CPI4343 - Optimizer debug messages for distributed query step &1 of &2 follow:” on page 290

- "CPI4345 - Temporary distributed result file &3 built for query" on page 290
- "CPI4346 - Optimizer debug messages for query join step &1 of &2 follow:" on page 291
- "CPI4347 - Query being processed in multiple steps" on page 291
- "CPI4348 - The ODP associated with the cursor was hard closed" on page 292
- "CPI4349 - Fast past refresh of the host variables values is not possible" on page 292
- "CPI434C - The OS/400 Query access plan was not rebuilt" on page 293

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

## CPI4321 - Access path built for &18 &19

CPI4321	
Message Text:	Access path built for &18 &19.
	<p>A temporary access path was built to access records from member &amp;6 of &amp;18 &amp;19 in library &amp;5 for reason code &amp;10. This process took &amp;11 minutes and &amp;12 seconds. The access path built contains &amp;15 entries. The access path was built using &amp;16 parallel tasks. A zero for the number of parallel tasks indicates that parallelism was not used. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. Perform specified ordering/grouping criteria.</li> <li>2. Perform specified join criteria.</li> <li>3. Perform specified record selection to minimize I/O wait time.</li> </ol> <p>The access path was built using the following key fields. The key fields and their corresponding sequence (ASCEND or DESCEND) will be shown:</p> <p>&amp;17. A key field of *MAP indicates the key field is an expression (derived field).</p> <p>The access path was built using sequence table &amp;13 in library &amp;14.</p> <p>A sequence table of *N indicates the access path was built without a sequence table. A sequence table of *I indicates the table was an internally derived table that is not available to the user.</p> <p>If &amp;18 &amp;19 in library &amp;5 is a logical file then the access path is built over member &amp;9 of physical file &amp;7 in library &amp;8.</p> <p>A file name starting with *QUERY or *N indicates the access path was built over a temporary file.</p>
Cause Text:	
	<p>If this query is run frequently, you may want to create an access path (index) similar to this definition for performance reasons. Create the access path using sequence table &amp;13 in library &amp;14, unless the sequence table is *N. If an access path is created, it is possible the query optimizer may still choose to create a temporary access path to process the query.</p> <p>If *MAP is returned for one of the key fields or *I is returned for the sequence table, then a permanent access path cannot be created. A permanent access path cannot be built with these specifications.</p>
Recovery Text:	

| 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**

CPI4322	
Message Text:	Access path built from keyed file &1.
	<p>A temporary access path was built using the access path from member &amp;3 of keyed file &amp;1 in library &amp;2 to access records from member &amp;6 of file &amp;4 in library &amp;5 for reason code &amp;10. This process took &amp;11 minutes and &amp;12 seconds. The access path built contains &amp;15 entries. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. Perform specified ordering/grouping criteria.</li> <li>2. Perform specified join criteria.</li> <li>3. Perform specified record selection to minimize I/O wait time</li> </ol> <p>The access path was built using the following key fields. The key fields and their corresponding sequence (ASCEND or DESCEND) will be shown:</p> <p>&amp;17.</p> <p>A key field of *MAP indicates the key field is an expression (derived field).</p> <p>The temporary access path was built using sequence table &amp;13 in library &amp;14.</p> <p>A sequence table of *N indicates the access path was built without a sequence table. A sequence table of *I indicates the table was an internally derived table that is not available to the user.</p> <p>If file &amp;4 in library &amp;5 is a logical file then the temporary access path is built over member &amp;9 of physical file &amp;7 in library &amp;8. Creating an access path from a keyed file generally results in improved performance.</p>
Cause Text:	

CPI4322	
Recovery Text:	<p>If this query is run frequently, you may want to create an access path (index) similar to this definition for performance reasons. Create the access path using sequence table &amp;13 in library &amp;14, unless the sequence table is *N. If an access path is created, it is possible the query optimizer may still choose to create a temporary access path to process the query.</p> <p>If *MAP is returned for one of the key fields or *I is returned for the sequence table, then a permanent access path cannot be created. A permanent access path cannot be built with these specifications.</p> <p>A temporary access path can only be created using index only access if all of the fields that were used by this temporary access path are also key fields for the access path from the keyed file.</p>

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**

CPI4323	
Message Text:	The OS/400 Query access plan has been rebuilt.

CPI4323	
Cause Text:	<p>The access plan was rebuilt for reason code &amp;13. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. A file or member is not the same object as the one referred to in the access plan. Some reasons include the object being re-created, restored, or overridden to a new object.</li> <li>2. Access plan was using a reusable Open Data Path (ODP), and the optimizer chose to use a non-reusable ODP.</li> <li>3. Access plan was using a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP</li> <li>4. The number of records in member &amp;3 of file &amp;1 in library &amp;2 has changed by more than 10%.</li> <li>5. A new access path exists over member &amp;6 of file &amp;4 in library &amp;5.</li> <li>6. An access path over member &amp;9 of file &amp;7 in library &amp;8 that was used for this access plan no longer exists or is no longer valid.</li> <li>7. OS/400 Query requires the access plan to be rebuilt because of system programming changes.</li> <li>8. The CCSID (Coded Character Set Identifier) of the current job is different than the CCSID used in the access plan</li> <li>9. The value of one of the following is different in the current job: date format, date separator, time format, or time separator.</li> <li>10. The sort sequence table specified has changed.</li> <li>11. The number of active processors or the size or paging option of the storage pool has changed.</li> <li>12. The system feature DB2 UDB Symmetric Multiprocessing has either been installed or removed.</li> <li>13. The value of the degree query attribute has changed either by the CHGSYSVAL or CHGQRYA CL commands or with the query options file &amp;15 in library &amp;16.</li> <li>14. A view is either being opened by a high level language open, or is being materialized.</li> <li>15. A sequence object or user-defined type or function is not the same object as the one referred to in the access plan; or, the SQL path used to generate the access plan is different than the current SQL path.</li> <li>16. Query attributes have been specified from the query options file &amp;15 in library &amp;16.</li> <li>17. The access plan was generated with a commitment control level that is different in the current job.</li> <li>18. The access plan was generated with a different static cursor answer set size.</li> <li>19. This is the first run of the query since a prepare or compile.</li> <li>20. Referential or check constraints for member &amp;19 of file &amp;17 in library &amp;18 have changed since the access plan was generated.</li> </ol> <p>If the reason code is 4, 5, 6, or 20 and the file specified in the reason code explanation is a logical file, then member &amp;12 of physical file &amp;10 in library &amp;11 is the file with the specified change.</p>
Recovery Text:	Excessive rebuilds should be avoided and may indicate an application design problem.

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

CPI4324	
Message Text:	Temporary file built for file &1.
Cause Text:	<p>A temporary file was built for member &amp;3 of file &amp;1 in library &amp;2 for reason code &amp;4. This process took &amp;5 minutes and &amp;6 seconds. The temporary file was required in order for the query to be processed. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. The file is a join logical file and its join-type (JDFTVAL) does not match the join-type specified in the query.</li> <li>2. The format specified for the logical file references more than one physical file.</li> <li>3. The file is a complex SQL view requiring a temporary file to contain the results of the SQL view.</li> <li>4. For an update-capable query, a subselect references a field in this file which matches one of the fields being updated.</li> <li>5. For an update-capable query, a subselect references SQL view &amp;1, which is based on the file being updated.</li> <li>6. For a delete-capable query, a subselect references either the file from which records are to be deleted or an SQL view or logical file based on the file from which records are to be deleted.</li> <li>7. The file is user-defined table function &amp;8 in &amp;2, and all the records were retrieved from the function. The processing time is not returned for this reason code.</li> <li>8. The file is a partition file requiring a temporary file for processing the grouping or join.</li> </ol>
Recovery Text:	You may want to change the query to refer to a file that does not require a temporary file to be built.

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, typically 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

CPI4325	
Message Text:	Temporary result file built for query.
	<p>A temporary result file was created to contain the results of the query for reason code &amp;4. This process took &amp;5 minutes and &amp;6 seconds. The temporary file created contains &amp;7 records. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. The query contains grouping fields (GROUP BY) from more than one file, or contains grouping fields from a secondary file of a join query that cannot be reordered.</li> <li>2. The query contains ordering fields (ORDER BY) from more than one file, or contains ordering fields from a secondary file of a join query that cannot be reordered.</li> <li>3. The grouping and ordering fields are not compatible.</li> <li>4. DISTINCT was specified for the query.</li> <li>5. Set operator (UNION, EXCEPT, or INTERSECT) was specified for the query.</li> <li>6. The query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 key fields specified for ordering.</li> <li>7. The query optimizer chose to use a sort rather than an access path to order the results of the query.</li> <li>8. Perform specified record selection to minimize I/O wait time.</li> <li>9. The query optimizer chose to use a hashing algorithm rather than an access path to perform the grouping for the query.</li> <li>10. The query contains a join condition that requires a temporary file.</li> <li>11. The query optimizer creates a run-time temporary file in order to implement certain correlated group by queries.</li> <li>12. The query contains grouping fields (GROUP BY, MIN/MAX, COUNT, and so on) and there is a read trigger on one or more of the underlying physical files in the query.</li> </ol>
Cause Text:	13. The query involves a static cursor or the SQL FETCH FIRST clause.
Recovery Text:	For more information about why a temporary result was used, refer to the Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13.

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 - &12 &13 processed in join position &10

CPI4326	
Message Text:	&12 &13 processed in join position &10.

CPI4326	
Cause Text:	<p>Access path for member &amp;5 of file &amp;3 in library &amp;4 was used to access records in member &amp;2 of file &amp;13 in library &amp;1 for reason code &amp;9. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. Perform specified record selection.</li> <li>2. Perform specified ordering/grouping criteria.</li> <li>3. Record selection and ordering/grouping criteria.</li> <li>4. Perform specified join criteria.</li> </ol> <p>If file &amp;13 in library &amp;1 is a logical file then member &amp;8 of physical file &amp;6 in library &amp;7 is the actual file in join position &amp;10.</p> <p>A file name starting with *TEMPX for the access path indicates it is a temporary access path built over file &amp;6.</p> <p>A file name starting with *N or *QUERY for the file indicates it is a temporary file.</p> <p>Index only access was used for this file within the query: &amp;11.</p> <p>A value of *YES for index only access processing indicates that all of the fields used from this file for this query can be found within the access path of file &amp;3. A value of *NO indicates that index only access could not be performed for this access path.</p> <p>Index only access is generally a performance advantage since all of the data can be extracted from the access path and the data space does not have to be paged into active memory.</p>
Recovery Text:	<p>Generally, to force a file to be processed in join position 1, specify an order by field from that file only.</p> <p>If ordering is desired, specifying ORDER BY fields over more than one file forces the creation of a temporary file and allows the optimizer to optimize the join order of all the files. No file is forced to be first.</p> <p>An access path can only be considered for index only access if all of the fields used within the query for this file are also key fields for that access path.</p> <p>Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for additional tips on optimizing a query's join order and index only access.</p>

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 &12 &13 processed in join position &10

CPI4327	
Message Text:	&12 &13 processed in join position &10.
Cause Text:	<p>Arrival sequence access was used to select records from member &amp;2 of file &amp;13 in library &amp;1.</p> <p>If file &amp;13 in library &amp;1 is a logical file then member &amp;8 of physical file &amp;6 in library &amp;7 is the actual file in join position &amp;10.</p> <p>A file name that starts with *QUERY for the file indicates it is a temporary file.</p>

CPI4327	
Recovery Text:	<p>Generally, to force a file to be processed in join position 1, specify an order by field from that file only.</p> <p>Refer to “Join optimization” on page 51 for additional tips on optimizing a query’s join order.</p>

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 about join position and join performance tips.

### CPI4328 - Access path of file &3 was used by query

CPI4328	
Message Text:	Access path of file &3 was used by query.
Cause Text:	<p>Access path for member &amp;5 of file &amp;3 in library &amp;4 was used to access records from member &amp;2 of &amp;12 &amp;13 in library &amp;1 for reason code &amp;9. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. Record selection.</li> <li>2. Ordering/grouping criteria.</li> <li>3. Record selection and ordering/grouping criteria.</li> </ol> <p>If file &amp;13 in library &amp;1 is a logical file then member &amp;8 of physical file &amp;6 in library &amp;7 is the actual file being accessed.</p> <p>Index only access was used for this query: &amp;11.</p> <p>A value of *YES for index only access processing indicates that all of the fields used for this query can be found within the access path of file &amp;3. A value of *NO indicates that index only access could not be performed for this access path.</p> <p>Index only access is generally a performance advantage since all of the data can be extracted from the access path and the data space does not have to be paged into active memory.</p>
Recovery Text:	<p>An access path can only be considered for index only access if all of the fields used within the query for this file are also key fields for that access path.</p> <p>Refer to Chapter 5, “Data access on DB2 UDB for iSeries: data access paths and methods,” on page 13 for additional tips on index only access.</p>

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 &12 &13

CPI4329	
Message Text:	Arrival sequence access was used for &12 &13.

CPI4329	
Cause Text:	<p>Arrival sequence access was used to select records from member &amp;2 of file &amp;13 in library &amp;1.</p> <p>If file &amp;13 in library &amp;1 is a logical file then member &amp;8 of physical file &amp;6 in library &amp;7 is the actual file from which records are being selected.</p> <p>A file name starting with *N or *QUERY for the file indicates it is a temporary file.</p>
Recovery Text:	<p>The use of an access path may improve the performance of the query if record selection is specified.</p> <p>If an access path does not exist, you may want to create one whose left-most key fields match fields in the record selection. Matching more key fields in the access path with fields in the record selection will result in improved performance.</p> <p>Generally, to force the use of an existing access path, specify order by fields that match the left-most key fields of that access path.</p> <p>For more information refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13.</p>

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

CPI432A	
Message Text:	Query optimizer timed out for file &1.

CPI432A	
Cause Text:	<p>The OS/400 Query optimizer timed out before it could consider all access paths built over member &amp;3 of file &amp;1 in library &amp;2.</p> <p>The list below shows the access paths considered before the optimizer timed out. If file &amp;1 in library &amp;2 is a logical file then the access paths specified are actually built over member &amp;9 of physical file &amp;7 in library &amp;8. Following each access path name in the list is a reason code which explains why the access path was not used. A reason code of 0 indicates that the access path was used to implement the query.</p> <p>The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. Access path was not in a valid state. The system invalidated the access path.</li> <li>2. Access path was not in a valid state. The user requested that the access path be rebuilt.</li> <li>3. Access path is a temporary access path (resides in library QTEMP) and was not specified as the file to be queried.</li> <li>4. The cost to use this access path, as determined by the optimizer, was higher than the cost associated with the chosen access method.</li> <li>5. The keys of the access path did not match the fields specified for the ordering/grouping criteria.</li> <li>6. The keys of the access path did not match the fields specified for the join criteria.</li> <li>7. Use of this access path would not minimize delays when reading records from the file as the user requested.</li> <li>8. The access path cannot be used for a secondary file of the join query because it contains static select/omit selection criteria. The join-type of the query does not allow the use of select/omit access paths for secondary files.</li> <li>9. File &amp;1 contains record ID selection. The join-type of the query forces a temporary access path to be built to process the record ID selection.</li> <li>10. and greater - View the second level message text of the next message issued (CPI432D) for an explanation of these reason codes.</li> </ol>
Recovery Text:	<p>To ensure an access path is considered for optimization specify that access path to be the queried file. The optimizer will first consider the access path of the file specified on the query. SQL-created indexes cannot be queried but can be deleted and recreated to increase the chance they will be considered during query optimization.</p> <p>The user may want to delete any access paths no longer needed.</p>

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

CPI432B	
Message Text:	Subselects processed as join query.
Cause Text:	Two or more SQL subselects were combined together by the query optimizer and processed as a join query. Processing subselects as a join query generally results in improved performance.
Recovery Text:	None — Generally, this method of processing is a good performing option.

## CPI432C - All access paths were considered for file &1

CPI432C	
Message Text:	All access paths were considered for file &1.
Cause Text:	<p>The OS/400 Query optimizer considered all access paths built over member &amp;3 of file &amp;1 in library &amp;2.</p> <p>The list below shows the access paths considered. If file &amp;1 in library &amp;2 is a logical file then the access paths specified are actually built over member &amp;9 of physical file &amp;7 in library &amp;8.</p> <p>Following each access path name in the list is a reason code which explains why the access path was not used. A reason code of 0 indicates that the access path was used to implement the query.</p> <p>The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. Access path was not in a valid state. The system invalidated the access path.</li> <li>2. Access path was not in a valid state. The user requested that the access path be rebuilt.</li> <li>3. Access path is a temporary access path (resides in library QTEMP) and was not specified as the file to be queried.</li> <li>4. The cost to use this access path, as determined by the optimizer, was higher than the cost associated with the chosen access method.</li> <li>5. The keys of the access path did not match the fields specified for the ordering/grouping criteria. For distributed file queries, the access path keys must exactly match the ordering fields if the access path is to be used when ALWCPYDTA(*YES or *NO) is specified.</li> <li>6. The keys of the access path did not match the fields specified for the join criteria.</li> <li>7. Use of this access path would not minimize delays when reading records from the file. The user requested to minimize delays when reading records from the file.</li> <li>8. The access path cannot be used for a secondary file of the join query because it contains static select/omit selection criteria. The join-type of the query does not allow the use of select/omit access paths for secondary files.</li> <li>9. File &amp;1 contains record ID selection. The join-type of the query forces a temporary access path to be built to process the record ID selection.</li> <li>10. and greater - View the second level message text of the next message issued (CPI432D) for an explanation of these reason codes.</li> </ol>
Recovery Text:	The user may want to delete any access paths no longer needed.

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

CPI432D	
Message Text:	Additional access path reason codes were used.
Cause Text:	<p>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 below rather than in those messages.</p> <ul style="list-style-type: none"> <li>• 10 - The user specified ignore decimal data errors on the query. This disallows the use of permanent access paths.</li> <li>• 11 - The access path contains static select/omit selection criteria which is not compatible with the selection in the query.</li> <li>• 12 - The access path contains static select/omit selection criteria whose compatibility with the selection in the query could not be determined. Either the select/omit criteria or the query selection became too complex during compatibility processing.</li> <li>• 13 - The access path contains one or more keys which may be changed by the query during an insert or update.</li> <li>• 14 - The access path is being deleted or is being created in an uncommitted unit of work in another process.</li> <li>• 15 - The keys of the access path matched the fields specified for the ordering/grouping criteria. However, the sequence table associated with the access path did not match the sequence table associated with the query.</li> <li>• 16 - The keys of the access path matched the fields specified for the join criteria. However, the sequence table associated with the access path did not match the sequence table associated with the query.</li> <li>• 17 - The left-most key of the access path did not match any fields specified for the selection criteria. Therefore, key row positioning could not be performed, making the cost to use this access path higher than the cost associated with the chosen access method.</li> <li>• 18 - The left-most key of the access path matched a field specified for the selection criteria. However, the sequence table associated with the access path did not match the sequence table associated with the query. Therefore, key row positioning could not be performed, making the cost to use this access path higher than the cost associated with the chosen access method.</li> <li>• 19 - The access path cannot be used because the secondary file of the join query is a select/omit logical file. The join-type requires that the select/omit access path associated with the secondary file be used or, if dynamic, that an access path be created by the system.</li> </ul>
Recovery Text:	See prior message CPI432A or CPI432C for more information.

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.



## CPI432F - Access path suggestion for file &1

CPI432F	
Message Text:	Access path suggestion for file &1.
	<p>To improve performance the query optimizer is suggesting a permanent access path be built with the key fields it is recommending. The access path will access records from member &amp;3 of file &amp;1 in library &amp;2.</p> <p>In the list of key fields that follow, the query optimizer is recommending the first &amp;10 key fields as primary key fields. The remaining key fields are considered secondary key fields and are listed in order of expected selectivity based on this query. Primary key fields are fields that significantly reduce the number of keys selected based on the corresponding selection predicate. Secondary key fields are fields that may or may not significantly reduce the number of keys selected. It is up to the user to determine the true selectivity of secondary key fields and to determine whether those key fields should be used when creating the access path.</p> <p>The query optimizer is able to perform key positioning over any combination of the primary key fields, plus one additional secondary key field. Therefore it is important that the first secondary key field be the most selective secondary key field. The query optimizer will use key selection with any remaining secondary key fields. While key selection is not as fast as key positioning it can still reduce the number of keys selected. Hence, secondary key fields that are fairly selective should be included. When building the access path all primary key fields should be specified first followed by the secondary key fields which are prioritized by selectivity. The following list contains the suggested primary and secondary key fields:</p>
Cause Text:	If file &1 in library &2 is a logical file then the access path should be built over member &9 of physical file &7 in library &8.
	If this query is run frequently, you may want to create the suggested access path for performance reasons. It is possible that the query optimizer will choose not to use the access path just created.
Recovery Text:	For more information, refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13.

You can also find more information at "Query optimizer index advisor" on page 86.

## CPI4330 - &6 tasks used for parallel &10 scan of file &1

CPI4330	
Message Text:	&6 tasks used for parallel &10 scan of file &1.

CPI4330	
Cause Text:	<p>&amp;6 is the average numbers of tasks used for a &amp;10 scan of member &amp;3 of file &amp;1 in library &amp;2.</p> <p>If file &amp;1 in library &amp;2 is a logical file, then member &amp;9 of physical file &amp;7 in library &amp;8 is the actual file from which records are being selected.</p> <p>A file name starting with *QUERY or *N for the file indicates a temporary result file is being used.</p> <p>The query optimizer has calculated that the optimal number of tasks is &amp;5 which was limited for reason code &amp;4. The reason code definitions are:</p> <ol style="list-style-type: none"> <li>1. The *NBRTASKS parameter value was specified for the DEGREE parameter of the CHGQRYA CL command.</li> <li>2. The optimizer calculated the number of tasks which would use all of the central processing units (CPU).</li> <li>3. The optimizer calculated the number of tasks which can efficiently run in this job's share of the memory pool.</li> <li>4. The optimizer calculated the number of tasks which can efficiently run using the entire memory pool</li> <li>5. The optimizer limited the number of tasks to equal the number of disk units which contain the file's data.</li> </ol> <p>The database manager may further limit the number of tasks used if the allocation of the file's data is not evenly distributed across disk units.</p>
Recovery Text:	<p>To disallow usage of parallel &amp;10 scan, specify *NONE on the query attribute degree.</p> <p>A larger number of tasks might further improve performance. The following actions based on the optimizer reason code might allow the optimizer to calculate a larger number:</p> <ol style="list-style-type: none"> <li>1. Specify a larger number of tasks value for the DEGREE parameter of the CHGQRYA CL command. Start with a value for number of tasks which is a slightly larger than &amp;5</li> <li>2. Simplify the query by reducing the number of fields being mapped to the result buffer or by removing expressions. Also, try specifying a number of tasks as described by reason code 1.</li> <li>3. Specify *MAX for the query attribute DEGREE.</li> <li>4. Increase the size of the memory pool.</li> <li>5. Use the CHGPF CL command or the SQL ALTER statement to redistribute the file's data across more disk units.</li> </ol> <p>For more information, refer to "Control parallel processing for queries" on page 111.</p>

## CPI4331 - &6 tasks used for parallel index created over file

CPI4331	
Message Text:	&6 tasks used for parallel index created over file &1.

CPI4331	
Cause Text:	<p>&amp;6 is the average numbers of tasks used for an index created over member &amp;3 of file &amp;1 in library &amp;2.</p> <p>If file &amp;1 in library &amp;2 is a logical file, then member &amp;9 of physical file &amp;7 in library &amp;8 is the actual file over which the index is being built.</p> <p>A file name starting with *QUERY or *N for the file indicates a temporary result file is being used.</p> <p>The query optimizer has calculated that the optimal number of tasks is &amp;5 which was limited for reason code &amp;4. The definition of reason codes are:</p> <ol style="list-style-type: none"> <li>1. The *NBRTASKS parameter value was specified for the DEGREE parameter of the CHGQRYA CL command.</li> <li>2. The optimizer calculated the number of tasks which would use all of the central processing units (CPU).</li> <li>3. The optimizer calculated the number of tasks which can efficiently run in this job's share of the memory pool.</li> <li>4. The optimizer calculated the number of tasks which can efficiently run using the entire memory pool.</li> </ol> <p>The database manager may further limit the number of tasks used for the parallel index build if either the allocation of the file's data is not evenly distributed across disk units or the system has too few disk units.</p>
Recovery Text:	<p>To disallow usage of parallel index build, specify *NONE on the query attribute degree.</p> <p>A larger number of tasks might further improve performance. The following actions based on the reason code might allow the optimizer to calculate a larger number:</p> <ol style="list-style-type: none"> <li>1. Specify a larger number of tasks value for the DEGREE parameter of the CHGQRYA CL command. Start with a value for number of tasks which is a slightly larger than &amp;5 to see if a performance improvement is achieved.</li> <li>2. Simplify the query by reducing the number of fields being mapped to the result buffer or by removing expressions. Also, try specifying a number of tasks for the DEGREE parameter of the CHGQRYA CL command as described by reason code 1.</li> <li>3. Specify *MAX for the query attribute degree.</li> <li>4. Increase the size of the memory pool.</li> </ol>

## CPI4332 - &1 host variables used in query

CPI4332	
Message Text:	&1 host variables used in query.
Cause Text:	<p>There were &amp;1 host variables defined for use in the query. The values used for the host variables for this open of the query follow: &amp;2.</p> <p>The host variables values displayed above may have been special values. An explanation of the special values follow:</p> <ul style="list-style-type: none"> <li>• DBCS data is displayed in hex format.</li> <li>• *N denotes a value of NULL.</li> <li>• *Z denotes a zero length string.</li> <li>• *L denotes a value too long to display in the replacement text.</li> <li>• *U denotes a value that could not be displayed.</li> </ul>
Recovery Text:	None

## CPI4333 - Hashing algorithm used to process join

CPI4333	
Message Text:	Hashing algorithm used to process join.
Cause Text:	<p>The hash join method is typically used for longer running join queries. The original query will be subdivided into hash join steps.</p> <p>Each hash join step will be optimized and processed separately. Debug messages which explain the implementation of each hash join step follow this message in the joblog.</p> <p>The list below shows the names of the files or the table functions used in this query. If the entry is for a file, the format of the entry in this list is the number of the hash join step, the filename as specified in the query, the member name as specified in the query, the filename actually used in the hash join step, and the member name actually used in the hash join step. If the entry is for a table function, the format of the entry in this list is the number of the hash join step and the function name as specified in the query.</p> <p>If there are two or more files or functions listed for the same hash step, then that hash step is implemented with nested loop join.</p>
Recovery Text:	<p>The hash join method is usually a good implementation choice, however, if you want to disallow the use of this method specify ALWCPYDTA(*YES).</p> <p>Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about hashing algorithm for join processing.</p>

## CPI4334 - Query implemented as reusable ODP

CPI4334	
Message Text:	Query implemented as reusable ODP.
Cause Text:	The query optimizer built the access plan for this query such that a reusable open data path (ODP) will be created. This plan will allow the query to be run repeatedly for this job without having to rebuild the ODP each time. This normally improves performance because the ODP is created only once for the job.
Recovery Text:	Generally, reusable ODPs perform better than non-reusable ODPs.

## CPI4335 - Optimizer debug messages for hash join step &1 foil

CPI4335	
Message Text:	Optimizer debug messages for hash join step &1 foil
Cause Text:	This join query is implemented using the hash join algorithm. The optimizer debug messages that follow provide the query optimization information about hash join step &1.
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about hashing algorithm for join processing.

## CPI4336 - Group processing generated

CPI4336	
Message Text:	Group processing generated.
Cause Text:	Group processing (GROUP BY) was added to the query step. Adding the group processing reduced the number of result records which should, in turn, improve the performance of subsequent steps.
Recovery Text:	For more information refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13

## CPI4337 - Temporary hash table build for hash join step &1

CPI4337	
Message Text:	Temporary hash table built for hash join step &1.
Cause Text:	A temporary hash table was created to contain the results of hash join step &1. This process took &2 minutes and &3 seconds. The temporary hash table created contains &4 records. The total size of the temporary hash table in units of 1024 bytes is &5. A list of the fields which define the hash keys follow:
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about hashing algorithm for join processing.

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

CPI4338	
Message Text:	&1 Access path(s) used for bitmap processing of file &2.
Cause Text:	<p>Bitmap processing was used to access records from member &amp;4 of file &amp;2 in library &amp;3.</p> <p>Bitmap processing is a method of allowing one or more access path(s) to be used to access the selected records from a file. Using bitmap processing, record selection is applied against each access path, similar to key row positioning, to create a bitmap. The bitmap has marked in it only the records of the file that are to be selected. If more than one access path is used, the resulting bitmaps are merged together using boolean logic. The resulting bitmap is then used to reduce access to just those records actually selected from the file.</p> <p>Bitmap processing is used in conjunction with the two primary access methods: arrival sequence (CPI4327 or CPI4329) or keyed access (CPI4326 or CPI4328). The message that describes the primary access method immediately precedes this message.</p> <p>When the bitmap is used with the keyed access method then it is used to further reduce the number of records selected by the primary access path before retrieving the selected records from the file.</p> <p>When the bitmap is used with arrival sequence then it allows the sequential scan of the file to skip records which are not selected by the bitmap. This is called skip sequential processing.</p> <p>The list below shows the names of the access paths used in the bitmap processing:</p> <p>If file &amp;2 in library &amp;3 is a logical file then member &amp;7 of physical file &amp;5 in library &amp;6 is the actual file being accessed.</p>

CPI4338	
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about bitmap processing.

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.

### CPI433D - Query options used to build the OS/400 query access plan

CPI433D	
Message Text:	Query options used to build the OS/400 query access plan.
Cause Text:	The access plan that was saved was created with query options retrieved from file &2 in library &1.
Recovery Text:	None

### CPI433F - Multiple join classes used to process join

CPI433F	
Message Text:	Multiple join classes used to process join.
Cause Text:	<p>Multiple join classes are used when join queries are written that have conflicting operations or cannot be implemented as a single query.</p> <p>Each join class step will be optimized and processed separately. Debug messages detailing the implementation of each join class follow this message in the joblog.</p> <p>The list below shows the file names of the files used in this query. The format of each entry in this list is the number of the join class step, the number of the join position in the join class step, the file name as specified in the query, the member name as specified in the query, the file name actually used in the join class step, and the member name actually used in the join class step.</p>
Recovery Text:	Refer to "Join optimization" on page 51 for more information about join classes.

### CPI4340 - Optimizer debug messages for join class step &1 foil

CPI4340	
Message Text:	Optimizer debug messages for join class step &1 follow:

CPI4340	
Cause Text:	This join query is implemented using multiple join classes. The optimizer debug messages that follow provide the query optimization information about join class step &1.
Recovery Text:	Refer to "Join optimization" on page 51 for more information about join classes.

### CPI4341 - Performing distributed query

CPI4341	
Message Text:	Performing distributed query.
Cause Text:	Query contains a distributed file. The query was processed in parallel on the following nodes: &1.
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming.

### CPI4342 - Performing distributed join for query

CPI4342	
Message Text:	Performing distributed join for query.
Cause Text:	<p>Query contains join criteria over a distributed file and a distributed join was performed, in parallel, on the following nodes: &amp;1.</p> <p>The library, file and member names of each file involved in the join follow: &amp;2.</p> <p>A file name beginning with *QQTDF indicates it is a temporary distributed result file created by the query optimizer and it will not contain an associated library or member name.</p>
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming.

### CPI4343 - Optimizer debug messages for distributed query step &1 of &2 follow:

CPI4343	
Message Text:	Optimizer debug messages for distributed query step &1 of &2 follow:
Cause Text:	A distributed file was specified in the query which caused the query to be processed in multiple steps. The optimizer debug messages that follow provide the query optimization information about distributed step &1 of &2 total steps.
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming.

### CPI4345 - Temporary distributed result file &3 built for query

CPI4345	
Message Text:	Temporary distributed result file &3 built for query.

CPI4345	
Cause Text:	<p>Temporary distributed result file &amp;3 was created to contain the intermediate results of the query for reason code &amp;6. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. Data from member &amp;2 of &amp;7 &amp;8 in library &amp;1 was directed to other nodes.</li> <li>2. Data from member &amp;2 of &amp;7 &amp;8 in library &amp;1 was broadcast to all nodes.</li> <li>3. Either the query contains grouping fields (GROUP BY) that do not match the partitioning keys of the distributed file or the query contains grouping criteria but no grouping fields were specified or the query contains a subquery.</li> <li>4. Query contains join criteria over a distributed file and the query was processed in multiple steps.</li> </ol> <p>A library and member name of *N indicates the data comes from a query temporary distributed file.</p> <p>File &amp;3 was built on nodes: &amp;9.</p> <p>It was built using partitioning keys: &amp;10.</p> <p>A partitioning key of *N indicates no partitioning keys were used when building the temporary distributed result file.</p>
Recovery Text:	<p>If the reason code is:</p> <ol style="list-style-type: none"> <li>1. Generally, a file is directed when the join fields do not match the partitioning keys of the distributed file. When a file is directed, the query is processed in multiple steps and processed in parallel. A temporary distributed result file is required to contain the intermediate results for each step.</li> <li>2. Generally, a file is broadcast when join fields do not match the partitioning keys of either file being joined or the join operator is not an equal operator. When a file is broadcast the query is processed in multiple steps and processed in parallel. A temporary distributed result file is required to contain the intermediate results for each step.</li> <li>3. Better performance may be achieved if grouping fields are specified that match the partitioning keys.</li> <li>4. Because the query is processed in multiple steps, a temporary distributed result file is required to contain the intermediate results for each step. See preceding message CPI4342 to determine which files were joined together.</li> </ol> <p>For more information about processing of distributed files, refer to the Distributed Database Programming,</p>

### CPI4346 - Optimizer debug messages for query join step &1 of &2 follow:

CPI4346	
Message Text:	Optimizer debug messages for query join step &1 of &2 follow:
Cause Text:	Query processed in multiple steps. The optimizer debug messages that follow provide the query optimization information about join step &1 of &2 total steps.
Recovery Text:	No recovery necessary.

### CPI4347 - Query being processed in multiple steps

CPI4347	
Message Text:	Query being processed in multiple steps.



CPI4347	
Cause Text	<p>The original query will be subdivided into multiple steps.</p> <p>Each step will be optimized and processed separately. Debug messages which explain the implementation of each step follow this message in the joblog.</p> <p>The list below shows the file names of the files used in this query. The format of each entry in this list is the number of the join step, the filename as specified in the query, the member name as specified in the query, the filename actually used in the step, and the member name actually used in the step.</p>
Recovery Text:	No recovery necessary.

### CPI4348 - The ODP associated with the cursor was hard closed

CPI4348	
Message Text:	The ODP associated with the cursor was hard closed.
Cause Text:	<p>The Open Data Path (ODP) for this statement or cursor has been hard closed for reason code &amp;1. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. Either the length of the new LIKE pattern is zero and the length of the old LIKE pattern is nonzero or the length of the new LIKE pattern is nonzero and the length of the old LIKE pattern is zero.</li> <li>2. An additional wildcard was specified in the LIKE pattern on this invocation of the cursor.</li> <li>3. SQL indicated to the query optimizer that the cursor cannot be refreshed.</li> <li>4. The system code could not obtain a lock on the file being queried.</li> <li>5. The length of the host variable value is too large for the the host variable as determined by the query optimizer.</li> <li>6. The size of the ODP to be refreshed is too large.</li> <li>7. Refresh of the local ODP of a distributed query failed.</li> <li>8. SQL hard closed the cursor prior to the fast path refresh code.</li> </ol>
Recovery Text:	In order for the cursor to be used in a reusable mode, the cursor cannot be hard closed. Look at the reason why the cursor was hard closed and take the appropriate actions to prevent a hard close from occurring.

### CPI4349 - Fast past refresh of the host variables values is not possible

CPI4349	
Message Text:	Fast past refresh of the host variable values is not possible.

CPI4349	
Cause Text:	<p>The Open Data Path (ODP) for this statement or cursor could not invoke the fast past refresh code for reason code &amp;1. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> <li>1. The new host variable value is not null and old host variable value is null or the new host variable value is zero length and the old host variable value is not zero length.</li> <li>2. The attributes of the new host variable value are not the same as the attributes of the old host variable value.</li> <li>3. The length of the host variable value is either too long or too short. The length difference cannot be handled in the fast path refresh code.</li> <li>4. The host variable has a data type of IGC ONLY and the length is not even or is less than 2 bytes.</li> <li>5. The host variable has a data type of IGC ONLY and the new host variable value does not contain an even number of bytes.</li> <li>6. A translate table with substitution characters was used.</li> <li>7. The host variable contains DBCS data and a CCSID translate table with substitution characters is required.</li> <li>8. The host variable contains DBCS that is not well formed. That is, a shift-in without a shift-out or visa versa.</li> <li>9. The host variable must be translated with a sort sequence table and the sort sequence table contains substitution characters.</li> <li>10. The host variable contains DBCS data and must be translated with a sort sequence table that contains substitution characters.</li> <li>11. The host variable is a Date, Time or Timestamp data type and the length of the host variable value is either too long or too short.</li> </ol>
Recovery Text:	Look at the reason why fast path refresh could not be used and take the appropriate actions so that fast path refresh can be used on the next invocation of this statement or cursor.

## CPI434C - The OS/400 Query access plan was not rebuilt

CPI434C	
Message Text:	The OS/400 Query access plan was not rebuilt.
Cause Text:	<p>The access plan for this query was not rebuilt. The optimizer determined that this access plan should be rebuilt for reason code &amp;13. However, the query attributes in the QAQQINI file disallowed the optimizer from rebuilding this access plan at this time.</p> <p>For a full explanation of the reason codes and their meanings, view the second level text of the message CPI4323.</p>
Recovery Text:	<p>Since the query attributes disallowed the query access plan from being rebuilt, the query will continue to be implemented with the existing access plan. This access plan may not contain all of the performance benefits that may have been derived from rebuilding the access plan.</p> <p>For more information about query attributes refer to "Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command" on page 96</p>

## 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"
- "SQL7911 - ODP reused" on page 295
- "SQL7912 - ODP created" on page 296
- "SQL7913 - ODP deleted" on page 296
- "SQL7914 - ODP not deleted" on page 296
- "SQL7915 - Access plan for SQL statement has been built" on page 297
- "SQL7916 - Blocking used for query" on page 297
- "SQL7917 - Access plan not updated" on page 297
- "SQL7918 - Reusable ODP deleted" on page 298
- "SQL7919 - Data conversion required on FETCH or embedded SELECT" on page 298
- "SQL7939 - Data conversion required on INSERT or UPDATE" on page 299

## SQL7910 - All SQL cursors closed

SQL7910	
Message Text:	SQL cursors closed.
Cause Text:	SQL cursors have been closed and all Open Data Paths (ODPs) have been deleted, except those that were opened by programs with the CLOSQLCSR(*ENDJOB) option or were opened by modules with the CLOSQLCSR(*ENDACTGRP) option. All SQL programs on the call stack have completed, and the SQL environment has been exited. This process includes the closing of cursors, the deletion of ODPs, the removal of prepared statements, and the release of locks.

SQL7910	
Recovery Text:	<p>To keep cursors, ODPs, prepared statements, and locks available after the completion of a program, use the CLOSQLCSR precompile parameter.</p> <ul style="list-style-type: none"> <li>• The *ENDJOB option will allow the user to keep the SQL resources active for the duration of the job</li> <li>• The *ENDSQL option will allow the user to keep SQL resources active across program calls, provided the SQL environment stays resident. Running an SQL statement in the first program of an application will keep the SQL environment active for the duration of that application.</li> <li>• The *ENDPGM option, which is the default for non-Integrated Language Environment® (ILE) programs, causes all SQL resources to only be accessible by the same invocation of a program. Once an *ENDPGM program has completed, if it is called again, the SQL resources are no longer active.</li> <li>• The *ENDMOD option causes all SQL resources to only be accessible by the same invocation of the module.</li> <li>• The *ENDACTGRP option, which is the default for ILE modules, will allow the user to keep the SQL resources active for the duration of the activation group.</li> </ul>

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

SQL7911	
Message Text:	ODP reused.
Cause Text:	An ODP that was previously created has been reused. There was a reusable Open Data Path (ODP) found for this SQL statement, and it has been used. The reusable ODP may have been from the same call to a program or a previous call to the program. A reuse of an ODP will not generate an OPEN entry in the journal.
Recovery Text:	None

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

SQL7912	
Message Text:	ODP created.
Cause Text:	An Open Data Path (ODP) has been created. No reusable ODP could be found. This occurs in the following cases: <ul style="list-style-type: none"><li>• This is the first time the statement has been run.</li><li>• A RCLRSC has been issued since the last run of this statement.</li><li>• The last run of the statement caused the ODP to be deleted.</li><li>• If this is an OPEN statement, the last CLOSE of this cursor caused the ODP to be deleted.</li><li>• The Application Server (AS) has been changed by a CONNECT statement.</li></ul>
Recovery Text:	If a cursor is being opened many times in an application, it is more efficient to use a reusable ODP, and not create an ODP every time. This also applies to repeated runs of INSERT, UPDATE, DELETE, and SELECT INTO statements. If ODPs are being created on every open, see the close message to determine why the ODP is being deleted.

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 “Retaining cursor positions for non-ILE program calls” on page 142 should be applied to this application.

## SQL7913 - ODP deleted

SQL7913	
Message Text:	ODP deleted.
Cause Text:	The Open Data Path (ODP) for this statement or cursor has been deleted. The ODP was not reusable. This could be caused by using a host variable in a LIKE clause, ordering on a host variable, or because the query optimizer chose to accomplish the query with an ODP that was not reusable.
Recovery Text:	See previous query optimizer messages to determine how the cursor was opened.

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 “Retaining cursor positions for non-ILE program calls” on page 142 should be applied to this application.

## SQL7914 - ODP not deleted

SQL7914	
Message Text:	ODP not deleted.
Cause Text:	The Open Data Path (ODP) for this statement or cursor has not been deleted. This ODP can be reused on a subsequent run of the statement. This will not generate an entry in the journal.
Recovery Text:	None

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

SQL7915	
Message Text:	Access plan for SQL statement has been built.
Cause Text:	SQL had to build the access plan for this statement at run time. This occurs in the following cases: <ul style="list-style-type: none"><li>• The program has been restored from a different release of OS/400, and this is the first time this statement has been run.</li><li>• All the files required for the statement did not exist at precompile time, and this is the first time this statement has been run.</li><li>• The program was precompiled using SQL naming mode, and the program owner has changed since the last time the program was called.</li></ul>
Recovery Text:	This is normal processing for SQL. Once the access plan is built, it will be used on subsequent runs of the statement.

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

SQL7916	
Message Text:	Blocking used for query.
Cause Text:	Blocking has been used in the implementation of this query. SQL will retrieve a block of records from the database manager on the first FETCH statement. Additional FETCH statements have to be issued by the calling program, but they do not require SQL to request more records, and therefore will run faster.
Recovery Text:	SQL attempts to utilize blocking whenever possible. In cases where the cursor is not update capable, and commitment control is not active, there is a possibility that blocking will be used.

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

SQL7917	
Message Text:	Access plan not updated.
Cause Text:	The query optimizer rebuilt the access plan for this statement, but the program could not be updated. Another job may be running the program. The program cannot be updated with the new access plan until a job can obtain an exclusive lock on the program. The exclusive lock cannot be obtained if another job is running the program, if the job does not have proper authority to the program, or if the program is currently being saved. The query will still run, but access plan rebuilds will continue to occur until the program is updated.
Recovery Text:	See previous messages from the query optimizer to determine why the access plan has been rebuilt. To ensure that the program gets updated with the new access plan, run the program when no other active jobs are using it.

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

SQL7918	
Message Text:	Reusable ODP deleted. Reason code &1.
	<p>An existing Open Data Path (ODP) was found for this statement, but it could not be reused for reason &amp;1. The statement now refers to different files or uses different override options than are in the ODP. Reason codes and their meanings are:</p> <ol style="list-style-type: none"> <li>1. Commitment control isolation level is not compatible.</li> <li>2. The statement contains SQL special register USER or CURRENT TIMEZONE, and the value for one of these registers has changed.</li> <li>3. The PATH used to locate an SQL function has changed.</li> <li>4. The job default CCSID has changed.</li> <li>5. The library list has changed, such that a file is found in a different library. This only affects statements with unqualified table names, when the table exists in multiple libraries.</li> <li>6. The file, library, or member for the original ODP was changed with an override.</li> <li>7. An OVRDBF or DLTOVR command has been issued. A file referred to in the statement now refers to a different file, library, or member.</li> <li>8. An OVRDBF or DLTOVR command has been issued, causing different override options, such as different SEQONLY or WAITRCD values.</li> <li>9. An error occurred when attempting to verify the statement override information is compatible with the reusable ODP information.</li> <li>10. The query optimizer has determined the ODP cannot be reused.</li> </ol>
Cause Text:	11. The client application requested not to reuse ODPs.
Recovery Text:	Do not change the library list, the override environment, or the values of the special registers if reusable ODPs are to be used.

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

SQL7919	
Message Text:	Data conversion required on FETCH or embedded SELECT.



SQL7919	
Cause Text:	<p>Host variable &amp;2 requires conversion. The data retrieved for the FETCH or embedded SELECT statement cannot be directly moved to the host variables. The statement ran correctly. Performance, however, would be improved if no data conversion was required. The host variable requires conversion for reason &amp;1</p> <ul style="list-style-type: none"> <li>• Reason 1 - host variable &amp;2 is a character or graphic string of a different length than the value being retrieved.</li> <li>• Reason 2 - host variable &amp;2 is a numeric type that is different than the type of the value being retrieved.</li> <li>• Reason 3 - host variable &amp;2 is a C character or C graphic string that is NUL-terminated, the program was compiled with option *CNULRQD specified, and the statement is a multiple-row FETCH.</li> <li>• Reason 4 - host variable &amp;2 is a variable length string and the value being retrieved is not.</li> <li>• Reason 5 - host variable &amp;2 is not a variable length string and the value being retrieved is.</li> <li>• Reason 6 - host variable &amp;2 is a variable length string whose maximum length is different than the maximum length of the variable length value being retrieved.</li> <li>• Reason 7 - a data conversion was required on the mapping of the value being retrieved to host variable &amp;2, such as a CCSID conversion</li> <li>• Reason 8 - a DRDA connection was used to get the value being retrieved into host variable &amp;2. The value being retrieved is either null capable or varying-length, is contained in a partial row, or is a derived expression.</li> <li>• Reason 10 - the length of host variable &amp;2 is too short to hold a TIME or TIMESTAMP value being retrieved.</li> <li>• Reason 11 - host variable &amp;2 is of type DATE, TIME or TIMESTAMP, and the value being retrieved is a character string.</li> <li>• Reason 12 - too many host variables were specified and records are blocked. Host variable &amp;2 does not have a corresponding column returned from the query.</li> <li>• Reason 13 - a DRDA connection was used for a blocked FETCH and the number of host variables specified in the INTO clause is less than the number of result values in the select list.</li> <li>• Reason 14 - a LOB Locator was used and the commitment control level of the process was not *ALL.</li> </ul>
Recovery Text:	To get better performance, attempt to use host variables of the same type and length as their corresponding result columns.

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

SQL7939	
Message Text:	Data conversion required on INSERT or UPDATE.



SQL7939	
Cause Text:	<p>The INSERT or UPDATE values cannot be directly moved to the columns because the data type or length of a value is different than one of the columns. The INSERT or UPDATE statement ran correctly. Performance, however, would be improved if no data conversion was required. The reason data conversion is required is &amp;1.</p> <ul style="list-style-type: none"> <li>• Reason 1 is that the INSERT or UPDATE value is a character or graphic string of a different length than column &amp;2.</li> <li>• Reason 2 is that the INSERT or UPDATE value is a numeric type that is different than the type of column &amp;2.</li> <li>• Reason 3 is that the INSERT or UPDATE value is a variable length string and column &amp;2 is not.</li> <li>• Reason 4 is that the INSERT or UPDATE value is not a variable length string and column &amp;2 is.</li> <li>• Reason 5 is that the INSERT or UPDATE value is a variable length string whose maximum length is different than the maximum length of column &amp;2.</li> <li>• Reason 6 is that a data conversion was required on the mapping of the INSERT or UPDATE value to column &amp;2, such as a CCSID conversion.</li> <li>• Reason 7 is that the INSERT or UPDATE value is a character string and column &amp;2 is of type DATE, TIME, or TIMESTAMP.</li> <li>• Reason 8 is that the target table of the INSERT is not a SQL table.</li> </ul>
Recovery Text:	To get better performance, try to use values of the same type and length as their corresponding columns.

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.

## **PRTSQLINF message reference**

- "SQL400A - Temporary distributed result file &1 was created to contain join result" on page 301
- "SQL400B - Temporary distributed result file &1 was created to contain join result" on page 302
- "SQL400C - Optimizer debug messages for distributed query step &1 and &2 follow" on page 302
- "SQL400D - GROUP BY processing generated" on page 302
- "SQL400E - Temporary distributed result file &1 was created while processing distributed subquery" on page 302
- "SQL4001 - Temporary result created" on page 303
- "SQL4002 - Reusable ODP sort used" on page 303
- "SQL4003 - UNION" on page 303
- "SQL4004 - SUBQUERY" on page 304
- "SQL4005 - Query optimizer timed out for table &1" on page 304
- "SQL4006 - All indexes considered for table &1" on page 304
- "SQL4007 - Query implementation for join position &1 table &2" on page 304
- "SQL4008 - Index &1 used for table &2" on page 305
- "SQL4009 - Index created for table &1" on page 305
- "SQL401A - Processing grouping criteria for query containing a distributed table" on page 305
- "SQL401B - Temporary distributed result table &1 was created while processing grouping criteria" on page 306
- "SQL401C - Performing distributed join for query" on page 306

- | • "SQL401D - Temporary distributed result table &1 was created because table &2 was directed" on page 306
- | • "SQL401E - Temporary distributed result table &1 was created because table &2 was broadcast" on page 307
- | • "SQL401F - Table &1 used in distributed join" on page 307
- | • "SQL4010 - Table scan access for table &1" on page 307
- | • "SQL4011 - Index scan-key row positioning used on table &1" on page 307
- | • "SQL4012 - Index created from index &1 for table &2" on page 308
- | • "SQL4013 - Access plan has not been built" on page 308
- | • "SQL4014 - &1 join column pair(s) are used for this join position" on page 308
- | • "SQL4015 - From-column &1.&2, to-column &3.&4, join operator &5, join predicate &6" on page 309
- | • "SQL4016 - Subselects processed as join query" on page 309
- | • "SQL4017 - Host variables implemented as reusable ODP" on page 309
- | • "SQL4018 - Host variables implemented as non-reusable ODP" on page 310
- | • "SQL4019 - Host variables implemented as file management row positioning reusable ODP" on page 310
- | • "SQL402A - Hashing algorithm used to process join" on page 310
- | • "SQL402B - Table &1 used in hash join step &2" on page 310
- | • "SQL402C - Temporary table created for hash join results" on page 311
- | • "SQL402D - Query attributes overridden from query options file &2 in library &1" on page 311
- | • "SQL4020 - Estimated query run time is &1 seconds" on page 311
- | • "SQL4021 - Access plan last saved on &1 at &2" on page 311
- | • "SQL4022 - Access plan was saved with SRVQRY attributes active" on page 312
- | • "SQL4023 - Parallel table prefetch used" on page 312
- | • "SQL4024 - Parallel index preload access method used" on page 312
- | • "SQL4025 - Parallel table preload access method used" on page 312
- | • "SQL4026 - Index only access used on table number &1" on page 313
- | • "SQL4027 - Access plan was saved with DB2 UDB Symmetric Multiprocessing installed on the system" on page 313
- | • "SQL4028 - The query contains a distributed table" on page 313
- | • "SQL4029 - Hashing algorithm used to process the grouping" on page 314
- | • "SQL4030 - &1 tasks specified for parallel scan on table &2." on page 314
- | • "SQL4031 - &1 tasks specified for parallel index create over table &2" on page 314
- | • "SQL4032 - Index &1 used for bitmap processing of table &2" on page 315
- | • "SQL4033 - &1 tasks specified for parallel bitmap create using &2" on page 315
- | • "SQL4034 - Multiple join classes used to process join" on page 315
- | • "SQL4035 - Table &1 used in join class &2" on page 316

## | **SQL400A - Temporary distributed result file &1 was created to contain join result**

SQL400A	
Message Text:	Temporary distributed result file &1 was created to contain join result. Result file was directed
Cause Text:	Query contains join criteria over a distributed file and a distributed join was performed in parallel. A temporary distributed result file was created to contain the results of the distributed join.

SQL400A	
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming information.

### SQL400B - Temporary distributed result file &1 was created to contain join result

SQL400B	
Message Text:	Temporary distributed result file &1 was created to contain join result. Result file was broadcast
Cause Text:	Query contains join criteria over a distributed file and a distributed join was performed in parallel. A temporary distributed result file was created to contain the results of the distributed join.
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming information.

### SQL400C - Optimizer debug messages for distributed query step &1 and &2 follow

SQL400C	
Message Text:	Optimizer debug messages for distributed query step &1 and &2 follow.
Cause Text:	A distributed file was specified in the query which caused the query to be processed in multiple steps. The optimizer debug messages that follow provide the query optimization information about the current step.
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming information.

### SQL400D - GROUP BY processing generated

SQL400D	
Message Text:	GROUP BY processing generated
Cause Text:	GROUP BY processing was added to the query step. Adding the GROUP BY reduced the number of result rows which should, in turn, improve the performance of subsequent steps.
Recovery Text:	For more information refer to the SQL Programming topic.

### SQL400E - Temporary distributed result file &1 was created while processing distributed subquery

SQL400E	
Message Text:	Temporary distributed result file &1 was created while processing distributed subquery
Cause Text:	A temporary distributed result file was created to contain the intermediate results of the query. The query contains a subquery which requires an intermediate result.

SQL400E	
Recovery Text:	<p>Generally, if the fields correlated between the query and subquery do not match the partition keys of the respective files, the query must be processed in multiple steps and a temporary distributed file will be built to contain the intermediate results.</p> <p>For more information about the processing of distributed files, refer to the Distributed Database Programming information.</p>

## SQL4001 - Temporary result created

SQL4001	
Message Text:	Temporary result created.
Cause Text:	<p>Conditions exist in the query which cause a temporary result to be created. One of the following reasons may be the cause for the temporary result:</p> <ul style="list-style-type: none"> <li>• The table is a join logical file and its join type (JDFTVAL) does not match the join-type specified in the query.</li> <li>• The format specified for the logical file refers to more than one physical table.</li> <li>• The table is a complex SQL view requiring a temporary table to contain the results of the SQL view.</li> <li>• The 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.</li> </ul>
Recovery Text:	Performance may be improved if the query can be changed to avoid temporary results.

## SQL4002 - Reusable ODP sort used

SQL4002	
Message Text:	Reusable ODP sort used
Cause Text:	<p>Conditions exist in the query which cause a sort to be used. This allowed the open data path (ODP) to be reusable. One of the following reasons may be the cause for the sort:</p> <ul style="list-style-type: none"> <li>• The 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.</li> <li>• The grouping and ordering columns are not compatible.</li> <li>• DISTINCT was specified for the query.</li> <li>• UNION was specified for the query.</li> <li>• The query had to be implemented using a sort. Key length of more than 2000 bytes, more than 120 ordering columns, or an ordering column containing a reference to an external user-defined function was specified for ordering.</li> <li>• The query optimizer chose to use a sort rather than an index to order the results of the query.</li> </ul>
Recovery Text:	A reusable ODP generally results in improved performance when compared to a non-reusable ODP.

## SQL4003 - UNION

SQL4003	
Message Text:	UNION

SQL4003	
Cause Text:	A UNION, EXCEPT, or INTERSECT operator was specified in the query. The messages preceding this keyword delimiter correspond to the subselect preceding the UNION, EXCEPT, or INTERSECT operator. The messages following this keyword delimiter correspond to the subselect following the UNION, EXCEPT, or INTERSECT operator.
Recovery Text:	None

## SQL4004 - SUBQUERY

SQL4004	
Message Text:	SUBQUERY
Cause Text:	The SQL statement contains a subquery. The messages preceding the SUBQUERY delimiter correspond to the subselect containing the subquery. The messages following the SUBQUERY delimiter correspond to the subquery.
Recovery Text:	None

## SQL4005 - Query optimizer timed out for table &1

SQL4005	
Message Text:	Query optimizer timed out for table &1
Cause Text:	The query optimizer timed out before it could consider all indexes built over the table. This is not an error condition. The query optimizer may time out in order to minimize optimization time. The query can be run in debug mode (STRDBG) to see the list of indexes which were considered during optimization. The table number refers to the relative position of this table in the query.
Recovery Text:	To ensure an index is considered for optimization, specify the logical file of the index as the table to be queried. The optimizer will first consider the index of the logical file specified on the SQL select statement. Note that SQL created indexes cannot be queried. An SQL index can be deleted and recreated to increase the chances it will be considered during query optimization. Consider deleting any indexes no longer needed.

## SQL4006 - All indexes considered for table &1

SQL4006	
Message Text:	All indexes considered for table &1
Cause Text:	The query optimizer considered all index built over the table when optimizing the query. The query can be run in debug mode (STRDBG) to see the list of indexes which were considered during optimization. The table number refers to the relative position of this table in the query.
Recovery Text:	None

## SQL4007 - Query implementation for join position &1 table &2

SQL4007	
Message Text:	Query implementation for join position &1 table &2

SQL4007	
Cause Text:	The join position identifies the order in which the tables are joined. A join position of 1 indicates this table is the first, or left-most, table in the join order. The table number refers to the relative position of this table in the query.
Recovery Text:	Join order can be influenced by adding an ORDER BY clause to the query. Refer to "Join optimization" on page 51 for more information about join optimization and tips to influence join order.

## SQL4008 - Index &1 used for table &2

SQL4008	
Message Text:	Index &1 used for table &2
Cause Text:	<p>The index was used to access rows from the table for one of the following reasons:</p> <ul style="list-style-type: none"> <li>• Row selection</li> <li>• Join criteria.</li> <li>• Ordering/grouping criteria.</li> <li>• Row selection and ordering/grouping criteria.</li> <li>• The table number refers to the relative position of this table in the query.</li> </ul> <p>The query can be run in debug mode (STRDBG) to determine the specific reason the index was used</p>
Recovery Text:	None

## SQL4009 - Index created for table &1

SQL4009	
Message Text:	Index created for table &1
Cause Text:	<p>A temporary index was built to access rows from the table for one of the following reasons:</p> <ul style="list-style-type: none"> <li>• Perform specified ordering/grouping criteria.</li> <li>• Perform specified join criteria.</li> </ul> <p>The table number refers to the relative position of this table in the query.</p>
Recovery Text:	<p>To improve performance, consider creating a permanent index if the query is run frequently. The query can be run in debug mode (STRDBG) to determine the specific reason the index was created and the key columns used when creating the index.</p> <p>NOTE: If permanent index is created, it is possible the query optimizer may still choose to create a temporary index to access the rows from the table.</p>

## SQL401A - Processing grouping criteria for query containing a distributed table

SQL401A	
Message Text:	Processing grouping criteria for query containing a distributed table

SQL401A	
Cause Text:	Grouping for queries that contain distributed tables can be implemented using either a one or two step method. If the one step method is used, the grouping columns (GROUP BY) match the partitioning keys of the distributed table. If the two step method is used, the grouping columns do not match the partitioning keys of the distributed table or the query contains grouping criteria but no grouping columns were specified. If the two step method is used, message SQL401B will appear followed by another SQL401A message.
Recovery Text:	For more information about processing of distributed tables, refer to the Distributed Database Programming information.

## SQL401B - Temporary distributed result table &1 was created while processing grouping criteria

SQL401B	
Message Text:	Temporary distributed result table &1 was created while processing grouping criteria
Cause Text:	A temporary distributed result table was created to contain the intermediate results of the query. Either the query contains grouping columns (GROUP BY) that do not match the partitioning keys of the distributed table or the query contains grouping criteria but no grouping columns were specified.
Recovery Text:	For more information about processing of distributed tables, refer to the Distributed Database Programming information.

## SQL401C - Performing distributed join for query

SQL401C	
Message Text:	Performing distributed join for query
Cause Text:	Query contains join criteria over a distributed table and a distributed join was performed in parallel. See the following SQL401F messages to determine which tables were joined together.
Recovery Text:	For more information about processing of distributed tables, refer to the Distributed Database Programming information.

## SQL401D - Temporary distributed result table &1 was created because table &2 was directed

SQL401D	
Message Text:	Temporary distributed result table &1 was created because table &2 was directed
Cause Text:	Temporary distributed result table was created to contain the intermediate results of the query. Data from a distributed table in the query was directed to other nodes.
Recovery Text:	Generally, a table is directed when the join columns do not match the partitioning keys of the distributed table. When a table is directed, the query is processed in multiple steps and processed in parallel. A temporary distributed result file is required to contain the intermediate results for each step.  For more information about processing of distributed tables, refer to the Distributed Database Programming information.



## SQL401E - Temporary distributed result table &1 was created because table &2 was broadcast

SQL401E	
Message Text:	Temporary distributed result table &1 was created because table &2 was broadcast
Cause Text:	Temporary distributed result table was created to contain the intermediate results of the query. Data from a distributed table in the query was broadcast to all nodes.
	Generally, a table is broadcast when join columns do not match the partitioning keys of either table being joined or the join operator is not an equal operator. When a table is broadcast the query is processed in multiple steps and processed in parallel. A temporary distributed result table is required to contain the intermediate results for each step.
Recovery Text:	For more information about processing of distributed tables, refer to the Distributed Database Programming information.

## SQL401F - Table &1 used in distributed join

SQL401F	
Message Text:	Table &1 used in distributed join
Cause Text:	Query contains join criteria over a distributed table and a distributed join was performed in parallel.
Recovery Text:	For more information about processing of distributed tables, refer to the Distributed Database Programming information.

## SQL4010 - Table scan access for table &1

SQL4010	
Message Text:	Table scan access for table &1
Cause Text:	Table scan access was used to select rows from the table. The table number refers to the relative position of this table in the query.
Recovery Text:	Table scan is generally a good performing option when selecting a high percentage of rows from the table. The use of an index, however, may improve the performance of the query when selecting a low percentage of rows from the table.

## SQL4011 - Index scan-key row positioning used on table &1

SQL4011	
Message Text:	Index scan-key row positioning used on table &1
	Index scan-key row positioning is defined as applying selection against the index to position directly to ranges of keys that match some or all of the selection criteria. Index scan-key row positioning only processes a subset of the keys in the index and is a good performing option when selecting a small percentage of rows from the table.
Cause Text:	The table number refers to the relative position of this table in the query.
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13for more information about index scan-key row positioning.



## SQL4012 - Index created from index &1 for table &2

SQL4012	
Message Text:	Index created from index &1 for table &2
Cause Text:	<p>A temporary index was created using the specified index to access rows from the queried table for one of the following reasons:</p> <ul style="list-style-type: none"> <li>• Perform specified ordering/grouping criteria.</li> <li>• Perform specified join criteria.</li> </ul> <p>The table number refers to the relative position of this table in the query.</p>
Recovery Text:	<p>Creating an index from an index is generally a good performing option. Consider creating a permanent index for frequently run queries. The query can be run in debug mode (STRDBG) to determine the key columns used when creating the index. NOTE: If a permanent index is created, it is possible the query optimizer may still choose to create a temporary index to access the rows from the table.</p>

## SQL4013 - Access plan has not been built

SQL4013	
Message Text:	Access plan has not been built
Cause Text:	<p>An access plan was not created for this query. Possible reasons may include:</p> <ul style="list-style-type: none"> <li>• Tables were not found when the program was created.</li> <li>• The query was complex and required a temporary result table.</li> <li>• Dynamic SQL was specified.</li> </ul>
Recovery Text:	If an access plan was not created, review the possible causes. Attempt to correct the problem if possible.

## SQL4014 - &1 join column pair(s) are used for this join position

SQL4014	
Message Text:	&1 join column pair(s) are used for this join position
Cause Text:	<p>The query optimizer may choose to process join predicates as either join selection or row selection. The join predicates used in join selection are determined by the final join order and the index used. This message indicates how many join column pairs were processed as join selection at this join position. Message SQL4015 provides detail on which columns comprise the join column pairs.</p> <p>If 0 join column pairs were specified then index scan-key row positioning with row selection was used instead of join selection.</p>
Recovery Text:	<p>If fewer join pairs are used at a join position than expected, it is possible no index exists which has keys matching the desired join columns. Try creating an index whose keys match the join predicates.</p> <p>If 0 join column pairs were specified then index scan-key row positioning was used. Index scan-key row positioning is normally a good performing option. Message SQL4011 provides more information about index scan-key row positioning.</p>

## SQL4015 - From-column &1.&2, to-column &3.&4, join operator &5, join predicate &6

SQL4015	
Message Text:	From-column &1.&2, to-column &3.&4, join operator &5, join predicate &6
Cause Text:	<p>Identifies which join predicate was implemented at the current join position. The replacement text parameters are:</p> <ul style="list-style-type: none"> <li>• &amp;1: The join 'from table' number. The table number refers to the relative position of this table in the query.</li> <li>• &amp;2: The join 'from column' name. The column within the join from table which comprises the left half of the join column pair. If the column name is *MAP, the column is an expression (derived field).</li> <li>• &amp;3: The join 'to table' number. The table number refers to the relative position of this table in the query.</li> <li>• &amp;4: The join 'to column' name. The column within the join to column which comprises the right half of the join column pair. If the column name is *MAP, the column is an expression (derived field).</li> <li>• &amp;5: The join operator. Possible values are EQ (equal), NE (not equal), GT (greater than), LT (less than), GE (greater than or equal), LE (less than or equal), and CP (cross join or cartesian product).</li> <li>• &amp;6: The join predicate number. Identifies the join predicate within this set of join pairs.</li> </ul>
Recovery Text:	Refer to "Join optimization" on page 51 for more information about joins.

## SQL4016 - Subselects processed as join query

SQL4016	
Message Text:	Subselects processed as join query
Cause Text:	The query optimizer chose to implement some or all of the subselects with a join query. Implementing subqueries with a join generally improves performance over implementing alternative methods.
Recovery Text:	None

## SQL4017 - Host variables implemented as reusable ODP

SQL4017	
Message Text:	Host variables implemented as reusable ODP
Cause Text:	The query optimizer has built the access plan allowing for the values of the host variables to be supplied when the query is opened. This query can be run with different values being provided for the host variables without requiring the access plan to be rebuilt. This is the normal method of handling host variables in access plans. The open data path (ODP) that will be created from this access plan will be a reusable ODP.
Recovery Text:	Generally, reusable open data paths perform better than non-reusable open data paths.

## SQL4018 - Host variables implemented as non-reusable ODP

SQL4018	
Message Text:	Host variables implemented as non-reusable ODP
Cause Text:	The query optimizer has implemented the host variables with a non-reusable open data path (ODP).
Recovery Text:	This can be a good performing option in special circumstances, but generally a reusable ODP gives the best performance.

## SQL4019 - Host variables implemented as file management row positioning reusable ODP

SQL4019	
Message Text:	Host variables implemented as file management row positioning reusable ODP
Cause Text:	The query optimizer has implemented the host variables with a reusable open data path (ODP) using file management row positioning.
Recovery Text:	Generally, a reusable ODP performs better than a non-reusable ODP.

## SQL402A - Hashing algorithm used to process join

SQL402A	
Message Text:	Hashing algorithm used to process join
Cause Text:	<p>The hash join algorithm is typically used for longer running join queries. The original query will be subdivided into hash join steps.</p> <p>Each hash join step will be optimized and processed separately. Access plan implementation information for each of the hash join steps is not available because access plans are not saved for the individual hash join dials. Debug messages detailing the implementation of each hash dial can be found in the joblog if the query is run in debug mode using the STRDBG CL command.</p>
Recovery Text:	<p>The hash join method is usually a good implementation choice, however, if you want to disallow the use of this method specify ALWCOPYDTA(*YES).</p> <p>Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about hashing algorithm for join processing.</p>

## SQL402B - Table &1 used in hash join step &2

SQL402B	
Message Text:	Table &1 used in hash join step &2
Cause Text:	<p>This message lists the table number used by the hash join steps. The table number refers to the relative position of this table in the query.</p> <p>If there are two or more of these messages for the same hash join step, then that step is a nested loop join.</p> <p>Access plan implementation information for each of the hash join step are not available because access plans are not saved for the individual hash steps. Debug messages detailing the implementation of each hash step can be found in the joblog if the query is run in debug mode using the STRDBG CL command.</p>

SQL402B	
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about hashing

## SQL402C - Temporary table created for hash join results

SQL402C	
Message Text:	Temporary table created for hash join results
Cause Text:	<p>The results of the hash join were written to a temporary table so that query processing could be completed. The temporary table was required because the query contained one or more of the following:</p> <ul style="list-style-type: none"> <li>• GROUP BY or summary functions</li> <li>• ORDER BY</li> <li>• DISTINCT</li> <li>• Expression containing columns from more than one table</li> <li>• Complex row selection involving columns from more than one table</li> </ul>
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about the hashing algorithm for join processing.

## SQL402D - Query attributes overridden from query options file &2 in library &1

SQL402D	
Message Text:	Query attributes overridden from query options file &2 in library &1
Cause Text:	None
Recovery Text:	None

## SQL4020 - Estimated query run time is &1 seconds

SQL4020	
Message Text:	Estimated query run time is &1 seconds
Cause Text:	The total estimated time, in seconds, of executing this query.
Recovery Text:	None

## SQL4021 - Access plan last saved on &1 at &2

SQL4021	
Message Text:	Access plan last saved on &1 at &2
Cause Text:	The date and time reflect the last time the access plan was successfully updated in the program object.
Recovery Text:	None

## SQL4022 - Access plan was saved with SRVQRY attributes active

SQL4022	
Message Text:	Access plan was saved with SRVQRY attributes active
Cause Text:	The access plan that was saved was created while SRVQRY was active. Attributes saved in the access plan may be the result of SRVQRY.
Recovery Text:	The query will be re-optimized the next time it is run so that SRVQRY attributes will not be permanently saved.

## SQL4023 - Parallel table prefetch used

SQL4023	
Message Text:	Parallel table prefetch used
Cause Text:	The query optimizer chose to use a parallel prefetch access method to reduce the processing time required for the table scan.
Recovery Text:	<p>Parallel prefetch can improve the performance of queries. Even though the access plan was created to use parallel prefetch, the system will actually run the query only if the following are true:</p> <ul style="list-style-type: none"><li>• The query attribute degree was specified with an option of *IO or *ANY for the application process.</li><li>• There is enough main storage available to cache the data being retrieved by multiple I/O streams. Normally, 5 megabytes would be a minimum. Increasing the size of the shared pool may improve performance.</li></ul> <p>For more information about parallel table prefetch, refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13</p>

## SQL4024 - Parallel index preload access method used

SQL4024	
Message Text:	Parallel index preload access method used
Cause Text:	The query optimizer chose to use a parallel index preload access method to reduce the processing time required for this query. This means that the indexes used by this query will be loaded into active memory when the query is opened.
Recovery Text:	<p>Parallel index preload can improve the performance of queries. Even though the access plan was created to use parallel preload, the system will actually use parallel preload only if the following are true:</p> <ul style="list-style-type: none"><li>• The query attribute degree was specified with an option of *IO or *ANY for the application process.</li><li>• There is enough main storage to load all of the index objects used by this query into active memory. Normally, a minimum of 5 megabytes would be a minimum. Increasing the size of the shared pool may improve performance.</li></ul> <p>For more information about parallel index preload, refer to the Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13.</p>

## SQL4025 - Parallel table preload access method used

SQL4025	
Message Text:	Parallel table preload access method used

SQL4025	
Cause Text:	The query optimizer chose to use a parallel table preload access method to reduce the processing time required for this query. This means that the data accessed by this query will be loaded into active memory when the query is opened.
Recovery Text:	<p>Parallel table preload can improve the performance of queries. Even though the access plan was created to use parallel preload, the system will actually use parallel preload only if the following are true:</p> <ul style="list-style-type: none"> <li>• The query attribute degree must have been specified with an option of *IO or *ANY for the application process.</li> <li>• There is enough main storage available to load all of the data in the file into active memory. Normally, 5 megabytes would be a minimum. Increasing the size of the shared pool may improve performance.</li> </ul> <p>For more information about parallel table preload, refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13.</p>

### SQL4026 - Index only access used on table number &1

SQL4026	
Message Text:	Index only access used on table number &1
Cause Text:	<p>Index only access is primarily used in conjunction with either index scan-key row positioning index scan-key selection. This access method will extract all of the data from the index rather than performing random I/O to the data space.</p> <p>The table number refers to the relative position of this table in the query.</p>
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about index only access.

### SQL4027 - Access plan was saved with DB2 UDB Symmetric Multiprocessing installed on the system

SQL4027	
Message Text:	Access plan was saved with DB2 UDB Symmetric Multiprocessing installed on the system
Cause Text:	<p>Text: The access plan saved was created while the system feature DB2 UDB Symmetric Multiprocessing was installed on the system. The access plan may have been influenced by the presence of this system feature.</p> <p>Having this system feature installed may cause the implementation of the query to change.</p>
Recovery Text:	For more information about how the system feature DB2 UDB Symmetric Multiprocessing can influence a query, refer to "Control parallel processing for queries" on page 111.

### SQL4028 - The query contains a distributed table

SQL4028	
Message Text:	The query contains a distributed table

SQL4028	
	A distributed table was specified in the query which may cause the query to be processed in multiple steps. If the query is processed in multiple steps, additional messages will detail the implementation for each step. Access plan implementation information for each step is not available because access plans are not saved for the individual steps.
Cause Text:	Debug messages detailing the implementation of each step can be found in the joblog if the query is run in debug mode using the STRDBG CL command.
Recovery Text:	For more information about how a distributed table can influence the query implementation refer to the Distributed Database Programming information.

## SQL4029 - Hashing algorithm used to process the grouping

SQL4029	
Message Text:	Hashing algorithm used to process the grouping
Cause Text:	The grouping specified within the query was implemented with a hashing algorithm.
	Implementing the grouping with the hashing algorithm is generally a performance advantage since an index does not have to be created. However, if you want to disallow the use of this method simply specify ALWCPYDTA(*YES).
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about the hashing algorithm.

## SQL4030 - &1 tasks specified for parallel scan on table &2.

SQL4030	
Message Text:	&1 tasks specified for parallel scan on table &2.
	The query optimizer has calculated the optimal number of tasks for this query based on the query attribute degree.
Cause Text:	The table number refers to the relative position of this table in the query.
	Parallel table or index scan can improve the performance of queries. Even though the access plan was created to use the specified number of tasks for the parallel scan, the system may alter that number based on the availability of the pool in which this job is running or the allocation of the table's data across the disk units.
Recovery Text:	Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about parallel scan.

## SQL4031 - &1 tasks specified for parallel index create over table &2

SQL4031	
Message Text:	&1 tasks specified for parallel index create over table &2
	The query optimizer has calculated the optimal number of tasks for this query based on the query attribute degree.
Cause Text:	The table number refers to the relative position of this table in the query.

SQL4031	
Recovery Text:	<p>Parallel index create can improve the performance of queries. Even though the access plan was created to use the specified number of tasks for the parallel index build, the system may alter that number based on the availability of the pool in which this job is running or the allocation of the table's data across the disk units.</p> <p>Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about parallel index create.</p>

## SQL4032 - Index &1 used for bitmap processing of table &2

SQL4032	
Message Text:	Index &1 used for bitmap processing of table &2
Cause Text:	<p>The index was used, in conjunction with query selection, to create a bitmap. The bitmap, in turn, was used to access rows from the table.</p> <p>This message may appear more than once per table. If this occurs, then a bitmap was created from each index of each message. The bitmaps were then combined into one bitmap using boolean logic and the resulting bitmap was used to access rows from the table.</p> <p>The table number refers to the relative position of this table in the query.</p>
Recovery Text:	<p>The query can be run in debug mode (STRDBG) to determine more specific information.</p> <p>Also, refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about bitmap processing.</p>

## SQL4033 - &1 tasks specified for parallel bitmap create using &2

SQL4033	
Message Text:	&1 tasks specified for parallel bitmap create using &2
Cause Text:	The query optimizer has calculated the optimal number of tasks to use to create the bitmap based on the query attribute degree.
Recovery Text:	<p>Using parallel index scan to create the bitmap can improve the performance of queries. Even though the access plan was created to use the specified number of tasks, the system may alter that number based on the availability of the pool in which this job is running or the allocation of the file's data across the disk units.</p> <p>Refer to Chapter 5, "Data access on DB2 UDB for iSeries: data access paths and methods," on page 13 for more information about parallel scan.</p>

## SQL4034 - Multiple join classes used to process join

SQL4034	
Message Text:	Multiple join classes used to process join



**SQL4034**

	<p>Multiple join classes are used when join queries are written that have conflicting operations or cannot be implemented as a single query.</p> <p>Each join class will be optimized and processed as a separate step of the query with the results written out to a temporary table.</p> <p>Access plan implementation information for each of the join classes is not available because access plans are not saved for the individual join class dials. Debug messages detailing the implementation of each join dial can be found in the joblog if the query is run in debug mode using the STRDBG CL command.</p>
Cause Text:	
Recovery Text:	Refer to "Join optimization" on page 51 for more information about join classes.

**SQL4035 - Table &1 used in join class &2**

**SQL4035**

Message Text:	Table &1 used in join class &2
	<p>This message lists the table numbers used by each of the join classes. The table number refers to the relative position of this table in the query.</p> <p>All of the tables listed for the same join class will be processed during the same step of the query. The results from all of the join classes will then be joined together to return the final results for the query.</p> <p>Access plan implementation information for each of the join classes are not available because access plans are not saved for the individual classes. Debug messages detailing the implementation of each join class can be found in the joblog if the query is run in debug mode using the STRDBG CL command.</p>
Cause Text:	
Recovery Text:	Refer to "Join optimization" on page 51 for more information about join classes.

---

## Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106-0032, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Software Interoperability Coordinator, Department 49XA  
3605 Highway 52 N  
Rochester, MN 55901  
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

- | The licensed program described in this information and all licensed material available for it are provided
- | by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement,
- | IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

- | SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS
- | PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER
- | EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR
- | CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND
- | NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

| UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR  
| ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

- | 1. LOSS OF, OR DAMAGE TO, DATA;
- | 2. SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL  
| DAMAGES; OR
- | 3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

| SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR  
| CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS  
| MAY NOT APPLY TO YOU.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

©IBM, August 2005. Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2005. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Programming Interface Information

| This information documents intended Programming Interfaces that allow the customer to write programs  
| to obtain the services of DB2 Universal Database for iSeries Database performance and query  
| optimization.

---

## Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- | DB2
- | DB2 Universal Database
- | DRDA
- | i5/OS
- | IBM
- | iSeries
- | Language Environment
- | Net.Data
- | Operating System/400
- | OS/400

Other company, product, and service names may be trademarks or service marks of others.

---

## Terms and conditions for downloading and printing information

| Permissions for the use of the information you have selected for download are granted subject to the  
| following terms and conditions and your indication of acceptance thereof.

| **Personal Use:** You may reproduce this information for your personal, noncommercial use provided that  
| all proprietary notices are preserved. You may not distribute, display or make derivative works of this  
| information, or any portion thereof, without the express consent of IBM.

| **Commercial Use:** You may reproduce, distribute and display this information solely within your  
| enterprise provided that all proprietary notices are preserved. You may not make derivative works of this  
| information, or reproduce, distribute or display this information or any portion thereof outside your  
| enterprise, without the express consent of IBM.

| Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either  
| express or implied, to the information or any data, software or other intellectual property contained  
| therein.

| IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of  
| the information is detrimental to its interest or, as determined by IBM, the above instructions are not  
| being properly followed.

| You may not download, export or re-export this information except in full compliance with all applicable  
| laws and regulations, including all United States export laws and regulations. IBM MAKES NO  
| GUARANTEE ABOUT THE CONTENT OF THIS INFORMATION. THE INFORMATION IS PROVIDED  
| "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING  
| BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT,  
| AND FITNESS FOR A PARTICULAR PURPOSE.

All material copyrighted by IBM Corporation.

| By downloading or printing information from this site, you have indicated your agreement with these  
| terms and conditions.

---

# Index

## A

- access method
  - bitmap probe 39
  - bitmap scan 37
  - buffer scan 44
  - create table 13
  - encoded vector index 21
  - encoded vector index probe 22
  - hash table 23
  - hash table probe 25
  - hash table scan 24
  - list scan 31
  - permanent objects 13
  - radix index 16
  - radix index probe 18
  - radix index scan 17
  - row number list probe 35
  - row number list scan 33
  - sorted list probe 28
  - sorted list scan 27
  - table probe 15
  - table scan 14
  - temporary bitmap 37
  - temporary buffer 44
  - temporary index 40
  - temporary index probe 43
  - temporary index scan 41
  - temporary list scan 31
  - temporary objects 23
  - temporary row number list 33
  - temporary sorted list 27
- access plan
  - validation 50
- access plan rebuilt
  - summary row 210
- advisor
  - query optimizer index 86
- ALLOCATE clause
  - performance implications 153
- allow copy data (ALWCPYDTA)
  - parameter 139
- ALWCPYDTA (allow copy data)
  - parameter 139
- APIs
  - statistics manager 118

## B

- bitmap
  - access method 37
- bitmap created
  - summary row 232
- bitmap merge
  - summary row 235
- bitmap probe
  - access method 39
- bitmap scan
  - access method 37
- blocking consideration
  - using, affect on performance 147

- blocking, SQL
  - improving performance 147
- buffer
  - scan access method 44

## C

- calls, number
  - using
    - FETCH statement 146
- canceling a query 109
- change query attributes 96
- Change Query Attributes (CHGQRYA)
  - command 76
- changing
  - query options file 100
- CHGQRYA (Change Query Attributes)
  - command 76
- CLOSQLCSR parameter
  - using 143
- command
  - CHGQRYA 96
  - CHGQRYA command 96
  - QAQQINI 97
  - QAQQINI command 97
- command (CL)
  - Change Query Attributes (CHGQRYA) 76
  - CHGQRYA (Change Query Attributes) 76
  - Delete Override (DLTOVR) 141
  - Display Job (DSPJOB) 75
  - Display Journal (DSPJRN) 142
  - DLTOVR (Delete Override) 141
  - DSPJOB (Display Job) 75
  - DSPJRN (Display Journal) 142
  - Override Database File (OVRDBF) 147
  - OVRDBF (Override Database File) 147
  - Print SQL Information (PRTSQLINF) 76, 109
  - QAQQINI 100
  - Start Database Monitor (STRDBMON) 76
  - STRDBMON (Start Database Monitor) 76
  - Trace Job (TRCJOB) 142
  - TRCJOB (Trace Job) 142
- commands
  - End Database Monitor (ENDDDBMON) 79
  - Start Database Monitor (STRDBMON) 79
- commitment control
  - displaying 75
- controlling parallel processing 111
- copy of the data
  - using to improve performance 152
- CQE engine 7

- create table
  - access method 13
- cursor
  - positions
    - retaining across program call 142, 143
    - rules for retaining 142
    - using to improve performance 142, 143

## D

- data
  - paging
    - interactively displayed to improve performance 149
    - selecting from multiple tables
      - affect on performance 57
- data path, open 293
- database monitor
  - end 79
  - examples 87, 90
  - logical file DDS 163
  - physical file DDS 155
  - start 79
- database monitor performance rows 80
- database query performance
  - monitoring 78
- DDS
  - database monitor logical file 163
  - database monitor physical file 155
- definitions
  - binary radix index 121
  - bitmap probe access method 39
  - bitmap scan access method 37
  - buffer scan access method 44
  - CQE 7
  - dial 52
  - encoded vector index 122
  - encoded vector index access
    - method 21
  - encoded vector index probe access
    - method 22
  - hash table probe access method 25
  - hash table scan access method 24
  - isolatable 57
  - list scan access method 31
  - miniplan 50
  - open data path 293
  - plan cache 11
  - primary table 52
  - query dispatcher 9
  - radix index 16
  - radix index probe 18
  - radix index scan access method 17
  - row number list probe access
    - method 35
  - row number list scan access
    - method 33
  - secondary tables 52
  - sorted list probe access method 28

- definitions (*continued*)
  - sorted list scan access method 27
  - SQE 7
  - statistics manager 10, 113
  - symmetrical multiprocessing 47
  - table create 13
  - table probe 15
  - table scan 14
  - temporary bitmap access method 37
  - temporary buffer access method 44
  - temporary hash table 23
  - temporary index access method 40
  - temporary index probe access method 43
  - temporary index scan access method 41
  - temporary list access method 31
  - temporary objects 23
  - temporary row number list access method 33
  - temporary sorted list access method 27
  - Visual Explain 94
- Delete Override (DLTOVR)
  - command 141
- detail row
  - rows retrieved 230
- Display Job (DSPJOB) command 75
- Display Journal (DSPJRN)
  - command 142
- distinct processing
  - summary row 241
- DSPJOB (Display Job) command 75

**E**

- encoded vector index
  - access method 21
  - creating 122
  - maintenance 123
  - recommendations 124
- encoded vector index probe
  - access method 22
- End Database Monitor (ENDDDBMON)
  - command 79
- ENDDDBMON (end database monitor)
  - command 79
- examples
  - database monitor 87, 90
  - governor 110
  - indexes 132
  - performance analysis 87, 88, 89
  - reducing the number of open database operation 140
  - selecting data from multiple tables 57

**F**

- file
  - query options 100

**G**

- generic query information
  - summary row 222, 227

- governor 107
  - \*DFT 109
  - \*RQD 109
  - \*SYSRPLY 109
  - CHGQRYA 107
  - JOB 108
  - QRYTIMLMT 107
  - time limit 108
- grouping
  - summary row 248, 251
- grouping optimization 61

## H

- hash join 53
- hash table
  - access method 23
  - probe access method 25
  - scan access method 24
  - summary row 239
- host variable and ODP implementation
  - summary row 217

## I

- improving performance 139, 152
  - blocking, using 147
  - join queries 59
  - paging interactively displayed data 149
  - PREPARE statement 149
  - retaining cursor positions across program call 142, 143
  - SELECT statements, using effectively 148
  - selecting data from multiple tables 57
  - SQL blocking 147
  - using
    - close SQL cursor (CLOSQLCSR) 142, 143
    - FETCH FOR n ROWS 146
    - INSERT n ROWS 147
    - precompile options 151

- index
  - access method 16
  - and the optimizer 125
  - basics 121
  - binary radix index 121
  - coding for effective indexes 129
  - comparing radix and encoded vector indexes 125
  - creating a strategy 121
  - encoded vector index 122
  - encoded vector index access method 21
  - encoded vector index maintenance 123
  - encoded vector index probe access method 22
  - examples 132
  - index probe access method 18
  - maintenance 121
  - recommendations for encoded vector indexes 124
  - scan access method 17

- index (*continued*)
  - statistics versus indexes 115
  - strategy 127
  - temporary
    - probe access method 43
    - scan access method 41
    - temporary access method 40
    - using with sort sequence 131
- index advisor
  - query optimizer 86
- index created
  - summary row 194
- information messages
  - open data path 293, 300
  - performance 271
- INSERT n ROWS
  - improving performance 147
- interactively displayed data, paging
  - affect on performance 149

## J

- JOB 108
- join
  - hash 53
  - optimization 51
- join optimization
  - performance tips 59
- join order
  - optimization 55
- join position 277
- join secondary dials
  - costing 55

## L

- limit, time 108
- list scan
  - access method 31
- live data
  - using to improve performance 139
- locks
  - analyzing 75
- logical file DDS
  - database monitor 163
- long object names
  - performance 151
- look ahead predicate generation 57

## M

- message
  - cause and user response 76
  - debug mode 76
  - open data path information 293, 300
  - performance information 271
  - PRTSQLINF 77
    - viewing with iSeries Navigator 77
  - viewing with iSeries Navigator 76
- monitor (ENDDDBMON) command, end database 79
- monitoring
  - database query performance 78



- multiple
  - table
    - improving performance when selecting data from 57

## N

- nested loop join 52
- number of calls
  - using a FETCH statement 146
- number of open database operations
  - improving performance by reducing 140

## O

- ODP implementation and host variable
  - summary row 217
- open
  - closing 141
  - determining number 142
  - effect on performance 140
  - reducing number 140
- open data path
  - definition 293
  - information messages 293
- OPNQRYF (Open Query File)
  - command 75
- optimization 49
  - grouping 61
  - join 51
  - join order 55
  - nested loop join 52
- optimizer
  - operation 49
  - query index advisor 86
- optimizer timed out
  - summary row 213
- options, precompile
  - improving performance by using 151
- output
  - all queries that performed table scans 88
  - SQL queries that performed table scans 88
- Override Database File (OVRDBF)
  - command 147

## P

- paging
  - interactively displayed data 149
- parallel processing
  - controlling
    - in jobs (CHGQRYA command) 112
    - system wide (QQRVDEGREE) value 111
- parameters, command
  - ALWCPYDTA (allow copy data) 139, 152
  - CLOSQPCSR (close SQL cursor) 142, 143
- path, open data 293
- performance 49
  - information messages 271

- performance (*continued*)
  - monitoring 75
  - monitoring query 78
  - open data path messages 293, 300
  - OPNQRYF 75
  - optimizing 75
  - tools 75
  - using long object names 151
- performance analysis
  - example 1 87
  - example 2 88
  - example 3 89
- performance considerations 109, 140
- performance improvement
  - blocking, using 147
  - paging interactively displayed data 149
  - PREPARE statement 149
  - reducing number of open database operation 140
  - retaining cursor positions across program call 142, 143
  - SELECT statements, using effectively 148
  - selecting data from multiple tables 57
  - SQL blocking 147
  - using copy of the data 152
  - using INSERT n ROWS 147
  - using live data 139
  - using precompile options 151
- performance rows
  - database monitor 80
- permanent objects
  - access method 13
- physical file DDS
  - database monitor 155
- plan cache 11
- precompile options
  - improving performance, using 151
- precompiler command
  - default 142, 143
- precompiler parameter
  - ALWCPYDTA 139
  - CLOSQPCSR 143
- predicate
  - transitive closure 56
- Predictive Query Governor 107
- PREPARE statement
  - improving performance 149
- Print SQL Information (PRTSQPCINF) 76, 109
- problems
  - join query performance 59
- program calls
  - rules for retaining cursor positions 143

## Q

- QAQQINI 100
- QDT 50
- QRYTIMLMT parameter
  - CHGQRYA (Change Query Attributes) command 76
- query
  - canceling 109

- Query Definition Template (QDT) 50
- query dispatcher 9
- query engine overview 7
- query optimizer 49
  - decision-making rules 50
- query optimizer index advisor 86
  - in Visual Explain 95
- query options
  - file 100
- query options file 97
  - changing 100
- query performance
  - monitoring 78
- query sort
  - summary row 199
- query time limit 108

## R

- radix index
  - access method 16
- radix index probe
  - access method 18
- radix index scan
  - access method 17
- reducing number of open database operations
  - improving performance, example 140
- Reorganize Physical File Member (RGZPFM) command
  - effect on variable-length columns 154
- resource
  - optimization 49
- retaining cursor positions
  - across program call
    - improving performance 142, 143
  - all program calls
    - rules 143
- row number list probe
  - access method 35
- row number list scan
  - access method 33
- rows
  - database monitor performance 80
- rows retrieved
  - detail row 230
- ROWS, INSERT n
  - improving performance 147
- rule
  - retaining cursor positions
    - program calls 143

## S

- SELECT statement
  - using effectively to improve performance 148
- selecting
  - data from multiple tables 57
- setting query time limit 110
- sort sequence
  - using indexes 131
- sorted list probe
  - access method 28
- sorted list scan
  - access method 27



- SQE engine 7
- SQL blocking
  - improving performance 147
- SQL information
  - summary row 168, 253
- Start Database Monitor (STRDBMON)
  - command 76, 79
- statements
  - FETCH
    - FOR n ROWS 146
    - number of calls 146
  - INSERT
    - n ROWS 147
  - PREPARE
    - improving performance 149
- statistics manager 10, 113
  - APIs 118
  - automatic statistics collection 114
  - automatic statistics refresh 114
  - determining existence 116
  - manually collecting and refreshing statistics 117
  - monitoring background statistics collection 116
  - replication of column statistics 116
  - statistics versus indexes 115
  - viewing statistics requests 115
- STRDBMON (Start Database Monitor)
  - command 76, 79
- STRDBMON/ENDDDBMON commands
  - summary row 229
- subquery merge
  - summary row 244
- subquery processing
  - summary row 216
- summary row
  - access plan rebuilt 210
  - bitmap created 232
  - bitmap merge 235
  - distinct processing 241
  - generic query information 222, 227
  - grouping 248, 251
  - hash table 239
  - host variable and ODP
    - implementation 217
  - index created 194
  - optimizer timed out 213
  - query sort 199
  - SQL information 168
  - STRDBMON/ENDDDBMON
    - commands 229
  - subquery merge 244
  - subquery processing 216
  - table locked 207
  - table scan 181
  - temporary table 203
  - using existing index 186
- summary rows
  - SQL information 253
  - using existing index 259, 261, 263, 265, 266, 268, 269
- symmetrical multiprocessing 47

## T

- table
  - multiple
    - improving performance when selecting data from 57
- table locked
  - summary row 207
- table probe
  - access method 15
- table scan
  - access method 14
  - summary row 181
- table scans
  - output for all queries 88
  - output for SQL queries 88
- temporary bitmap
  - access method 37
  - probe access method 39
  - scan access method 37
- temporary buffer
  - scan access method 44
- temporary index
  - access method 40
  - probe access method 43
  - scan access method 41
- temporary list
  - access method 31
- temporary list scan
  - access method 31
- temporary row number list
  - access method 33
  - probe access method 35
  - scan access method 33
- temporary sorted list
  - access method 27
  - probe access method 28
  - scan access method 27
- temporary table
  - summary row 203
- tools
  - performance 75
- Trace Job (TRCJOB) command 142
- transitive closure 56

## U

- using
  - a copy of the data 139, 152
  - allow copy data (ALWCOPYDATA) 139, 152
  - close SQL cursor (CLOSQLCSR) 139, 143
  - FETCH statement 146
- using existing index
  - summary row 186, 259, 261, 263, 265, 266, 268, 269
- using JOB parameter 110
- using SQL
  - application programs 49

## V

- variable-length data
  - tips 153
- Visual Explain 94
  - index advisor 95

Visual Explain (continued)  
starting 95





Printed in USA