



IBM i

Database

Performance and Query Optimization

7.1





IBM i

Database

Performance and Query Optimization

7.1

Note

Before using this information and the product it supports, read the information in “Notices,” on page 393.

| This edition applies to IBM i 7.1 (product number 5770-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 IBM Corporation 1998, 2010.

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

Contents

Database performance and query optimization 1

| | | |
|--|---|-----|
| | What's new for IBM i 7.1 | 1 |
| | PDF file for Database performance and query optimization | 2 |
| | Query engine overview | 2 |
| | SQE and CQE engines | 3 |
| | Query dispatcher | 4 |
| | Statistics manager | 5 |
| | Global Statistics Cache | 6 |
| | Plan cache | 6 |
| | Data access methods | 8 |
| | Permanent objects and access methods | 8 |
| | Temporary objects and access methods | 22 |
| | Objects processed in parallel | 50 |
| | Spreading data automatically | 51 |
| | Processing queries: Overview | 52 |
| | How the query optimizer makes your queries more efficient | 52 |
| | General query optimization tips | 52 |
| | Access plan validation | 53 |
| | Single table optimization | 54 |
| | Join optimization | 55 |
| | Distinct optimization | 65 |
| | Grouping optimization | 66 |
| | Ordering optimization | 72 |
| | View implementation | 73 |
| | Materialized query table optimization | 75 |
| | Recursive query optimization | 85 |
| | Adaptive Query Processing | 95 |
| | Optimizing query performance using query optimization tools | 99 |
| | DB2 for IBM i – Health Center | 99 |
| | Monitoring your queries using the Database Monitor | 121 |
| | Using System i Navigator with detailed monitors | 132 |
| | Index advisor | 137 |
| | Viewing your queries with Visual Explain | 141 |
| | Optimizing performance using the Plan Cache | 146 |
| | Verifying the performance of SQL applications | 155 |
| | Examining query optimizer debug messages in the job log | 155 |
| | Print SQL Information | 156 |
| | Query optimization tools: Comparison | 157 |
| | Changing the attributes of your queries | 158 |
| | Collecting statistics with the statistics manager | 186 |
| | Displaying materialized query table columns | 192 |
| | Managing check pending constraints columns | 193 |
| | Creating an index strategy | 194 |
| | Binary radix indexes | 194 |

| | | |
|--|---|-----|
| | Encoded vector indexes | 201 |
| | Comparing binary radix indexes and encoded vector indexes | 207 |
| | Indexes & the optimizer | 208 |
| | Indexing strategy | 216 |
| | Coding for effective indexes | 217 |
| | Using indexes with sort sequence | 220 |
| | Index examples | 221 |
| | Application design tips for database performance | 229 |
| | Using live data | 230 |
| | Reducing the number of open operations | 231 |
| | Retaining cursor positions | 234 |
| | Programming techniques for database performance | 236 |
| | Use the OPTIMIZE clause | 236 |
| | Use FETCH FOR n ROWS | 237 |
| | Use INSERT n ROWS | 238 |
| | Control database manager blocking | 239 |
| | Optimize the number of columns that are selected with SELECT statements | 240 |
| | Eliminate redundant validation with SQL PREPARE statements | 240 |
| | Page interactively displayed data with REFRESH(*FORWARD) | 241 |
| | Improve concurrency by avoiding lock waits | 241 |
| | General DB2 for i performance considerations | 242 |
| | Effects on database performance when using long object names | 242 |
| | Effects of precompile options on database performance | 242 |
| | Effects of the ALWCPYDTA parameter on database performance | 243 |
| | Tips for using VARCHAR and VARGRAPHIC data types in databases | 244 |
| | Using field procedures to provide column level encryption | 246 |
| | SYSTOOLS | 249 |
| | Using SYSTOOLS | 249 |
| | Database monitor formats | 251 |
| | Database monitor SQL table format | 251 |
| | Optional database monitor SQL view format | 258 |
| | Query optimizer messages reference | 350 |
| | Query optimization performance information messages | 350 |
| | Query optimization performance information messages and open data paths | 374 |
| | PRTSQLINF message reference | 379 |

Appendix. Notices 393

| | | |
|--|---|-----|
| | Programming interface information | 395 |
| | Trademarks | 395 |
| | Terms and conditions | 395 |

Database performance and query optimization

The goal of database performance tuning is to minimize the response time of your queries by making the best use of your system resources. The best use of these resources involves 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, the applications used on your system, and how the conflicting uses of your database might affect 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 yield the largest performance increases over the widest variety of situations. Focus your analysis on these areas.

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.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 390.

What's new for IBM i 7.1

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

- “Global Statistics Cache” on page 6
- “Encoded vector index index-only access” on page 17
- “Encoded vector index symbol table scan” on page 18
- “Encoded vector index symbol table probe” on page 21
- “Encoded vector index INCLUDE aggregates” on page 18
- “Array unnest temporary table” on page 49
- “Adaptive Query Processing” on page 95
- “Health Center SQL procedures” on page 100
- “Sparse indexes” on page 194
- “View index build status” on page 213
- “Improve concurrency by avoiding lock waits” on page 241
- “Using field procedures to provide column level encryption” on page 246
- “SYSTOOLS” on page 249

What's new

The following revisions or additions have been made to the Performance and query optimization documentation since the first 7.1 publication:

- **April 2016 update :**
 - An additional query option was added to QAQQINI : ALLOW_EVI_ONLY_ACCESS. For details, see “QAQQINI query options” on page 162
- **May 2015 update :**
 - Additional options were added to the QAQQINI query option Memory_Pool_Preference. For details, see “QAQQINI query options” on page 162

- **October 2014 update:**
 - The database monitor topic has been updated: “Monitoring your queries using the Database Monitor” on page 121
 - The SQL Plan Cache topic has been updated: “Optimizing performance using the Plan Cache” on page 146

How to see what's new or changed

To help you see where technical changes have been made, this information uses:

- The  image to mark where new or changed information begins.
- The  image to mark where new or changed information ends.

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

PDF file for Database performance and query optimization

View and print a PDF of this information.

To view or download the PDF version of this document, select Database performance and query optimization (about 5537 KB).

Other information

You can also view or print any of the following PDF files:

- Preparing for and Tuning the SQL Query Engine on DB2[®] for i5/OS 
- SQL Performance Diagnosis on IBM[®] DB2 Universal Database[™] for iSeries 

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 preceding link).
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDF files. You can download a free copy from Adobe (<http://get.adobe.com/reader/>) .

Query engine overview

IBM DB2 for i provides two query engines to process queries: Classic Query Engine (CQE) and 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[®], 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 process that determines which path a query takes can give you a better understanding of query performance.

Within SQE, several more components were created and other existing components were updated. Additionally, new data access methods are possible with SQE that are not supported under CQE.

Related information:

Embedded SQL programming

SQL programming

Query (QSQRY) API

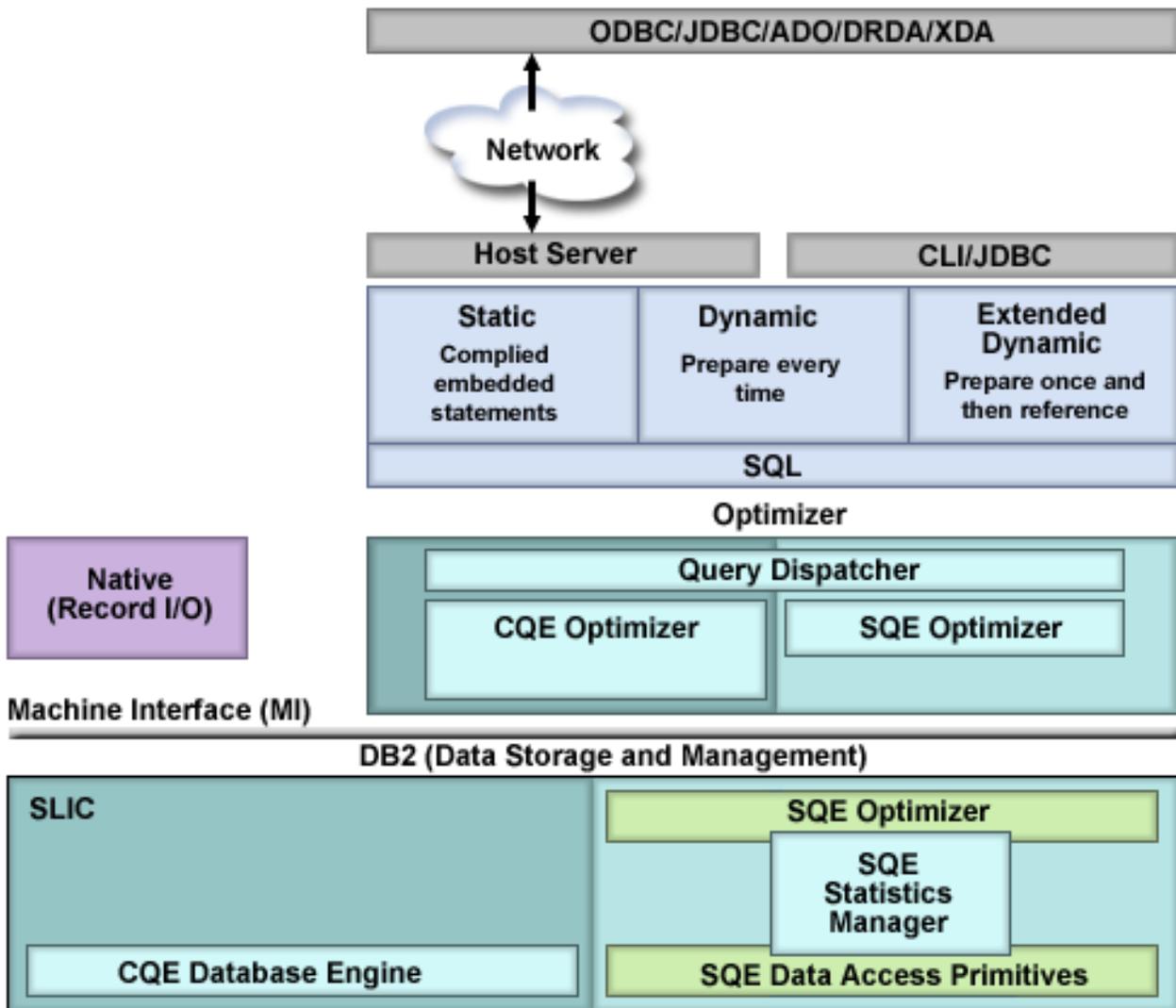
Open Query File (OPNQRYF) command

Run SQL Statements (RUNSQLSTM) command

SQE and CQE engines

| It is important to understand the implementation differences of query management and processing in
| CQE versus SQE.

| The following figure shows an overview of the IBM DB2 for i architecture. It shows the delineation
| between CQE and SQE, how query processing is directed by the query dispatcher, and where each SQE
| component fits. The functional separation of each SQE component is clearly evident. 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 implementation translates into enhanced performance efficiency.



As seen in the previous graphic, the query runs from any query interface to the optimizer and the query dispatcher. The query dispatcher determines whether the query is implemented with CQE or SQE.

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. It cannot be bypassed.

Currently, the dispatcher routes 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
- tables with Read triggers
- Read-only queries with more than 1000 dataspace, or updatable queries with more than 256 dataspace.
- DB2 Multisystem tables
- multi-format logical files
- non-SQL queries, for example the QQQQry API, Query/400, or OPNQRYF

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

Related reference:

“MQT supported function” on page 76

Although an 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.

Statistics manager

In CQE, the retrieval of statistics is a function of the Optimizer. When the Optimizer needs to know information about a table, it looks at the table description to retrieve the row count and table size. If an index is available, the Optimizer might extract information about the data in the table. In SQE, the collection and management of statistics is handled by a separate component called the statistics manager. The statistics manager leverages all the same statistical sources as CQE, but adds more sources and capabilities.

The statistics manager does not actually run or optimize the query. Instead, 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 tracks the following information:

Cardinality of values

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 are 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

The top *m* most frequent values of a column together with a count of how frequently each value occurs. This information is obtained by using 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

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

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. Most 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 on single columns of a table and stored internally. 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.

Related reference:

“Collecting statistics with the statistics manager” on page 186

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 is current.

Global Statistics Cache

In SQE, the DB2 Statistics Manager stores actual row counts into a Global Statistics Cache. In this manner, the Statistics Manager refines its estimates over time as it learns where estimates have deviated from actual row counts.

Both completed queries and currently executing queries might be inspected by the “Adaptive Query Processing” on page 95 (AQP) task, which compares estimated row counts to actual row counts. If there are any significant discrepancies, the AQP task notifies the DB2 Statistics Manager (SM). The SM stores this actual row count (also called observed row count) into a Global Statistics Cache (GSC).

If the query which generated the observed statistic in the GSC is reoptimized, the actual row count estimate is used in determining a new query plan. Further, if a different query asks for the same or a similar row count, the SM could return the stored actual row count from the GSC. Faster query plans can be generated by the query optimizer.

Typically, observed statistics are for complex predicates such as with a join. A simple example is a query joining three files A, B, and C. There is a discrepancy between the estimate and actual row count of the join of A and B. The SM stores an observed statistic into the GSC. Later, if a different join query of A, B, and Z is submitted, the SM recalls the observed statistic of the A and B join. The SM considers that observed statistic in its estimate of the A, B, and Z join.

The Global Statistics Cache is an internal DB2 object, and the contents of it are not directly observable.

Plan cache

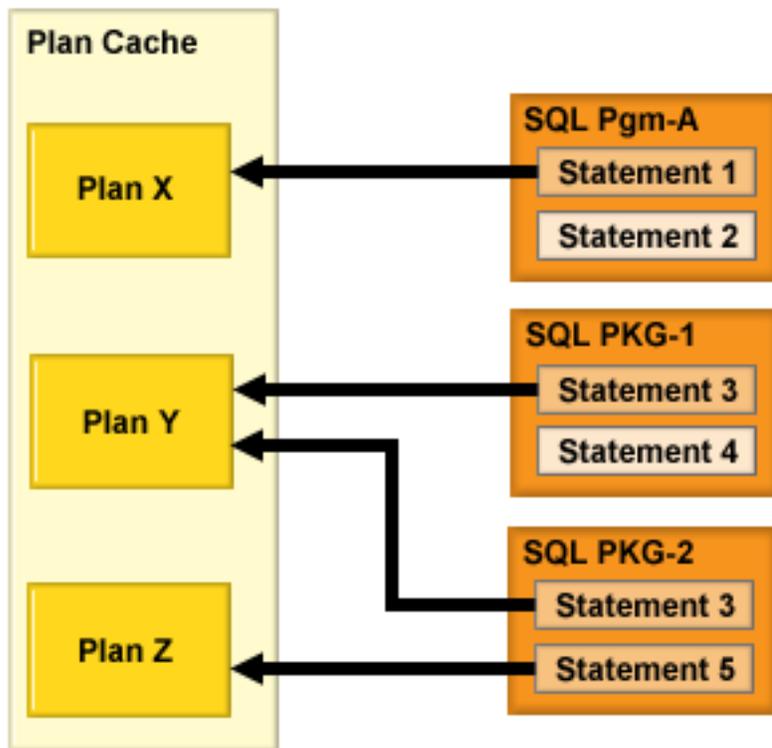
The plan cache is a repository that contains the access plans for queries that were optimized by SQE.

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. The purposes of the plan cache are to:

- Facilitate the reuse of a query access plan when the same query is re-executed
- Store runtime information for subsequent use in future query optimizations
- Provide performance information for analysis and tuning

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, for example, when creating an index, all queries can benefit from this updated access plan. This updated access plan eliminates the need to reoptimize the query, resulting in greater efficiency.

The following graphic shows the concept of reusability of the query access plans stored in the plan cache:



As shown in the previous graphic, statements from packages and programs are stored in unique plans in the plan cache. If Statement 3 exists in both SQL package 1 and SQL package 2, the plan is stored once in the plan cache. The plan cache is interrogated each time a query is executed. If an access plan exists that satisfies the requirements of the query, 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 with new query access plans as they are created. When new statistics or indexes become available, an existing plan is updated the next time the query is run. The plan cache is also automatically updated by the database with runtime information as the queries are run. It is created with an overall size of 512 MB.

Each plan cache entry contains the original query, the optimized query access plan, and cumulative runtime information gathered during the runs of the query. In addition, several instances of query runtime objects are stored with a plan cache entry. These runtime objects are the real executable objects and temporary storage containers (hash tables, sorts, temporary indexes, and so on) used to run the query.

When the plan cache exceeds its designated size, a background task is automatically scheduled to remove plans from the plan cache. Access plans are deleted based upon age, how frequently it is used, and how much cumulative resources (CPU/IO) were consumed.

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 typically been between 10,000 to 20,000 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 for a single SQL statement can be maintained in the plan cache. Although the SQL statement is the primary key into the plan cache, different environmental settings can cause additional access plans to be stored. 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 share of available memory for the job in the current pool
- Different ALWCPYDTA settings
- Different selectivity based on changing host variable values used in selection (WHERE clause)

Currently, the plan cache can maintain a maximum of three different access plans for the same SQL statement. As new access plans are created for the same SQL statement, older 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 conditions include:

- Specifying REOPTIMIZE_ACCESS_PLAN(*YES) or (*FORCE) in the QAQQINI table or in Run SQL Scripts
- Deleting or recreating the table that the access plan refers to
- Deleting an index that is used by the access plan

Related reference:

“Effects of the ALWCPYDTA parameter on database performance” on page 243

Some complex queries can perform better by using a sort or hashing method to evaluate the query instead of using or creating an index.

“Changing the attributes of your queries” on page 158

You can modify different types of query attributes for a job with the **Change Query Attributes (CHGQRYA)** CL command. You can also use the System i® Navigator Change Query Attributes interface.

“Optimizing performance using the Plan Cache” on page 146

The SQL Plan Cache contains a wealth of information about the SQE queries being run through the database. Its contents are viewable through the System i Navigator GUI interface. Certain portions of the plan cache can also be modified.

Data access methods

Data access methods are used 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 only two types of permanent database objects that can be used as source material for a query — tables and indexes. Indexes include binary radix and encoded vector indexes.

In addition, the query engine might need to create temporary objects to hold interim results or references during the execution of an access plan. The DB2 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.

Permanent objects and access methods

There are three basic types of access methods used to manipulate the permanent and temporary database 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.

Table 1. Permanent object data access methods

| Permanent objects | Scan operations | Probe operations |
|----------------------|--|----------------------------|
| Table | Table scan | Table probe |
| Radix index | Radix index scan | Radix index probe |
| Encoded vector index | Encoded vector index symbol table scan | Encoded vector index probe |

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 determines the most efficient way to extract the data from the table in order to satisfy the query. These ways could include scanning or probing the table or using an index to extract the data.

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 the rows in the table to determine if they satisfy the selection criteria specified in the query. It does this processing 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 the rows in order to satisfy the query. The scan operation itself is 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 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"> Minimizes page I/O operations through asynchronous pre-fetching of the rows since the pages are scanned sequentially Requests a larger I/O to fetch the data efficiently |
| Considerations | <ul style="list-style-type: none"> All rows in the table are examined regardless of the selectivity of the query Rows marked as deleted are still paged into memory even though none are selected. You can reorganize the table to remove deleted rows. |
| Likely to be used | <ul style="list-style-type: none"> When expecting many rows returned from the table When the number of large I/Os needed to scan is fewer than the number of small I/Os required to probe the table |

Table 2. Table scan attributes (continued)

| | |
|--------------------------------|---|
| Data access method | Table scan |
| Example SQL statement | SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01'AND 'E01' OPTIMIZE FOR ALL ROWS |
| Messages indicating use | <ul style="list-style-type: none"> Optimizer Debug: CPI4329 – Arrival sequence was used for file EMPLOYEE PRTSQLINF: SQL4010 – Table scan access for table 1. |
| SMP parallel enabled | Yes |
| Also referred to as | Table Scan, Preload |
| Visual Explain icon |  |

Related concepts:

“Nested loop join implementation” on page 55

DB2 for i 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**.

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.

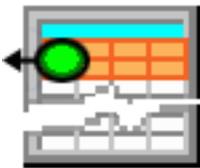
This can include index operations as well as temporary row number lists or bitmaps. The processing for a table probe is typically random. It requests a small I/O to retrieve only the row in question and does not attempt to bring in any extraneous rows. This method leads to efficient processing for smaller result sets because only rows needed to satisfy the query are processed, rather than scanning all rows.

However, since the sequence of the row numbers is not known in advance, little pre-fetching can be performed to bring the data into main memory. This randomness can result in most of the I/Os associated with table probe 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"> Requests smaller I/Os to prevent paging rows into memory that are not needed Can be used with any access method that generates a row number for the table probe to process |
| Considerations | Because of the synchronous random I/O the probe can perform poorly when many rows are selected |

Table 3. Table probe attributes (continued)

| Data access method | Table probe |
|-------------------------|--|
| Likely to be used | <ul style="list-style-type: none"> When row numbers (from indexes or temporary row number lists) are used, but data from the underlying table is required for further processing of the query When processing any remaining selection or projection of the values |
| 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. These example messages illustrate the use of a data access method that generates a row number used to perform the table probe.</p> <ul style="list-style-type: none"> Optimizer Debug: CPI4328 – Access path of file X1 was used by query PRTSQLINF: SQL4008 – Index X1 used for table 1. SQL4011 – Index scan-key row positioning (probe) used on table 1. |
| SMP parallel enabled | Yes |
| Also referred to as | Table Probe, Preload |
| Visual Explain icon |  |

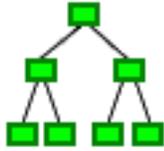
Radix index

An SQL index (or keyed sequence access path) is a permanent object that is created over a table. The index is 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 index. When the optimizer matches a query to index key columns, it can use the index to help satisfy query selection, ordering, grouping, or join requirements.

Typically, using an index also includes a table probe to provide access to columns needed to satisfy the query that cannot be found as index keys. If all the columns necessary to satisfy the query can be found as index keys, then the table probe is not required. 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:

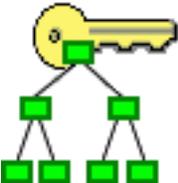
A radix index scan operation is used to retrieve the rows from a table in a keyed sequence. Like a table scan, all the rows in the index are sequentially processed, but the resulting row numbers are sequenced based upon the key columns.

The sequenced rows can be used by the optimizer to satisfy a portion of the query request (such as ordering or grouping). They can also be used to provide faster throughput by performing selection against the index keys rather than all the rows in the table. Since the index I/Os only contain keys, typically more rows can be paged into memory in one I/O than rows in a table with many columns.

Table 4. Radix index scan attributes

| Data access method | Radix index scan |
|--------------------------------|---|
| Description | Sequentially scan and process all 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"> • Only those index entries that match any selection continue to be processed • Potential to extract all the data from the index key values, thus eliminating the need for a Table Probe • Returns the rows back in a sequence based upon the keys of the index |
| Considerations | Generally requires a Table Probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when many rows are selected because of the random I/O associated with the Table Probe. |
| Likely to be used | <ul style="list-style-type: none"> • When asking for or expecting only a few rows to be returned from the index • When sequencing the rows is required for the query (for example, ordering or grouping) • When the selection columns cannot be matched against the leading key columns of the index |
| 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"> • Optimizer Debug: CPI4328 -- Access path of file X1 was used by query. • PRSQLINF: SQL4008 -- Index X1 used for table 1. |
| SMP parallel enabled | Yes |

Table 4. Radix index scan attributes (continued)

| Data access method | Radix index scan |
|---------------------|--|
| Also referred to as | Index Scan Index Scan, Preload Index Scan, Distinct Index Scan Distinct, Preload Index Scan, Key Selection |
| Visual Explain icon |  |

Related reference:

“Effects of the ALWCOPYDTA parameter on database performance” on page 243

Some complex queries can perform better by using a sort or hashing method to evaluate the query instead of using or creating an index.

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 scan is that the rows returned are first identified by a probe operation to subset them.

The optimizer attempts to match the columns used for some or all 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 key values. Only those keys from the series of ranges are paged into main memory.

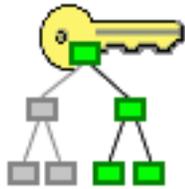
The resulting row numbers generated by the probe can then be further processed by any remaining selection against the index keys or a table probe operation. This method provides for quick access to only the rows of the index that satisfy the selection.

The main function of a radix index probe is to provide quick selection against the index keys. In addition, the row sequencing can be used to satisfy other portions of the query, such as ordering or grouping. Since the index I/Os are only for rows that match the probe selection, no extraneous processing is performed on rows that do not match. This savings in I/Os against rows that are not a part of the result set is one of the primary advantages for this operation.

Table 5. Radix index probe attributes

| Data access method | Radix index probe |
|--------------------|--|
| Description | 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 are used to generate a table row number. |
| Advantages | <ul style="list-style-type: none"> • Only those index entries that match any selection continue to be processed • Provides quick access to the selected rows • Potential to extract all the data from the index key values, thus eliminating the need for a Table Probe • Returns the rows back in a sequence based upon the keys of the index |

Table 5. Radix index probe attributes (continued)

| Data access method | Radix index probe |
|-------------------------|---|
| Considerations | Generally requires a Table Probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when many rows are selected because of the random I/O associated with the Table Probe. |
| Likely to be used | <ul style="list-style-type: none"> • When asking for or expecting only a few rows to be returned from the index • When sequencing the rows is required the query (for example, ordering or grouping) • When the selection columns match the leading key columns of the index |
| Example SQL statement | <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> |
| Messages indicating use | <ul style="list-style-type: none"> • Optimizer Debug: CPI4328 -- Access path of file X1 was used by query. • PRTSQLLINE: SQL4008 -- Index X1 used for table 1. SQL4011 -- Index scan-key row positioning used on table 1. |
| SMP parallel enabled | Yes |
| Also referred to as | Index Probe Index Probe, Preload Index Probe, Distinct Index Probe Distinct, Preload Index Probe, Key Positioning Index Scan, Key Row Positioning |
| Visual Explain icon |  |

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, index X1 is used to position to the first index entry that matches the selection built over both columns LastName and WorkDept. The selection is rewritten into a series of ranges that match all

the leading key columns used from the index X1. The probe is then based upon the composite concatenated values for all the leading keys. The pseudo-SQL for this rewritten SQL might look as follows:

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

All the key entries that satisfy the probe operation are used to generate a row number for the table associated with the index (for example, Employee). The row number is 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 the rows that satisfy the index probe operation have been processed. In this example, all the index entries processed and rows retrieved met the index probe criteria.

Additional selection might be added that cannot use an index probe, such as selection against columns which are not leading key columns of the index. Then the optimizer performs an index scan operation within the range of probed values. This process still allows for selection to be performed before the Table Probe operation.

Related concepts:

“Nested loop join implementation” on page 55

DB2 for i 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**.

Related reference:

“Effects of the ALWCPYDTA parameter on database performance” on page 243

Some complex queries can perform better by using a sort or hashing method to evaluate the query instead of using or creating an index.

Encoded vector index

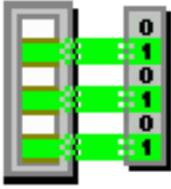
An encoded vector index is a permanent object that provides access to a table. This access is done by assigning codes to distinct key values and then representing those values in a vector.

The size of the vector matches 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 byte, 2 bytes, or 4 bytes in length. The key length depends upon the number of distinct values that need to be represented in the vector. Because of their compact size and relative simplicity, the EVI can be used to process large amounts of data efficiently.

An encoded vector index is used to represent the values stored in a table. However, 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 with a table probe to specify the rows in the table that the query needs to process.

The main difference in the table probe using an encoded vector index vs. a radix index is that the I/O paging 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.

Visual explain icon:



Related concepts:

“Encoded vector indexes” on page 201

An encoded vector index (EVI) is used to provide fast data access in decision support and query reporting environments.

“EVI maintenance” on page 204

There are unique challenges to maintaining EVIs. The following table shows a progression of how EVIs are maintained, the conditions under which EVIs are most effective, and where EVIs are least effective, based on the EVI maintenance characteristics.

Encoded vector index probe:

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.

Table 6. Encoded vector index probe attributes

| Data access method | Encoded vector index probe |
|------------------------------|---|
| Description | 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. |
| Advantages | <ul style="list-style-type: none"> • Only those index entries that match any selection continue to be processed • Provides quick access to the selected rows • Returns the row numbers in ascending sequence so that the Table Probe can be more aggressive in pre-fetching the rows for its operation |
| Considerations | EVIs are usually 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. EVIs always require a Table Probe to be performed on the result of the EVI probe operation. |
| Likely to be used | <ul style="list-style-type: none"> • When the selection columns match the leading key columns of the index • When an encoded vector index exists and savings in reduced I/O against the table justifies the extra cost. This cost includes probing the EVI and fully populating the temporary row number list. |
| Example SQL statement | <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> |

Table 6. Encoded vector index probe attributes (continued)

| Data access method | Encoded vector index probe |
|-------------------------|--|
| Messages indicating use | <ul style="list-style-type: none"> • Optimizer Debug: <ul style="list-style-type: none"> CPI4329 -- Arrival sequence was used for file EMPLOYEE. CPI4338 -- 3 Access path(s) used for bitmap processing of file EMPLOYEE. • PRTSQLINF: <ul style="list-style-type: none"> 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. |
| SMP parallel enabled | Yes |
| Also referred to as | Encoded Vector Index Probe, Preload |
| Visual Explain icon |  |

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 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 few rows are returned, the optimizer probably chooses the binary radix tree index and performs the random I/O against the table. However, selecting more rows causes the optimizer to use the EVIs, because of the savings from the more efficiently scheduled I/O against the table.

Encoded vector index index-only access:

The encoded vector index can also be used for index-only access.

The EVI can be used for more than generating a bitmap or row number list to provide an asynchronous I/O map to the desired table rows. The EVI can also be used by two index-only access methods that can be applied specific to the symbol table itself. These two index-only access methods are the EVI symbol table scan and the EVI symbol table probe.

| These two methods can be used with GROUP BY or DISTINCT queries that can be satisfied by the symbol table. This symbol table-only access can be further employed in aggregate queries by adding INCLUDE values to the encoded vector index.

| The following information is a summary of the symbol table-only scan and probe access methods.

| Use the following links to learn in-depth information.

| **Related concepts:**

| “Encoded vector indexes” on page 201

| An encoded vector index (EVI) is used to provide fast data access in decision support and query reporting environments.

| “How the EVI works” on page 202

| EVIs work in different ways for costing and implementation.

| **Related reference:**

| “Index grouping implementation” on page 67

| There are two primary ways to implement grouping using an index: Ordered grouping and pre-summarized processing.

| **Encoded vector index symbol table scan:**

| An encoded vector index symbol table scan operation is used to retrieve the entries from the symbol table portion of the index.

| All entries (symbols) in the symbol table are sequentially scanned if a scan is chosen. The symbol table can be used by the optimizer to satisfy GROUP BY or DISTINCT portions of a query request.

| Selection is applied to every entry in the symbol table. The selection must be applied to the symbol table keys unless the EVI was created as a sparse index, with a WHERE clause. In that case, a portion of the selection is applied as the symbol table is built and maintained. The query request must include matching predicates to use the sparse EVI.

| All entries are retrieved directly from the symbol table portion of the index without any access to the vector portion of the index. There is also no access to the records in the associated table over which the EVI is built.

| **Encoded vector index INCLUDE aggregates**

| To enhance the ability of the EVI symbol table to provide aggregate answers, the symbol table can be created to contain additional INCLUDE values. These are ready-made numeric aggregate results, such as SUM, COUNT, AVG, or VARIANCE values requested over non-key data. These aggregates are specified using the INCLUDE keyword on the CREATE ENCODED VECTOR INDEX request.

| These included aggregates are maintained in real time as rows are inserted, updated, or deleted from the corresponding table. The symbol table maintains these additional aggregate values in addendum to the EVI keys for each symbol table entry. Because these are numeric results and finite in size, the symbol table is still a desirable compact size.

| These included aggregates are over non-key columns in the table where the grouping is over the corresponding EVI symbol table defined keys. The aggregate can be over a single column or a derivation.

Table 7. Encoded vector index symbol table scan attributes

| Data access method | Encoded vector index symbol table scan |
|--------------------------|--|
| Description | Sequentially scan and process all the symbol table entries associated with the index. When there is selection (WHERE clause), it is applied to every entry in the symbol table. An exception is made in the case of a sparse EVI, where the selection is applied as the index is created and maintained. Selected entries are retrieved directly without any access to the vector or the associated table. |
| Advantages | <ul style="list-style-type: none"> • Pre-summarized results are readily available • Only processes the unique values in the symbol table, avoiding processing table records. • Extract all the data from the index unique key values or INCLUDE values, thus eliminating the need for a Table Probe or vector scan. • With INCLUDE, provides ready-made numeric aggregates, eliminating the need to access corresponding table rows to perform the aggregation |
| Considerations | <p>Dramatic performance improvement for grouping queries where the resulting number of groups is relatively small compared to the number of records in the underlying table. Can perform poorly when there are many groups involved such that the symbol table is large. Poor performance is even more likely if a large portion of the symbol table has been put into the overflow area.</p> <p>Dramatic performance improvement for grouping queries when the aggregate is specified as an INCLUDE value of the symbol table.</p> |
| Likely to be used | <ul style="list-style-type: none"> • When asking for GROUP BY, DISTINCT, COUNT, or COUNT DISTINCT from a single table and the referenced columns are in the key definition. • When the number of unique values in the columns of the key definition is small relative to the number of records in the underlying table. • When there is no selection (WHERE clause) within the query or the selection does not reduce the result set much. • When the symbol table key satisfies the GROUP BY, and requested aggregates, like SUM or COUNT, are specified as INCLUDE values. • when the query is run with commitment control *NONE or *CHG. |

Table 7. Encoded vector index symbol table scan attributes (continued)

| Data access method | Encoded vector index symbol table scan |
|--------------------------------|---|
| Example SQL statement | <pre>CREATE ENCODED VECTOR INDEX EVI1 ON Sales (Region)</pre> <p>Example 1</p> <pre>SELECT Region, count(*) FROM Sales GROUP BY Region OPTIMIZE FOR ALL ROWS</pre> <p>Example 2</p> <pre>SELECT DISTINCT Region FROM Sales OPTIMIZE FOR ALL ROWS</pre> <p>Example 3</p> <pre>SELECT COUNT(DISTINCT Region) FROM Sales</pre> <p>Example 4 uses the INCLUDE option. The sums of revenue and cost of goods per sales region is maintained in real time.</p> <pre>CREATE ENCODED VECTOR INDEX EVI2 ON Sales(Region) INCLUDE(SUM(Revenue), SUM(CostOfGoods))</pre> <pre>SELECT Region, SUM(Revenue), SUM(CostOfGoods) FROM Sales GROUP BY Region</pre> |
| Messages indicating use | <ul style="list-style-type: none"> Optimizer Debug: CPI4328 -- Access path of file EVI1 was used by query. PRTSQLINF: SQL4008 -- Index EVI1 used for table 1.SQL4010 |
| Also referred to as | Encoded Vector Index Table Scan, Preload |
| Visual Explain icon |  |

Related concepts:

“Encoded vector indexes” on page 201

An encoded vector index (EVI) is used to provide fast data access in decision support and query reporting environments.

“How the EVI works” on page 202

EVI works in different ways for costing and implementation.

Related reference:

“Index grouping implementation” on page 67

There are two primary ways to implement grouping using an index: Ordered grouping and pre-summarized processing.

Related information:

SQL INCLUDE statement

| **Encoded vector index symbol table probe:**

| An encoded vector index symbol table probe operation is used to retrieve entries from the symbol table portion of the index. Scanning the entire symbol table is not necessary.

| The symbol table can be used by the optimizer to satisfy GROUP BY or DISTINCT portions of a query request.

| The optimizer attempts to match the columns used for some or all the selection against the leading keys of the EVI index. It then rewrites the selection into a series of ranges that can be used to probe directly into the symbol table. Only those symbol table pages from the series of ranges are paged into main memory.

| The resulting symbol table entries generated by the probe operation can then be further processed by any remaining selection against EVI keys. This strategy provides for quick access to only the entries of the symbol table that satisfy the selection.

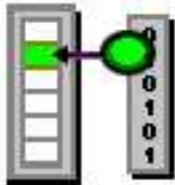
| Like an encoded vector symbol table scan, a symbol table probe can return ready-made aggregate results if INCLUDE is specified when the EVI is created.

| All entries are retrieved directly from the symbol table portion of the index without any access to the vector portion of the index. In addition, it is unnecessary to access the records in the associated table over which the EVI is built.

| *Table 8. Encoded vector index symbol table probe attributes*

| Data access method | Encoded vector index symbol table probe |
|--------------------|--|
| Description | |
| Advantages | Probe the symbol table entries associated with the index. When there is selection (WHERE clause), it is applied to every entry in the symbol table that meets the probe criteria. If there are sparse EVIs, the selection is applied as the EVI is created and maintained. Selected entries are retrieved directly without any access to the vector or the associated table. |
| Considerations | <ul style="list-style-type: none"> • Pre-summarized results are readily available • Only processes the unique values in the symbol table, avoiding processing table records. • Extracts all the data from the index unique key values or include values, or both, thus eliminating the need for a table probe or vector scan • With INCLUDE, provides ready-made numeric aggregates, eliminating the need to access corresponding table rows to perform the aggregation |
| Likely to be used | <ul style="list-style-type: none"> • When asking for GROUP BY, DISTINCT, COUNT, or COUNT DISTINCT from a single table and the referenced columns are in the key definition. • When the number of unique values in the columns of the key definition is small relative to the number of records in the underlying table. • When there is selection (WHERE clause) that reduces the selection from the Symbol Table and the WHERE clause involves leading, probable keys. • When the symbol table key satisfies the GROUP BY and the WHERE clause reduces selection to the leading keys, and aggregates are specified as INCLUDE values. • When the query is run with commitment control *NONE or *CHG. |

Table 8. Encoded vector index symbol table probe attributes (continued)

| Data access method | Encoded vector index symbol table probe |
|-------------------------|---|
| Example SQL statement | <pre>CREATE ENCODED VECTOR INDEX EVI1 ON Sales (Region)</pre> <p>Example 1</p> <pre>SELECT Region, COUNT(*) FROM Sales WHERE Region in ('Quebec', 'Manitoba') GROUP BY Region OPTIMIZE FOR ALL ROWS</pre> <p>Example 2</p> <pre>CREATE ENCODED VECTOR INDEX EVI2 ON Sales(Region) INCLUDE(SUM(Revenue), SUM(CostOfGoods)) SELECT Region, SUM(Revenue), SUM(CostOfGoods) FROM Sales WHERE Region = 'PACIFIC' GROUP BY Region</pre> |
| Messages indicating use | <ul style="list-style-type: none"> Optimizer Debug: CPI4328 -- Access path of file EVI1 was used by query. PRTSQLINF: SQL4008 -- Index EVI1 used for table 1.SQL4010 |
| Also referred to as | Encoded Vector Index Table Probe, Preload |
| Visual Explain icon |  |

Related concepts:

- “Encoded vector indexes” on page 201
- An encoded vector index (EVI) is used to provide fast data access in decision support and query reporting environments.
- “How the EVI works” on page 202
- EVI works in different ways for costing and implementation.

Related reference:

- “Index grouping implementation” on page 67
- There are two primary ways to implement grouping using an index: Ordered grouping and pre-summarized processing.

Related information:

- SQL INCLUDE statement

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 9. Temporary object data access methods

| Temporary create objects | Scan operations | Probe operations |
|--------------------------------|----------------------|-----------------------|
| Temporary hash table | Hash table scan | Hash table probe |
| Temporary sorted list | Sorted list scan | Sorted list probe |
| Temporary distinct sorted list | Sorted list scan | N/A |
| Temporary list | List scan | N/A |
| Temporary values list | Values list scan | N/A |
| Temporary row number list | Row number list scan | Row number list probe |
| Temporary bitmap | Bitmap scan | Bitmap probe |
| Temporary index | Temporary index scan | Temporary index probe |
| Temporary buffer | Buffer scan | N/A |
| Queue | N/A | N/A |
| Array unnest temporary table | Temporary table scan | 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. The hash table remains resident within main memory to avoid any I/Os associated with either the scan or probe against the temporary object. The optimizer determines the optimal hash table size based on the number of unique column combinations used as keys for the creation.

Additionally the hash table can be populated with all the necessary columns to satisfy any further processing. This population avoids any random I/Os associated with a table probe operation.

However, the optimizer can selectively include columns in the hash table when the calculated size exceeds the memory pool storage available for the query. In these 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 can populate the hash table with distinct values. If the query contains grouping or distinct processing, then all the rows with the same key value are not required in the hash table. The rows are still collated, but the distinct processing is performed during the population of the hash table itself. This method allows a simple scan 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 the entries contained within the hash table are processed.

The optimizer considers a hash table scan when the data values need to be collated together, but sequencing of the data is not required. A hash table scan allows the optimizer to generate a plan that takes advantage of any non-join selection while creating the temporary hash table.

An additional benefit is that the temporary hash table data structure will typically cause the table data to remain resident within main memory after creation. Resident table data reduces paging on the subsequent hash table scan operation.

Table 10. Hash table scan attributes

| Data access method | Hash table scan |
|-------------------------|---|
| Description | Read all the entries in a temporary hash table. The hash table can perform distinct processing to eliminate duplicates. Or the temporary hash table can collate all the rows with the same value together. |
| Advantages | <ul style="list-style-type: none">• Reduces the random I/O to the table associated with longer running queries that might otherwise use an index to collate the data• Selection can be performed before generating the hash table to subset the number of rows in the temporary object |
| Considerations | 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">• When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA)• When the data is required to be collated based upon a column or columns for distinct or grouping |
| 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 indicates a hash scan was used.</p> <ul style="list-style-type: none">• Optimizer Debug: CPI4329 -- Arrival sequence was used for file EMPLOYEE.• PRSQLINE: SQL4010 -- Table scan access for table 1. SQL4029 -- Hashing algorithm used to process the grouping. |
| SMP parallel enabled | Yes |
| Also referred to as | Hash Scan, Preload Hash Table Scan Distinct Hash Table Scan Distinct, Preload |

Table 10. Hash table scan attributes (continued)

| Data access method | Hash table scan |
|---------------------|---|
| 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. When the hash table is probed, the values used to probe into the hash table are extracted from the join-from criteria specified in the selection.

These values are sent through the same hashing algorithm used to populate the temporary hash table. They determine if any rows have a matching equal value. All the matching join rows are then returned to be further processed by the query.

Table 11. Hash table probe attributes

| Data access method | Hash table probe |
|-----------------------|--|
| Description | The temporary hash table is quickly probed based upon the join criteria. |
| Advantages | <ul style="list-style-type: none"> • Provides quick access to the selected rows that match probe criteria • Reduces the random I/O to the table associated with longer running queries that use an index to collate the data • Selection can be performed before generating the hash table to subset the number of rows in the temporary object |
| Considerations | 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. |
| Likely to be used | <ul style="list-style-type: none"> • When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) • When the data is required to be collated based upon a column or columns for join processing • The join criteria was specified using an equals (=) operator |
| Example SQL statement | <pre>SELET * FROM Employee XXX, Department YYY WHERE XXX.WorkDept = YYY.DeptNbr OPTIMIZE FOR ALL ROWS</pre> |

Table 11. Hash table probe attributes (continued)

| Data access method | Hash table probe |
|-------------------------|---|
| Messages indicating use | <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 indicates a hash probe was used.</p> <ul style="list-style-type: none"> • Optimizer Debug: <ul style="list-style-type: none"> CPI4327 -- File EMPLOYEE processed in join position 1. CPI4327 -- File DEPARTMENT processed in join position 2. • PRSQLINF: <ul style="list-style-type: none"> 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. |
| SMP parallel enabled | Yes |
| Also referred to as | <p>Hash Table Probe, Preload</p> <p>Hash Table Probe Distinct</p> <p>Hash Table Probe Distinct, Preload</p> |
| Visual Explain icon |  |

The hash table probe access method is 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 in selecting rows from the underlying table, without regard for join criteria. This single pass through the underlying table can now use a table scan or existing index to select the rows needed for the hash table population.

Since hash tables are constructed so that most of the hash table remains 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 is required to finish processing this table. This method causes further I/O savings.

Related concepts:

“Nested loop join implementation” on page 55

DB2 for i 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**.

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 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 the necessary columns to satisfy any further processing. This population avoids any random I/Os associated with a table probe operation.

However, the optimizer can selectively include columns in the sorted list when the calculated size exceeds 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 the entries contained within the sorted list are processed.

A sorted list scan is considered when the data values need to be sequenced. A sorted list scan allows the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary sorted list.

An additional benefit is that the data structure will usually cause the table data within the sorted list to remain resident within main memory after creation. This resident data reduces paging on the subsequent sorted list scan operation.

Table 12. Sorted list scan attributes

| Data access method | Sorted list scan |
|-----------------------|---|
| Description | Read all the entries in a temporary sorted list. The sorted list can perform distinct processing to eliminate duplicate values or take advantage of the temporary sorted list to sequence all the rows. |
| Advantages | <ul style="list-style-type: none"> • Reduces the random I/O to the table associated with longer running queries that would otherwise use an index to sequence the data. • Selection can be performed prior to generating the sorted list to subset the number of rows in the temporary object |
| Considerations | 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. |

Table 12. Sorted list scan attributes (continued)

| Data access method | Sorted list scan |
|-------------------------|--|
| Likely to be used | <ul style="list-style-type: none"> When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) When the data is required to be ordered based upon a column or columns for ordering or distinct processing |
| Example SQL statement | <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> |
| Messages indicating use | <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 indicates a sorted list scan was used.</p> <ul style="list-style-type: none"> Optimizer Debug: <pre>CPI4328 -- Access path of file X1 was used by query. CPI4325 -- Temporary result file built for query.</pre> PRTSQLINF: <pre>SQL4008 -- Index X1 used for table 1. SQL4002 -- Reusable ODP sort used.</pre> |
| SMP parallel enabled | No |
| Also referred to as | <p>Sorted List Scan, Preload</p> <p>Sorted List Scan Distinct</p> <p>Sorted List Scan Distinct, Preload</p> |
| Visual Explain icon |  |

Sorted list probe:

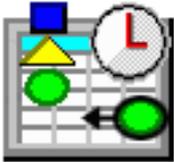
A sorted list probe operation is used to retrieve rows from a temporary sorted list based upon a probe lookup operation.

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

Table 13. Sorted list probe attributes

| Data access method | Sorted list probe |
|--------------------|---|
| Description | The temporary sorted list is quickly probed based upon the join criteria. |

Table 13. Sorted list probe attributes (continued)

| Data access method | Sorted list probe |
|--------------------------------|--|
| Advantages | <ul style="list-style-type: none"> • Provides quick access to the selected rows that match probe criteria • Reduces the random I/O to the table associated with longer running queries that otherwise use an index to collate the data • Selection can be performed before generating the sorted list to subset the number of rows in the temporary object |
| Considerations | 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. |
| Likely to be used | <ul style="list-style-type: none"> • When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) • When the data is required to be collated based upon a column or columns for join processing • The join criteria was specified using a non-equals operator |
| Example SQL statement | <pre>SELECT * FROM Employee XXX, Department YYY WHERE XXX.WorkDept > YYY.DeptNo OPTIMIZE FOR ALL ROWS</pre> |
| Messages indicating use | <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 indicates a sorted list probe was used.</p> <ul style="list-style-type: none"> • Optimizer Debug: <pre>CPI4327 -- File EMPLOYEE processed in join position 1. CPI4327 -- File DEPARTMENT processed in join position 2.</pre> • 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. SQL4010 -- Table scan access for table 2.</pre> |
| SMP parallel enabled | Yes |
| Also referred to as | Sorted List Probe, Preload Sorted List Probe Distinct Sorted List Probe Distinct, Preload |
| Visual Explain icon |  |

The sorted list probe access method is 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 optimizer chooses the most efficient implementation to select the rows from the underlying table without regard to any join criteria. This single pass through the underlying table can use a Table Scan or an existing index to select the rows needed to populate the sorted list.

Since sorted lists are constructed so that most of the temporary object remains resident within main memory, the sorted list I/O is minimal. If the sorted list was populated with all necessary table columns, no additional Table Probe is required to finish processing the table, causing further I/O savings.

Related concepts:

“Nested loop join implementation” on page 55

DB2 for i 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**.

Temporary distinct sorted list

A temporary distinct sorted list combines the features of the temporary hash table and the temporary sorted list.

Like the hash table, the temporary distinct sorted list allows the optimizer to collate the rows based on a column or set of columns. Like the sorted list, the temporary distinct sorted list also allows the optimizer to sequence the rows.

A temporary distinct sorted list contains a hash table data structure set up for efficient access to aggregate rows during population. In addition, a binary tree data structure is maintained over the hash table data structure so that the data can be accessed in sequence. The sorted aspect of the data structure allows for the efficient computation of super-aggregate rows in SQL statements that contain GROUP BY ROLLUP.

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

Visual explain icon:



Sorted list scan:

During the sorted list scan, the entire temporary distinct sorted list is scanned and all the entries contained within the temporary are processed.

The optimizer uses the sorted list scan when the data values need to be aggregated and sequenced. The optimizer generates this plan that can take advantage of any non-join selection while creating the temporary distinct sorted list. The data structure of the temporary distinct sorted list will typically cause the table data to remain resident within main memory after creation. This memory-resident data reduces paging on the subsequent sorted list scan.

Table 14. Sorted list scan attributes

| Data access method | Sorted list scan |
|--------------------|--|
| Description | Reads all the entries in a temporary distinct sorted list |
| Advantages | <ul style="list-style-type: none"> • Allows efficient computation of ROLLUP super-aggregate rows. • Reduces the random I/O to the table associated with longer running queries that might otherwise use an index to collate the data. • Selection can be performed before generating the distinct sorted list to subset the number of rows in the temporary object. |

Table 14. Sorted list scan attributes (continued)

| | |
|--------------------------------|---|
| Data access method | Sorted list scan |
| Considerations | Used for GROUP BY ROLLUP processing. Can perform poorly when the entire temporary object does not stay resident in memory as it is being processed. |
| Likely to be used | <ul style="list-style-type: none"> • When the use of temporary results is allowed in the query environmental parameter (ALWCPYDTA) • When a GROUP BY ROLLUP is in the SQL statement |
| Messages indicating use | N/A |
| SMP parallel enabled | Yes |
| Also referred to as | N/A |
| Visual Explain icon |  |

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 various reasons, some of which include an overly complex view or derived table, Symmetric Multiprocessing (SMP) or 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 is 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 15. List scan attributes

| | |
|---------------------------|---|
| Data access method | List scan |
| Description | Sequentially scan and process all the rows in the temporary list. |

Table 15. List scan attributes (continued)

| Data access method | List scan |
|--------------------------------|--|
| Advantages | <ul style="list-style-type: none"> The temporary list and list scan can be used by the optimizer to minimize repetition of an operation or to simplify the optimizer logic flow. Selection can be performed before generating the list to subset the number of rows in the temporary object. |
| Considerations | 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"> When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA). When DB2 Symmetric Multiprocessing is used for the query. |
| 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 indicates a list scan was used.</p> <ul style="list-style-type: none"> 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> 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> |
| 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 the rows of the DEPARTMENT table for each join possibility, the selection against the DEPARTMENT table is performed once. The results are stored in the temporary list. The temporary list is then scanned for the Cartesian product join.

Temporary values list

The temporary values list allows the optimizer to store rows of data specified in a VALUES clause of a SELECT or CREATE VIEW statement.

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.

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

Visual explain icon:



Values list scan:

During a values list scan operation, the entire temporary values list is scanned and all the rows of data are processed.

Table 16. Values list scan attributes

| Data access method | Values list scan |
|--------------------------------|---|
| Description | Sequentially scan and process all the rows of data in the temporary values list. |
| Advantages | The temporary values list and values list scan can be used by the optimizer to simplify the optimizer logic flow. |
| Likely to be used | When a VALUES clause is specified in the from-clause of an SQL fullselect |
| Example SQL statement | <pre>SELECT EMPNO, 'empproject' FROM EMPPROJACT WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112') UNION VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new')</pre> |
| Messages indicating use | <p>There are multiple ways in which a values list scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine indicates a values list scan was used.</p> <ul style="list-style-type: none"> • Optimizer Debug: CPI4329 -- Arrival sequence was used for file *VALUES. • PRTSQLINF: SQL4010 -- Table scan access for table 1. |
| SMP parallel enabled | Yes |
| Also referred to as | Values List, Preload |
| Visual Explain icon | |

Temporary row number list

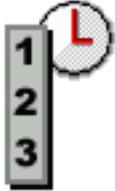
The temporary row number list is a temporary object that allows the optimizer to sequence rows based upon their row address (their row number). The row number list can be either scanned or probed by the optimizer to satisfy different operations of the query.

A temporary row number list is a data structure where the rows are organized for quick and efficient retrieval. The row number list only contains the row number for the associated row. Since no table data is present, a table probe operation is typically associated with it in order to retrieve the underlying table data. Because the row numbers are sorted, the random I/O associated with the table probe operation is

performed more efficiently. The database manager performs pre-fetch or look-ahead logic to determine if multiple rows are located on adjacent pages. If so, the table probe requests 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:

The entire temporary row number list is scanned and all the row addresses contained within the row number list are processed. The optimizer considers this plan when there is an applicable encoded vector index or if the index probe or scan random I/O can be reduced. The random I/O can be reduced by first preprocessing and sorting the row numbers associated with the Table Probe.

The use of a row number list scan allows 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 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 table paging never visits 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 is that the list scan is over a list of row addresses while the bitmap scan is over a bitmap representing the addresses.

Table 17. Row number list scan

| Data access method | Row number list scan |
|--------------------------|--|
| Description | Sequentially scan and process all 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"> • The temporary row number list only contains address, no data, so the temporary can be efficiently scanned within memory. • The row numbers contained within the temporary object are sorted to provide efficient I/O processing to access the underlying table. • Selection is performed as the row number list is generated to subset the number of rows in the temporary object. |
| Considerations | Since the row number list contains only the addresses of the selected rows in the table, a separate Table Probe fetches the table rows. |
| Likely to be used | <ul style="list-style-type: none"> • When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA). • 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. • When multiple indexes over the same table need to be combined in order to minimize the number of selected rows. |

Table 17. Row number list scan (continued)

| Data access method | Row number list scan |
|-------------------------|--|
| 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 OPTIMIZE FOR 99999 ROWS</pre> |
| Messages indicating use | <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 indicates a row number list scan was used.</p> <ul style="list-style-type: none"> Optimizer Debug: <pre>CPI4329 -- Arrival sequence was used for file EMPLOYEE. CPI4338 -- 3 Access path(s) used for bitmap processing of file EMPLOYEE.</pre> PRTSQLINF: <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> |
| SMP parallel enabled | Yes |
| Also referred to as | Row Number List Scan, Preload |
| Visual Explain icon |  |

Using the example above, the optimizer created a temporary row number list for each of the indexes used by this query. These indexes included a radix index and two encoded vector indexes. Each index row number list was scanned and merged into a final composite row number list representing the intersection of all the index row number lists. The final row number list is then used by the Table Probe to determine which rows are selected and processed for the query results.

Row number list probe:

A row number list probe 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 it matches the selection used to generate the list.

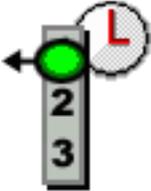
The use of a row number list probe operation allows the optimizer to generate a plan that can take advantage of any sequencing provided by an index, but still use the row number list to perform additional selection before any Table probe operations.

A row number list probe is identical to a bitmap probe operation. The only difference is that the list probe is over a list of row addresses while the bitmap probe is over a bitmap representing the addresses.

Table 18. 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"> • The temporary row number list only contains a row address, no data, so the temporary can be efficiently probed within memory. • The row numbers represented within the row number list are sorted to provide efficient lookup processing to test the underlying table. • Selection is performed as the row number list is generated to subset the number of selected rows in the temporary object. |
| Considerations | Since the row number list contains only the addresses of the selected rows in the table, a separate Table Probe fetches the table rows. |
| Likely to be used | <ul style="list-style-type: none"> • When the use of temporary results is allowed by the query environmental parameter (ALWCOPYDTA). • 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. • When multiple indexes over the same table need to be combined in order to minimize the number of selected rows. |
| 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 indicates a row number list probe was used.</p> <ul style="list-style-type: none"> • 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> • 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> |
| SMP parallel enabled | Yes |
| Also referred to as | Row Number List Probe, Preload |

Table 18. Row number list probe (continued)

| Data access method | Row number list probe |
|---------------------|---|
| 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 requires that the resulting rows be sequenced by the WorkDept column, the row number list cannot be scanned for 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 list with the row address extracted from the index probe, the sequencing of the keys in the index X1 is preserved. The row can still be tested against the selected rows within the row number list.

Temporary bitmap

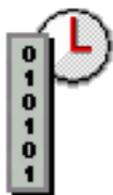
The temporary bitmap is a temporary object that allows the optimizer to sequence rows based upon their row address (their row number). The bitmap can be either scanned or probed by the optimizer to satisfy different operations of the query.

A temporary bitmap is a data structure that uses a bitmap to represent all the row numbers for a table. Since each row is represented by a separate bit, all the rows within a table can be represented in a fairly condensed form. When a row is selected, the bit within the bitmap that corresponds to the selected row is set on. After the temporary bitmap is populated, all the selected rows can be retrieved in a sorted manner for quick and efficient retrieval. The temporary bitmap only represents the row number for the associated selected rows.

No table data is present within the temporary bitmap. A table probe operation is typically associated with the bitmap 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 performs pre-fetch or look-ahead logic to determine if multiple rows are located on adjacent pages. If so, the table probe requests 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 the row addresses contained within the bitmap are processed. The optimizer considers this plan when there is an applicable

encoded vector index or if the index probe or scan random I/O can be reduced. The random I/O can be reduced by first preprocessing and sorting the row numbers associated with the Table Probe.

The use of a bitmap scan allows 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 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 table paging never visits 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 is that the list scan is over a list of row addresses while the bitmap scan is over a bitmap representing the addresses.

Table 19. Bitmap scan attributes

| Data access method | Bitmap scan attributes |
|------------------------------|--|
| Description | Sequentially scan and process all 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. |
| Advantages | <ul style="list-style-type: none"> • The temporary bitmap only contains a reference to a row address, no data, so the temporary can be efficiently scanned within memory. • The row numbers represented within the temporary object are sorted to provide efficient I/O processing to access the underlying table. • Selection is performed as the bitmap is generated to subset the number of selected rows in the temporary object. |
| Considerations | Since the bitmap contains only the addresses of the selected rows in the table, a separate Table Probe fetches the table rows. |
| Likely to be used | <ul style="list-style-type: none"> • When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA). • 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. • When multiple indexes over the same table need to be combined in order to minimize the number of selected rows. |
| 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 OPTIMIZE FOR 99999 ROWS</pre> |

Table 19. Bitmap scan attributes (continued)

| Data access method | Bitmap scan attributes |
|--------------------------------|---|
| Messages indicating use | <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 indicates a bitmap scan was used.</p> <ul style="list-style-type: none"> • Optimizer Debug: <ul style="list-style-type: none"> CPI4329 -- Arrival sequence was used for file EMPLOYEE. CPI4338 -- 3 Access path(s) used for bitmap processing of file EMPLOYEE. • PRSQLINF: <ul style="list-style-type: none"> 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. |
| SMP parallel enabled | Yes |
| Also referred to as | Bitmap Scan, Preload Row Number Bitmap Scan Row Number Bitmap Scan, Preload Skip Sequential Scan |
| Visual Explain icon |  |

Using the example above, the optimizer created a temporary bitmap for each of the indexes used by this query. These indexes included a radix index and two encoded vector indexes. Each index temporary bitmap was scanned and merged into a final composite bitmap representing the intersection of all the index temporary bitmaps. The final bitmap is then used by the Table Probe operation to determine which rows are selected and 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 it matches the selection used to generate the bitmap.

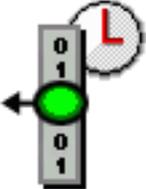
The use of a bitmap probe operation allows the optimizer to generate a plan that can take advantage of any sequencing provided by an index, but still use the bitmap to perform additional selection before any Table Probe operations.

A bitmap probe is identical to a row number list probe operation. The only difference is that the list probe is over a list of row addresses while the bitmap probe is over a bitmap representing the addresses.

Table 20. 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"> • The temporary bitmap only contains a reference to a row address, no data, so the temporary can be efficiently probed within memory. • The row numbers represented within the bitmap are sorted to provide efficient lookup processing to test the underlying table. • Selection is performed as the bitmap is generated to subset the number of selected rows in the temporary object. |
| Considerations | Since the bitmap contains only the addresses of the selected rows in the table, a separate Table Probe fetches the table rows. |
| Likely to be used | <ul style="list-style-type: none"> • When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA). • When the cost of creating and probing the bitmap is justified by reducing the number of Table Probe operations that must be performed. • When multiple indexes over the same table need to be combined in order to minimize the number of selected rows. |
| 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 indicates a bitmap probe was used.</p> <ul style="list-style-type: none"> • 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> • 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> |
| SMP parallel enabled | Yes |
| Also referred to as | <p>Bitmap Probe, Preload</p> <p>Row Number Bitmap Probe</p> <p>Row Number Bitmap Probe, Preload</p> |

Table 20. Bitmap probe attributes (continued)

| Data access method | Bitmap probe attributes |
|---------------------|---|
| 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 requires that the resulting rows be sequenced by the WorkDept column, the bitmap cannot be scanned for 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 bitmap with the row address extracted from the index probe, the sequencing of the keys in the index X1 is preserved. The row can still be tested 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 the same attributes and benefits as a radix index created through the CREATE INDEX SQL statement or **Create Logical File (CRTLF)** CL command.

Additionally, the temporary index is optimized for use by the optimizer to satisfy a specific query request. This optimization includes setting the logical page size and applying any selection to the index to speed up its use after creation.

The temporary index can be used to satisfy various query requests:

- Ordering
- Grouping/Distinct
- Joins
- Record selection

Generally a temporary index is a more expensive temporary object to create than other temporary objects. It can be populated by a table scan, or by one or more index scans or probes. The optimizer considers all the methods available when determining which method to use to produce the rows for the index creation. This process is like the costing and selection of the other temporary objects used by the optimizer.

One significant advantage of the temporary index over other temporary objects is that it is the only temporary object maintained if the underlying table changes. The temporary index is identical to a radix index in that any inserts or updates against the table are reflected immediately through normal index maintenance.

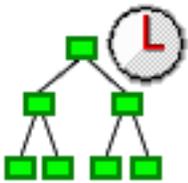
SQE usage of temporary indexes is different from CQE usage in that SQE allows reuse. References to temporary indexes created and used by the SQE optimizer are kept in the system Plan Cache. A temporary index is saved for reuse by other instances of the same query or other instances of the same query running in a different job. It is also saved for potential reuse by a different query that can benefit from the use of the same temporary index.

By default, an SQE temporary index persists until the Plan Cache entry for the last referencing query plan is removed. You can control this behavior by setting the CACHE_RESULTS QAQQINI value. The default for this INI value allows the optimizer to keep temporary indexes around for reuse.

Changing the INI value to '*JOB' prevents the temporary index from being saved in the Plan Cache; the index does not survive a hard close. The *JOB option causes the SQE optimizer use of temporary indexes to behave more like the CQE optimizer. The temporary index has a shorter life, but is still shared as long as there are active queries using it. This behavior can be desirable in cases where there is concern about increased maintenance costs for temporary indexes that persist for reuse.

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

Visual explain icon:



Temporary index scan:

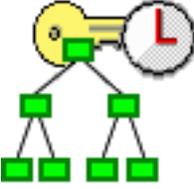
A temporary index scan operation is identical to the index scan operation that is performed upon the permanent radix index. It is still used to retrieve the rows from a table in a keyed sequence; however, the temporary index object must first be created. All the rows in the index are sequentially processed, but the resulting row numbers are 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).

Table 21. Temporary index scan attributes

| Data access method | Temporary index scan |
|------------------------------|--|
| Description | Sequentially scan and process all the keys associated with the temporary index. |
| Advantages | <ul style="list-style-type: none"> • Potential to extract all the data from the index key values, thus eliminating the need for a Table Probe • Returns the rows back in a sequence based upon the keys of the index |
| Considerations | Generally requires a Table Probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when many rows are selected because of the random I/O associated with the Table Probe. |
| Likely to be used | <ul style="list-style-type: none"> • When sequencing the rows is required for the query (for example, ordering or grouping) • When the selection columns cannot be matched against the leading key columns of the index • When the overhead cost associated with the creation of the temporary index can be justified against other alternative methods to implement this query |
| Example SQL statement | <pre>SELECT * FROM Employee WHERE WorkDept BETWEEN 'A01' AND 'E01' ORDER BY LastName OPTIMIZE FOR ALL ROWS</pre> |

Table 21. Temporary index scan attributes (continued)

| Data access method | Temporary index scan |
|-------------------------|--|
| Messages indicating use | <ul style="list-style-type: none"> Optimizer Debug: CPI4321 -- Access path built for file EMPLOYEE. PRTSQLINF: SQL4009 -- Index created for table 1. |
| 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 determines 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 considered when determining how selection is performed.

Temporary index probe:

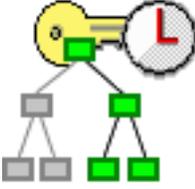
A temporary index probe operation is identical to the index probe operation that is performed on the permanent radix index. Its main function is to provide quick access against the index keys of the temporary index. However, it can still be used to retrieve the rows from a table in a keyed sequence.

The temporary index is used by the optimizer to satisfy the join portion of the query request.

Table 22. Temporary index probe attributes

| Data access method | Temporary index probe |
|--------------------|---|
| Description | 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 is used to generate a table row number. |

Table 22. Temporary index probe attributes (continued)

| Data access method | Temporary index probe |
|--------------------------------|---|
| Advantages | <ul style="list-style-type: none"> • Only those index entries that match any selection continue to be processed. Provides quick access to the selected rows • Potential to extract all the data from the index key values, thus eliminating the need for a Table Probe • Returns the rows back in a sequence based upon the keys of the index |
| Considerations | Generally requires a Table Probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when many rows are selected because of the random I/O associated with the Table Probe. |
| Likely to be used | <ul style="list-style-type: none"> • When the ability to probe the rows required for the query (for example, joins) exists • When the selection columns cannot be matched against the leading key columns of the index • When the overhead cost associated with the creation of the temporary index can be justified against other alternative methods to implement this query |
| Example SQL statement | <pre> SELECT * FROM Employee XXX, Department YYY WHERE XXX.WorkDept = YYY.DeptNo OPTIMIZE FOR ALL ROWS </pre> |
| Messages indicating use | <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 indicates a temporary index probe was used.</p> <ul style="list-style-type: none"> • 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> • PRSQLLINE: <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> |
| SMP parallel enabled | Yes |
| Also referred to as | <p>Index Probe</p> <p>Index Probe, Preload</p> <p>Index Probe, Distinct</p> <p>Index Probe Distinct, Preload</p> <p>Index Probe, Key Selection</p> |
| Visual Explain icon |  <p>The icon depicts a yellow key with a red 'L' on its handle. The key is positioned above a tree diagram. The tree has a root node (a green square) which branches into two child nodes (grey squares). Each of these child nodes further branches into two more child nodes (green squares). The key's handle points to the root node of this tree structure.</p> |

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 difference between a temporary buffer and a temporary list is that the buffer does not need to be fully populated before its results are processed.

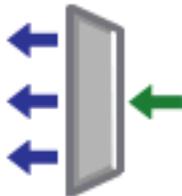
The temporary buffer acts as a serialization point between parallel and non-parallel portions of a query. The operations used to populate the buffer cannot be performed in parallel, whereas the operations that fetch rows from the buffer can be performed in parallel.

The temporary buffer is required for SQE because the index scan and index probe operations are not SMP parallel-enabled for this engine. Unlike CQE, which performs these index operations in parallel, SQE does not subdivide the index operation work 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. Any remaining work within the query is 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 is used when a query is processed using DB2 Symmetric Multiprocessing, yet a portion of the query is unable to be parallel processed. 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 is 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 need to be fully populated before the start of the scan operation. A temporary list requires that the list is fully populated before fetching any rows.

Table 23. Buffer scan attributes

| Data access method | Buffer scan |
|--------------------|---|
| Description | Sequentially scan and process all the rows in the temporary buffer. Enables SMP parallelism to be performed over a non-parallel portion of the query. |

Table 23. Buffer scan attributes (continued)

| Data access method | Buffer scan |
|--------------------------------|--|
| Advantages | <ul style="list-style-type: none"> • The temporary buffer can be used to enable parallelism over a portion of a query that is non-parallel • The temporary buffer does not need to be fully populated in order to start fetching rows |
| Considerations | 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"> • When the query is attempting to take advantage of DB2 Symmetric Multiprocessing • When a portion of the query cannot be performed in parallel (for example, index scan or index probe) |
| 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"> • 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> • PRTSQLINF: <pre>SQL4027 -- Access plan was saved with DB2 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> |
| SMP parallel enabled | Yes |
| Also referred to as | Not applicable |
| 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 Table Probe operation for this query, DB2 Symmetric Multiprocessing is used to perform the random probe into the table. Since the index probe is not SMP parallel-enabled for SQE, it is placed within a temporary buffer to control access to the selected index entries.

Queue

The Queue is a temporary object that the optimizer uses to feed recursion by putting data values needed for the recursion on it. This data typically includes those values used on the recursive join predicate, and other recursive data accumulated or manipulated during the recursive process.

The Queue has two operations allowed:

- Enqueue: puts data on the queue
- Dequeue: takes data off the queue

A queue is an efficient data structure because it contains only the data needed to feed the recursion or directly modified by the recursion process. Its size is managed by the optimizer.

Unlike other temporary objects created by the optimizer, the queue is not populated all at once by the underlying query node tree. It is a real-time temporary holding area for values feeding the recursion. In this regard, a queue is not considered temporary, as it does not prevent the query from running if `ALWCOPYDATA(*NO)` was specified. The data can flow from the query at the same time the recursive values are inserted into the queue and used to retrieve additional join rows.

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

Visual explain icon:



Enqueue:

During an enqueue operation, an entry is put on the queue. The entry contains key values used by the recursive join predicates or data manipulated as a part of the recursion process. The optimizer always supplies an enqueue operation to collect the required recursive data on the query node directly above the Union All.

Table 24. Enqueue Attributes

| | |
|--------------------------------|--|
| Data Access Method | Enqueue |
| Description | Places an entry on the queue needed to cause further recursion |
| Advantages | <ul style="list-style-type: none"> • Required as a source for the recursion. Only enqueues required values for the recursion process. Each entry has short life span, until it is dequeued. • Each entry on the queue can seed multiple iterative fullselects that are recursive from the same RCTE or view. |
| Likely to be used | A required access method for recursive queries |
| Example SQL statement | <pre> WITH RPL (PART, SUBPART, QUANTITY) AS (SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY FROM PARTLIST ROOT WHERE ROOT.PART = '01' UNION ALL SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY FROM RPL PARENT, PARTLIST CHILD WHERE PARENT.SUBPART = CHILD.PART) SELECT DISTINCT PART, SUBPART, QUANTITY FROM RPL </pre> |
| Messages indicating use | There are no explicit messages that indicate the use of an enqueue |
| SMP parallel enabled | Yes |
| Also referred to as | Not applicable |

Table 24. Enqueue Attributes (continued)

| Data Access Method | Enqueue |
|---------------------|---|
| Visual Explain icon |  |

Use the CYCLE option in the definition of the recursive query if the data reflecting the parent-child relationship could be cyclic, causing an infinite recursion loop. CYCLE prevents already visited recursive key values from being put on the queue again for a given set of related (ancestry chain) rows.

Use the SEARCH option in the definition of the recursive query to return the results of the recursion in the specified parent-child hierarchical ordering. The search choices are Depth or Breadth first. Depth first means that all the descendents of each immediate child are returned before the next child is returned. Breadth first means that each child is returned before their children are returned.

SEARCH requires not only the specification of the relationship keys, the columns which make up the parent-child relationship, and the search type of Depth or Breadth. It also requires an ORDER BY clause in the main query on the provided sequence column in order to fully implement the specified ordering.

Dequeue:

During a dequeue operation, an entry is taken off the queue. Those values specified by recursive reference are fed back in to the recursive join process.

The optimizer always supplies a corresponding enqueue, dequeue pair of operations for each recursive common table expression or recursive view in the specifying query. Recursion ends when there are no more entries to pull off the queue.

Table 25. Dequeue Attributes

| Data Access Method | Dequeue |
|--------------------|---|
| Description | Removes an entry off the queue. Minimally, provides one side of the recursive join predicate that feeds the recursive join and other data values that are manipulated through the recursive process. The dequeue operation is always on the left side of the inner join with constraint, where the right side is the target child rows. |
| Advantages | <ul style="list-style-type: none"> • Provides quick access to recursive values • Allows for post selection of local predicate on recursive data values |
| Likely to be used | <ul style="list-style-type: none"> • A required access method for recursive queries • A single dequeued value can feed the recursion of multiple iterative fullselects that reference the same RCTE or view |

Table 25. Dequeue Attributes (continued)

| Data Access Method | Dequeue |
|-------------------------|--|
| Example SQL statement | <pre> WITH RPL (PART, SUBPART, QUANTITY) AS (SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY FROM PARTLIST ROOT WHERE ROOT.PART = '01' UNION ALL SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY FROM RPL PARENT, PARTLIST CHILD WHERE PARENT.SUBPART = CHILD.PART) SELECT DISTINCT PART, SUBPART, QUANTITY FROM RPL </pre> |
| Messages indicating use | There are no explicit messages that indicate the use of the dequeue operation. |
| SMP parallel enabled | Yes |
| Also referred to as | Not applicable |
| Visual Explain icon |  |

Array unnest temporary table

The array unnest temporary table is a temporary object that holds the output of an UNNEST of an array or a list of arrays. It can be viewed vertically, with each column of array values having the same format. The temporary table contains one or more arrays specified by the user in an UNNEST clause of a SELECT statement.

UNNEST creates a temporary table with the arrays specified as columns in the table. If more than one array is specified, the first array provides the first column in the result table. The second array provides the second column, and so on.

The arrays might be of different lengths. Shorter arrays are primed with nulls to match the length of the longest array in the list.

If WITH ORDINALITY is specified, an extra counter column of type BIGINT is appended to the temporary table. The ordinality column contains the index position of the elements in the arrays.

The array unnest temporary table is an internal data structure and can only be created by the database manager.



Visual explain icon:

Related reference:

“QAQQINI query options” on page 162
 There are different options available for parameters in the QAQQINI file.

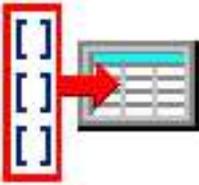
Related information:

- | Array support in SQL procedures
- | Debugging an SQL routine
- | table-reference

| **Array unnest temporary table scan:**

| During an array unnest temporary table scan operation, the temporary table is processed one row at a time.

| *Table 26. Array unnest temporary table scan operation*

| | |
|--------------------------------|--|
| Data access method | Array unnest temporary table scan |
| Description | Sequentially scan and process all the rows of data in the unnest temporary table. |
| Advantages | The array unnest temporary table and temporary table scan can be used to simplify the logic flow of the optimizer for processing arrays. |
| Likely to be used | When an UNNEST clause is specified in the from-clause of an SQL fullselect. |
| Example SQL statement | <pre>CREATE PROCEDURE processCustomers() BEGIN DECLARE ids INTARRAY; DECLARE names STRINGARRAY; set ids = ARRAY[5,6,7]; set names = ARRAY['Ann', 'Bob', 'Sue']; INSERT INTO customerTable(id, name, order) (SELECT Customers.id, Customers.name, Customers.order FROM UNNEST(ids, names) WITH ORDINALITY AS Customers(id, name, order)); END CALL processCustomers()</pre> |
| Messages indicating use | <p>There are multiple ways in which an array unnest temporary table scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine indicates an array unnest temporary table scan was used.</p> <ul style="list-style-type: none"> • Optimizer Debug: CPI4329 -- Arrival sequence was used for file *UNNEST_1. • PRSQLINF: SQL4010 -- Table scan access for table 1. |
| SMP parallel enabled | Yes |
| Also referred to as | |
| Visual Explain icon |  |

| **Objects processed in parallel**

The DB2 Symmetric Multiprocessing feature provides the optimizer with additional methods for retrieving data that include parallel processing. Symmetrical multiprocessing is a form of parallelism achieved on a single system where multiple CPU and I/O processors sharing memory and disk work simultaneously toward a single result.

This parallel processing means that the database manager can have more than one (or all) of the system processors working on a single query simultaneously. The performance of a CPU-bound query can be improved with this feature on multiple-processor systems by distributing the processor load across more than one processor.

The preceding tables indicate what data access methods are enabled to take advantage of the DB2 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 by one of the following methods:

- System value QQRYDEGREE
- Query option file
- DEGREE parameter on the **Change Query Attributes (CHGQRYA)** command
- SQL SET CURRENT DEGREE statement

Once parallelism has been enabled, a set of database system tasks or threads is created at system 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 the tasks do much of the parallel I/O and CPU processing, the I/O and CPU resource accounting is 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 must be run in a shared storage pool with the *CALC paging option, as this method causes more efficient use of active memory.

Related concepts:

“Nested loop join implementation” on page 55

DB2 for i 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**.

Related reference:

“Changing the attributes of your queries” on page 158

You can modify different types of query attributes for a job with the **Change Query Attributes (CHGQRYA)** CL command. You can also use the System i Navigator Change Query Attributes interface.

Related information:

SET CURRENT DEGREE statement

Performance system values: Parallel processing for queries and indexes

Adjusting performance automatically

Work with Active Jobs (WRKACTJOB) command

Change Query Attributes (CHGQRYA) command

DB2 Symmetric Multiprocessing

Spreading data automatically

DB2 for i automatically spreads the data across the disk devices available in the auxiliary storage pool (ASP) where the data is allocated. This process 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 for i 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 unevenness

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 could be spread again by saving, deleting, and then restoring the table.

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

Processing queries: Overview

This overview of the query optimizer provides guidelines for designing queries that perform and use system 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 are still correct.

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

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

- How the system accesses data.
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 Symmetric Multiprocessing operating system feature.
- Query optimizer
The query optimizer identifies the valid techniques which can be used to implement the query and selects the most efficient technique.

How the query optimizer makes your queries more efficient

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. Understand the techniques employed by the query optimizer for performing this task.

The optimizer is an important part of DB2 for i 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.

General query optimization tips

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

- Create indexes whose leftmost key columns match your selection predicates to help supply the optimizer with selectivity values (key range estimates).
- For join queries, create indexes that match your join columns to help the optimizer determine the average number of matching rows.
- Minimize extraneous mapping by specifying only columns of interest on the query. For example, specify only the columns you need to query on the SQL SELECT statement instead of specifying SELECT *. Also, specify FOR FETCH ONLY if the columns do not need to be updated.

- If your queries often use table scan, use the **Reorganize Physical File Member (RGZPFM)** command to remove deleted rows from tables, or the **Change Physical File (CHGPF)** 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 IBM i Access ODBC driver and Query Management driver always use this mode. If ALWCPYDTA(*YES) is specified, the query optimizer attempts to implement the query without copies of the data, but might create copies if required. If ALWCPYDTA(*NO) is specified, copies of the data are 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.

Related information:

Reorganize Physical File Member (RGZPFM) command

Change Physical File (CHGPF) command

Access plan validation

An access plan is a control structure that describes the actions necessary to satisfy each query request. It 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.

To improve performance, an access plan is saved once it is built (see following exceptions), to be available for potentially future runs of the query. However, the optimizer has dynamic replan capability. This means that even if a previously built (and saved) plan is found, the optimizer could rebuild it if a more optimal plan is possible. This process allows for maximum flexibility while still taking advantage of saved plans.

- For dynamic SQL, an access plan is created at prepare or open time. However, optimization uses the host variable values to determine an optimal plan. Therefore, a plan built at prepare time could be rebuilt the first time the query is opened (when the host variable values are present).
- For an IBM i program that contains static embedded SQL, an access plan is initially created at compile time. Again, since optimization uses the host variable values to determine an optimal plan, the compile-time plan could be rebuilt the first time the query is opened.
- For Open Query File (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.

In all the preceding cases where a plan is saved, including static SQL, dynamic replan can still apply as the queries are run over time.

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 a query by calculating an *implementation cost* based on the current state of the database. The optimizer uses two costs in its decision: 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 in 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 key range estimate when the selection predicates match the index left-most keys, or by available column statistics.
- 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 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 CPU needed to Index Probe or Index Scan, and possible Table Probes.
 - 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.

SQE orders the indexes so that the best indexes are examined first. Once an index is found 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. The time limit 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 takes to actually execute the query. Dynamic SQL queries are subject to the optimizer time restrictions. Static SQL query 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. For CQE, the optimizer generally considers five or six indexes for each table of a join before running out of optimization time. Because of this processing, it is normal for the optimizer to spend longer lengths of time analyzing queries against the tables.

- Determines the cost of using a temporary bitmap
 - Order the indexes that can be used for bit mapping. In general the indexes that select the smallest number of entries are examined first.
 - Determine the cost of using this index for bit mapping 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 for i 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.

Nested loop join implementation

DB2 for i 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 is 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 is implemented using either an index or a hash table.

Index nested loop join implementation

During the join, DB2 for i:

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. Chooses the access to the first secondary table:
 - If using an index, Radix Index Probe is used to locate the first row satisfying the join condition for the secondary table. The probe uses 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, many searches could be against the later dials. Any inefficiencies in the processing of the later dials can significantly inflate the query processing time. This reason is 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 could be created. Some join queries build temporary indexes over secondary dials even when an index exists for all the join keys. Because efficiency is important for secondary dials of longer running queries, the optimizer could build a temporary index containing only entries with 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, a hash temporary result table is created containing all rows from 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 the structure of an index. Less CPU processing is required to build and probe a hash table.
- The rows in the hash result table contain all the data required by the query. 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 system.
 - If using a Sorted List Probe, a sorted list result is created containing all the rows from 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, locate the first row that satisfies the join condition or local row selection columns of the secondary table. The join could 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 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 satisfying the join condition can be selected, the processing backs up to the logical previous dial. It 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 eligible to be the primary, then the optimizer costs the join with that table as the primary table, performing the grouping and ordering with an index.
- If ordering and grouping is specified on two or more tables or if temporary objects are allowed, DB2 for i 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 method allows the optimizer to consider any table of the join query as a candidate for the primary table.
 2. Perform the ordering or grouping 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 is 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 method 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.

Related concepts:

“Objects processed in parallel” on page 50

The DB2 Symmetric Multiprocessing feature provides the optimizer with additional methods for retrieving data that include parallel processing. Symmetrical multiprocessing is a form of parallelism achieved on a single system where multiple CPU and I/O processors sharing memory and disk work simultaneously toward a single result.

Related reference:

“Table scan” on page 9

A table scan is the easiest and simplest operation that can be performed against a table. It sequentially processes all the rows in the table to determine if they satisfy the selection criteria specified in the query. It does this processing in a way to maximize the I/O throughput for the table.

“Sorted list probe” on page 28

A sorted list probe operation is used to retrieve rows from a temporary sorted list based upon a probe lookup operation.

“Hash table probe” on page 25

A hash table probe operation is used to retrieve rows from a temporary hash table based upon a probe lookup operation.

“Radix index probe” on page 13

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 scan is that the rows returned are first identified by a probe operation to subset them.

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 following situations:

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

Join specifications not implemented for the dial are deferred until a later dial or, if an inner join, processed as row selection.

For a given dial, the only join specifications which are usable as join columns are those being joined to a *previous* dial. For example, the second dial can only use 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 like the index created for a select/omit keyed logical file. The temporary index for the previous example has key columns of EMPNO and EMSTDATE.

Since the optimizer tries a combination of join and local row selection, you can achieve almost all the advantages of a temporary index by using an existing index. In the preceding example, using either implementation, an existing index could be used or a temporary index could be created. A temporary index is built with the local row selection on PROJNO applied during the index creation. The temporary index has 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), the local row selection can be applied **at the same time** as the join selection. This method contrasts to applying the local selection before the join selection, as happens when the temporary index is created. Or applying the local selection after the join selection, as happens when only the first key column of the index matches the join column.

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

Join order optimization

- | The SQE optimizer allows join reordering for a join logical file. However, the join order is fixed if CQE
- | runs a query that references a join logical file. The join order is also fixed if the OPNQRYF
- | JORDER(*FILE) parameter is specified. In addition, the join order is fixed if 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 ordering or grouping is processed in one step, 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 might 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 forced, the optimizer loops through the dials in the order specified, determining the lowest cost access methods.

Related information:

Open Query File (OPNQRYF) command

Change Query Attributes (CHGQRYA) command

Full outer join

Full outer join is supported by the SQE optimizer. Just as right outer and right exception join are rewritten to the supported join types of inner, left outer or left exception, a full outer join is also rewritten.

A full outer join of A FULL OUTER JOIN B is equivalent to a (A LEFT OUTER JOIN B) UNION ALL (B LEFT EXCEPTION JOIN A). The following example illustrates the rewrite.

```
SELECT EMPNO, LASTNAME, DEPTNAME
FROM CORPDATA.EMPLOYEE XXX
FULL OUTER JOIN CORPDATA.DEPARTMENT YYY
ON XXX.WORKDEPT = YYY.DEPTNO
```

This query is rewritten as the following:

```
SELECT EMPNO, LASTNAME, DEPTNAME
FROM CORPDATA.EMPLOYEE XXX
LEFT OUTER JOIN CORPDATA.DEPARTMENT YYY
ON XXX.WORKDEPT = YYY.DEPTNO
UNION ALL
SELECT EMPNO, LASTNAME, DEPTNAME
FROM CORPDATA.DEPARTMENT YYY
LEFT EXCEPTION JOIN CORPDATA.EMPLOYEE XXX
ON XXX.WORKDEPT = YYY.DEPTNO
```

A query with multiple FULL OUTER JOIN requests, such as A FULL OUTER JOIN B FULL OUTER JOIN C can quickly become complicated in this rewritten state. This complication is illustrated in the following example.

If not running in live data mode, the optimizer could facilitate performance both during optimization and runtime by encapsulating intermediate results in a temporary data object. This object can be optimized once and plugged into both the scanned and probed side of the rewrite. These shared temporary objects eliminate the need to make multiple passes through the specific tables to satisfy the request.

In this example, the result of the (A FULL OUTER JOIN B) is a candidate for encapsulation during its FULL OUTER join with C.

```
A FULL OUTER JOIN B FULL OUTER JOIN C
```

This query is rewritten as the following:

```
((A LEFT OUTER JOIN B) UNION ALL (B LEFT EXCEPTION JOIN A)) LEFT OUTER JOIN C )
UNION ALL
(C LEFT EXCEPTION JOIN ((A LEFT OUTER JOIN B) UNION ALL (B LEFT EXCEPTION JOIN A)))
```

FULL OUTER implies that both sides of the join request can generate NULL values in the resulting answer set. Local selection in the WHERE clause of the query could result in the appropriate downgrade of the FULL OUTER to a LEFT OUTER or INNER JOIN.

If you want FULL OUTER JOIN behavior and local selection applied, specify the local selection in the ON clause of the FULL OUTER JOIN, or use common table expressions. For example:

```
WITH TEMPEMP AS (SELECT * FROM CORPDATA.EMPLOYEE XXX WHERE SALARY > 10000)
SELECT EMPNO, LASTNAME, DEPTNAME
FROM TEMPEMP XXX
FULL OUTER JOIN CORPDATA.DEPARTMENT YYY
ON XXX.WORKDEPT = YYY.DEPTNO
```

Join cost estimation and index selection

As the query optimizer compares the various possible access choices, it must assign a numeric cost value to each candidate. The optimizer uses 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

In steps 3 and 5 in “Join order optimization” on page 58, the optimizer estimates cost and chooses an access method for a given dial combination. The choices made are like the choices for row selection, except that a plan using a probe must be chosen.

The costing value 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 might be loaded into active memory as the result of a **Set Object Access (SETOBJACC)** CL command. Use of this command could significantly improve the performance of a query. However, the 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 system. No allowance is given for system 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 system running the query.
- The values in a column are uniformly distributed across the table. For example, if 10% of the table rows have the same value, then on average, every 10th row in the table contains that value.
- The column values are independent from any other column values in a row, unless there is an index available whose key definition is (A, B). Multi-key field indexes allow the optimizer to detect when the values between columns are correlated.

For example, a column named A has a value of 1 in 50% of the rows in a table. A column named B has a value of 2 in 50% of the rows. 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 in the join cost calculation for secondary dials are:

- the number of rows selected in all previous dials

- the number of rows which match, on average, each of the rows selected from previous dials.

Both of these factors can be derived by estimating the number of matching rows for a given dial.

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

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

For example, when the join operator is less-than, the expected number of matching rows is 0.33 * (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.

Related information:

Set Object Access (SETOBJACC) command

Transitive closure predicates

For join queries, the query optimizer could 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.

See the following examples:

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

The optimizer modifies the query to:

```
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 omits 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 (OPNQRYF %WLDCRD, or *CT).
- The predicate is not connected to other predicates by OR.

The query optimizer generates a new predicate, whether a predicate exists in the WHERE clause (OPNQRYF QRYSLT parameter).

Some predicates are redundant. Redundant predicates occur 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 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 (LPG)

A special type of transitive closure called look ahead predicate generation (LPG) might be costed for joins. In this case, the optimizer tries to minimize the random I/O of a join by pre-applying the query results to a large fact table. LPG is typically 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 could decide to internally modify the query to be:

```
WITH HT AS (SELECT *
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 places the results of the "subquery" into a temporary hash table. The hash table of the subquery can be applied in one of two methods against the EMP_ACT (fact) table:

- The distinct values of the hash tables are retrieved. For each distinct value, an index over EMP_ACT is probed to determine which records are returned for that value. Those record identifiers are normally then stored and sorted (sometimes the sorting is omitted, depending on the total number of record ids expected). Once the ids are determined, the subset of EMP_ACT records can be accessed more efficiently than in a traditional nested loop join processing.
- EMP_ACT can be scanned. For each record, the hash table is probed to see if the record joins at all to EMPLOYEE. This method allows for efficient access to EMP_ACT with a more efficient record rejection method than in a traditional nested loop join process.

Note: LPG processing is part of the normal processing in the SQL Query Engine. CQE only considers the first method, requires that the index in question by an EVI and also requires use of the STAR_JOIN and FORCE_JOIN_ORDER QAQQINI options.

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

The following suggestion is only applicable to CQE and 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 CQE optimizer does not generate transitive closure predicates between two 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.

If the select-statement you are considering accesses two or more tables, all the recommendations suggested in "Creating an index strategy" on page 194 apply. 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

Multiple join types (inner, left outer, right outer, left exception, and right exception) can be specified in the query using the JOIN syntax. However, the DB2 for i can only support one join type of inner, left outer, or left exception join for the entire query. The optimizer determines the overall join type for the query and reorders the files to achieve the correct semantics.

Note: This section does not apply to SQE or OPNQRYF.

The optimizer evaluates the join criteria, along with any row selection, to determine the join type for each dial and the entire query. Then the optimizer generates additional selection using the relative row number of the tables to simulate the different types of joins that occur within the query.

Null values are returned for any unmatched rows in either a left outer or an exception join. Any isolatable selection specified for that dial, including any additional join criteria specified in the WHERE clause, causes all the unmatched rows to be eliminated. (The exception is when the selection is for an IS NULL predicate.) This elimination causes the dial join type to change 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 is evaluated as an "inner join" condition. The second is an isolatable selection predicate that eliminates any unmatched rows. Either of these predicates can cause the join type for the DEPARTMENT table to change from a left outer join to an inner join.

Even though the join between the EMPLOYEE and DEPARTMENT tables was changed to an inner join, the entire query remains 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. The number of rows satisfying the join criteria can become large before selection that either selects or omits the unmatched rows based on that individual dial join type is applied.

Sources of join query performance problems

The optimization algorithms described earlier benefit most join queries, but the performance of a few queries might be degraded.

This occurs when:

- An index is not available which provides average number of duplicate values statistics for the potential join columns.
- The optimizer uses default filter factors to estimate the number of rows when applying local selection to the table when indexes or column statistics do not exist over the selection columns.
Creating indexes over the selection columns allows the optimizer to make a more accurate filtering estimate by using key range estimates.
- The particular values selected for the join columns yield a 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.

Join performance tips

If you have a join query performing poorly, or you are creating an application which uses join queries, these tips could be useful.

Table 27. Checklist for Creating an Application that Uses Join Queries

| What to Do | How It Helps |
|---|---|
| <p>Check the database design. Make sure that there are indexes available over all the join columns and row selection columns or both. The optimizer provides index advice in several places to aid in this process:</p> <ul style="list-style-type: none"> • the index advisor under System i Navigator - Database • the advised information under Visual Explain • the advised information in the 3020 record in the database monitor | <p>The query optimizer can select an efficient access method because it can determine the average number of duplicate values. Many queries could use the existing index and avoid the cost of creating a temporary index or hash table.</p> |
| <p>Check the query to see whether some complex predicates could be added to other dials to allow the optimizer to get better selectivity for each dial.</p> | <p>The query optimizer does not add predicates for predicates connected by OR or non-isolatable predicates, or predicate operator LIKE. Modify the query by adding additional predicates to help.</p> |
| <p>Specify ALWCPYDTA(*OPTIMIZE) or ALWCPYDTA(*YES)</p> | <p>The query is creating a temporary index or hash table, and the processing time could be better if the existing index or hash table was used. Specify ALWCPYDTA(*YES).</p> <p>The query is not creating a temporary index or hash table, and the processing time could be better if a temporary index was created. Specify ALWCPYDTA(*OPTIMIZE).</p> <p>Alternatively, specify OPTIMIZE FOR n ROWS to inform the optimizer that the application reads every resulting row. Set n to a large number. You can also set n to a small number before ending the query.</p> |
| <p>For OPNQRYF, specify OPTIMIZE(*FIRSTIO) or OPTIMIZE(*ALLIO)</p> | <p>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 biases the optimizer toward using existing objects. If you want to optimize the retrieval time for the entire answer set, use *ALLIO. This option could cause the optimizer to create temporary indexes or hash tables to minimize I/O.</p> |

Table 27. Checklist for Creating an Application that Uses Join Queries (continued)

| What to Do | How It Helps |
|---|--|
| Star join queries | <p>A join in which one table is joined with all secondary tables consecutively is sometimes called a star join. If all secondary join predicates contain a column reference to a particular table, place that table 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 could be changed to force EMPLOYEE into join position one by using the query options file (QAQQINI) FORCE_JOIN_ORDER parameter of *YES. In these examples, the join type is a join with no default values returned (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 using 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. | Ordering is specified and all key columns are from a single dial. The optimizer can consider all possible join orders with ALWCPYDTA(*OPTIMIZE). |
| Specify join predicates to prevent all the rows from one table from being joined to every row in the other table. | Improves performance by reducing the join fan-out. It is best if every secondary table has at least one join predicate that references one of its columns as a 'join-to' column. |

Distinct optimization

Distinct is used to compare a value with another value.

There are two methods to write a query that returns distinct values in SQL. One method uses the DISTINCT keyword:

```

SELECT DISTINCT COL1, COL2
FROM TABLE1

```

The second method uses GROUP BY:

```

SELECT COL1, COL2
FROM TABLE1
GROUP BY COL1, COL2

```

All queries that contain a `DISTINCT`, and are run using SQE, rewritten into queries using `GROUP BY`. This rewrite enables queries using `DISTINCT` to take advantage of the many grouping techniques available to the optimizer.

Distinct to Grouping implementation

The following example query has a `DISTINCT`:

```
SELECT DISTINCT COL1, COL2
FROM T1
WHERE COL2 > 5 AND COL3 = 2
```

The optimizer rewrites it into this query:

```
SELECT COL1, COL2
FROM T1
WHERE COL2 > 5 AND COL3 = 2
GROUP BY COL1, COL2
```

Distinct removal

A query containing a `DISTINCT` over whole-file aggregation (no grouping or selection) allows the `DISTINCT` to be removed. For example, look at this query with `DISTINCT`:

```
SELECT DISTINCT COUNT(C1), SUM(C1)
FROM TABLE1
```

The optimizer rewrites this query as the following:

```
SELECT COUNT(C1), SUM(C1)
FROM TABLE1
```

If the `DISTINCT` and the `GROUP BY` fields are identical, the `DISTINCT` can be removed. If the `DISTINCT` fields are a subset of the `GROUP BY` fields (and there are no aggregates), the `DISTINCT`s can be removed.

Grouping optimization

DB2 for i has certain techniques to use when the optimizer encounters grouping. The query optimizer chooses its methods for optimizing your query.

Hash grouping 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 is most likely longer than other grouping implementations because the hash table must be built and populated first. Once the hash table is 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 hash grouping method is most effective

The hash grouping 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 becomes too large. The size in turn slows 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 ratio estimate of the optimizer 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, except for 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

There are two primary ways to implement grouping using an index: Ordered grouping and pre-summarized processing.

Ordered grouping

This implementation uses the Radix Index Scan or the Radix Index Probe access methods to perform the grouping. An index is required that contains all 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 the key values grouped, the first group result can be returned in less time than the hashing method. This index performance is faster because the hashing method requires a temporary result. This implementation can be beneficial if an application does not need to retrieve all the group results, or if an index exists that matches the grouping columns.

When the grouping is implemented with an index and a permanent index does not 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.

Pre-summarized processing

This SQE-only implementation uses an Encoded Vector Index to extract the summary information already in the symbol table of the index. The EVI symbol table contains the unique key values and a count of the number of table records that have that unique value. The grouping for the columns of the index key is already performed. If the query references a single table and performs simple aggregation, the EVI might be used for quick access to the grouping results. For example, consider the following query:

```
SELECT COUNT(*), col1
FROM t1
GROUP BY col1
```

If an EVI exists over t1 with a key of col1, the optimizer can rewrite the query to access the precomputed grouping answer in the EVI symbol table.

This rewrite can result in dramatic improvements when the number of table records is large and the number of resulting groups is small, relative to the size of the table.

This method is also possible with selection (WHERE clause), as long as the reference columns are in the key definition of the EVI.

For example, consider the following query:

```
SELECT COUNT(*), col1
FROM t1
WHERE col1 > 100
GROUP BY col1
```

This query can be rewritten by the optimizer to use the EVI. This pre-summarized processing works for DISTINCT processing, GROUP BY and for column function COUNT. All columns of the table referenced in the query must also be in the key definition of the EVI.

So, for example, the following query can be made to use the EVI:

```
SELECT DISTINCT col1
FROM t1
```

However, this query cannot:

```
SELECT DISTINCT col1
FROM t1
WHERE col2 > 1
```

This query cannot use the EVI because it references col2 of the table, which is not in the key definition of the EVI. If multiple columns are defined in the EVI key, for example, col1 and col2, it is important to use the left-most columns of the key. For example, if an EVI existed with a key definition of (col1, col2), but the query referenced only col2, it is unlikely the EVI is used.

| **EVI INCLUDE aggregates**

| A more powerful example of pre-summarized processing can be facilitated by the use of the INCLUDE keyword on the index create. By default, COUNT(*) is implied on the creation of an EVI. Additional numeric aggregates specified over non-key data can further facilitate pre-determined or ready-made aggregate results during query optimization.

| For example, suppose the following query is a frequently requested result set, queried in whole or as part of a subquery comparison.

```
| SELECT AVG(col2)
| FROM t1
| GROUP BY col1
```

| Create the following EVI to predetermine the value of AVG(col2).

```
| CREATE ENCODED VECTOR INDEX eviT1 ON t1(col1) INCLUDE(AVG(col2))
```

| eviT1 delivers distinct values for col1 and COUNT(*) specific to the group by of col1. eviT1 can be used to generate an asynchronous bitmap or RRN list for accessing the table rows for specific col1 values. In addition, eviT1 computes an additional aggregate, AVG(col2), over the same group by column (col1) by specifying the INCLUDE aggregate.

| INCLUDE aggregates are limited to those aggregates that result in numeric values: SUM, COUNT, AVG, STDDEV, and so on. These values can be readily maintained as records are inserted, deleted, or updated in the base table.

| MIN or MAX are two aggregates that are not supported as INCLUDE aggregates. Deleting the current
| row contributing to the MIN or MAX value would result in the need to recalculate, potentially accessing
| many rows, and reducing performance.

| INCLUDE values can also contain aggregates over derivations. For example, if you have a couple of
| columns that contribute to an aggregate, that derivation can be specified, for example, as
| SUM(col1+col2+col3).

| It is recommended that EVIs with INCLUDE aggregates only contain references to columns or
| column-specific derivations, for example, SUM(salary+bonus).

| In many environments, queries that contain derivations using constants convert those constants to
| parameter markers. This conversion allows a much higher degree of ODP reuse. However, it can be more
| difficult to match the parameter value to a literal in the index definition.

| The optimizer does attempt to match constants in the EVI with parameter markers or host variable values
| in the query. However, in some complex cases this support is limited and could result in the EVI not
| matching the query.

| Pre-summarized processing can also take advantage of EVIs with INCLUDE in a JOIN situation.

| For example, see the following aggregate query over the join of two tables.

| **EVI INCLUDE aggregate example**

```
| SELECT deptname, sum(salary)
| FROM DEPARTMENT, EMPLOYEE
| WHERE deptno=workdept
| GROUP BY deptname
```

| By providing an EVI with INCLUDE index, as follows, and with optimizer support to push down
| aggregates to the table level when possible, the resulting implementation takes advantage of the
| ready-made aggregates already supplied by EVI employeeSumByDept. The implementation never needs
| to touch or aggregate rows in the Employee table.

```
| CREATE ENCODED VECTOR INDEX employeeSumByDept ON employee(workdept)
| INCLUDE(sum(salary))
```

| Aggregate pushdown results in a rewrite with EVI INCLUDE implementation, conceptually like the
| following query.

```
| SELECT deptname, sum(sum(salary))
| FROM department,
|      (SELECT workdept, sum(salary) FROM employee group by workdept) employee_2
| WHERE deptno=workdept
```

| Instead of department joining to all the rows in the employee table, it now has the opportunity to join to
| the predetermined aggregates, the sum of salary by department number, in the EVI symbol table. This
| results in significant reduction in processing and IO.

Related concepts:

“How the EVI works” on page 202

EVI work in different ways for costing and implementation.

Related reference:

“Encoded vector index symbol table scan” on page 18

An encoded vector index symbol table scan operation is used to retrieve the entries from the symbol
table portion of the index.

Related information:

SQL INCLUDE statement

Optimizing grouping by eliminating grouping columns

All 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 does not help determine a unique group.

This processing allows the optimizer to consider more indexes to implement the query. It also reduces the number of columns that are 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 could implement the query. If a temporary index or hash is required then EMPNO is not used.

Note: Even though EMPNO can be removed from the list of grouping columns, the optimizer might use a permanent index that 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. Additional grouping columns are added only 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. It is a specialized version of ordered grouping that processes few records in each group rather than all records in each group.

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 algorithm improves 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 and the MIN or MAX operand is the next index key column 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 chooses to use the index IX1. The SLIC runtime code scans the index until it finds the first non-null value for SALARY. Assuming that SALARY is not null, the runtime code positions to the first index key and return that key value as the MAX of salary. No more index keys are processed.

Example 2, using SQL:

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

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

The query optimizer chooses to use Index IX2. The database manager positions to the first group for DEPT where JOB equals 'CLERK' and returns the SALARY. The code then skips 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. 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. 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 DEPARTMENT.DEPTNO, 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 always involve a temporary file before the group by processing. Therefore, these queries slow down.

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

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

Related information:

Add Physical File Trigger (ADDPFTRG) command

Grouping set optimization

The optimizer uses all the previously mentioned grouping optimizations for individual grouping sets specified in the query.

If multiple temporary result sets are needed to implement all the grouping sets, they can all be populated using one pass through the data. This one-pass population occurs even if different types of temporary result sets are used to implement various grouping sets.

A temporary result type called sorted distinct list is used specifically for ROLLUP implementations. This temporary result set is used to compute the aggregate rows: the grouping set that includes all expressions listed in the ROLLUP clause. Hash grouping is used internally to quickly find the current grouping value. The entries in the temporary result sets are also sorted. This sorting allows the aggregate results to be used to compute the super-aggregate rows in the rollup result set without creating additional temporary result sets.

ROLLUPs can also be implemented using a radix index over the columns in the rollup without creating a temporary result set.

| The optimizer can compute all the grouping sets in a given ROLLUP using at most one temporary result
| set. Therefore, it is advantageous for the optimizer to look for the rollup pattern in grouping set queries.

The optimizer tries to find the ROLLUP pattern in a list of individual grouping sets. For example, the following GROUP BY clause:

```
GROUP BY GROUPING SETS ((A, B, C), (B, D), (A, B), (A), ())
```

is rewritten to:

```
GROUP BY GROUPING SETS ((ROLLUP(A, B, C)), (B, D))
```

This rewrite allows the query to be implemented using at most two temporary results sets rather than 4.

Queries containing a CUBE clause is broken down into a union of ROLLUPs and grouping sets. For example:

```
CUBE(A, B, C)
```

is equivalent to:

```
(ROLLUP(A, B, C)), (ROLLUP'(B, C)), (ROLLUP'(C, A))
```

The ROLLUP' notation is an internal representation of a ROLLUP operation that does not include a grand total row in its result set. So, ROLLUP'(B, C) is equivalent to GROUP BY GROUPING SETS ((B,C), (B)). This CUBE rewrite implements at most three temporary result sets, rather than the 8 that might be needed had the query not been rewritten.

Ordering optimization

This section describes how DB2 for i 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 the ordering columns as contiguous leftmost 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 the ordered results, or if an index exists that matches the ordering columns. When the ordering is implemented with an index, and a permanent index does not 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 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 does not help determine in the order.

The optimizer can now consider more indexes as it implements the query. The number of columns that are added as key columns to a temporary index is also reduced. 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 logic 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, derived tables (nested table expressions or NTEs), and common table expressions (CTEs) are implemented by the query optimizer using one of two methods.

These methods are:

- The optimizer combines the query select statement with the select statement of the view.

- 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 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.

See the following examples:

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

Using SQL:

```
SELECT LASTNAME, FIRSTNAME, SALARY
FROM D21EMPL
WHERE JOB='CLERK'
```

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

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

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

Note: The new composite query that the query optimizer generates is not visible to users. Only the original query against the view is 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 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. For SQE, view materialization is optional. The following types of queries require view materialization:

- The outermost view select contains grouping, the query contains grouping, and refers to a column derived from a column function in the view 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. Occurs when a view references a view, and a view composite cannot be created because of one of the previous listed reasons. Does not apply to nested table expressions and common table expressions.
- The Common table expression (CTE) is referenced more than once in the query FROM clause. Also, the CTE SELECT clause references a MODIFIES or EXTERNAL ACTION UDF.

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

See the following 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 is 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 process is not 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 AND
A.WORKDEPT LIKE 'D%' AND AVGSAL>10000
```

The isolatable selection from the query is added to the 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 performance enhancement 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 looks for any applicable MQTs. The optimizer can implement the query using a given MQT, provided it is a faster implementation choice.

Materialized Query Tables are created using the `SQL CREATE TABLE` statement. Alternatively, the `ALTER TABLE` statement could 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 could also be maintained by the user using `INSERT`, `UPDATE`, and `DELETE` statements.

Related information:

Create Table statement

MQT supported function

Although an 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.

The supported function in the MQT query by the MQT matching algorithm includes:

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

There is limited support in the MQT matching algorithm for the following:

- Scalar subselects
- User Defined Functions (UDFs) and user-defined table functions
- Recursive Common Table Expressions (RCTE)
- The following scalar functions:
 - ATAN2
 - DAYNAME
 - DBPARTITIONNAME
 - DECRYPT_BIT
 - DECRYPT_BINARY
 - DECRYPT_CHAR
 - DECRYPT_DB
 - DIFFERENCE
 - DLVALUE
 - DLURLPATH
 - DLURLPATHONLY
 - DLURLSEVER
 - DLURLSCHEME
 - DLURLCOMPLETE
 - ENCRYPT_AES
 - ENCRYPT_RC2
 - ENCRYPT_TDES
 - GENERATE_UNIQUE
 - GETHINT
 - IDENTITY_VAL_LOCAL
 - INSERT
 - MONTHNAME
 - MONTHS_BETWEEN
 - NEXT_DAY

- RAND
- RAISE_ERROR
- REPEAT
- REPLACE
- ROUND_TIMESTAMP
- SOUNDEX
- TIMESTAMP_FORMAT
- TIMESTAMPDIFF
- TRUNC_TIMESTAMP
- VARCHAR_FORMAT
- WEEK_ISO

It is recommended that the MQT only contain references to columns and column functions. In many environments, queries that contain constants have the constants converted to parameter markers. This conversion allows a much higher degree of ODP reuse. The MQT matching algorithm attempts to match constants in the MQT with parameter markers or host variable values in the query. However, in some complex cases this support is limited and could result in the MQT not matching the query.

Related concepts:

“Query dispatcher” on page 4

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. It cannot be bypassed.

Related reference:

“Details on the MQT matching algorithm” on page 80

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

Using MQTs during query optimization

Before using MQTs, you need to consider your environment attributes.

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

- The query must specify ALWCPYDTA(*OPTIMIZE) or INSENSITIVE cursor.
- The query must not be a SENSITIVE cursor.
- The table to be replaced with an 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 an 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 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.
- | • The field procedure encoded comparison (QAQQINI FIELDPROC_ENCODED_COMPARISON option)
- | level of the MQT must be greater than or equal to the query specified field procedure encoded
- | comparison level.

MQT examples

The following are examples of using MQTs.

Example 1

The first example is a query that returns information about employees whose job is DESIGNER. The original query:

```
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:

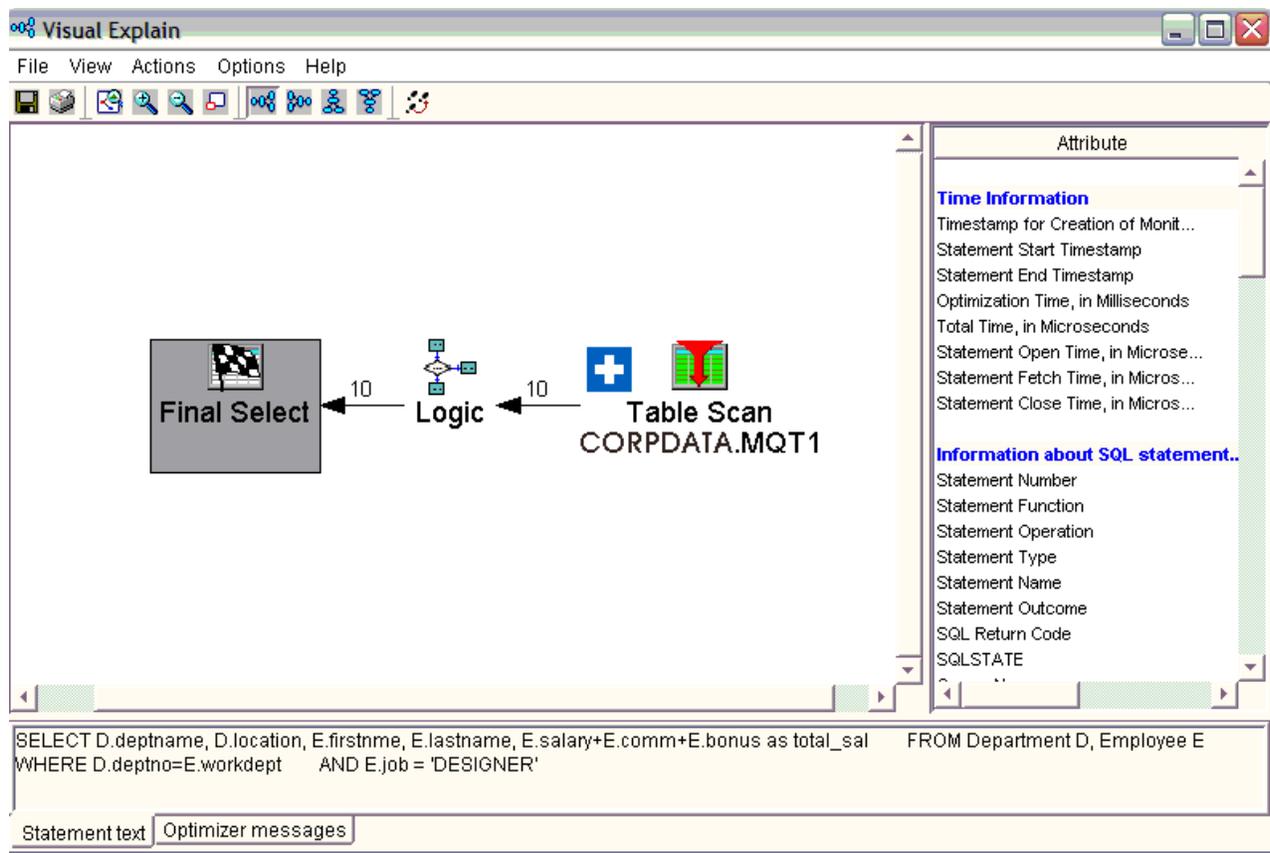
```
CREATE TABLE MQT1
  AS (SELECT D.deptname, D.location, E.firstnme, E.lastname, E.salary, E.comm, E.bonus, E.job
FROM Department D, Employee E
WHERE D.deptno=E.workdept)
DATA INITIALLY IMMEDIATE REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

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 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. The result expression, M.salary+M.comm+M.bonus, is calculated. 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:

```
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:

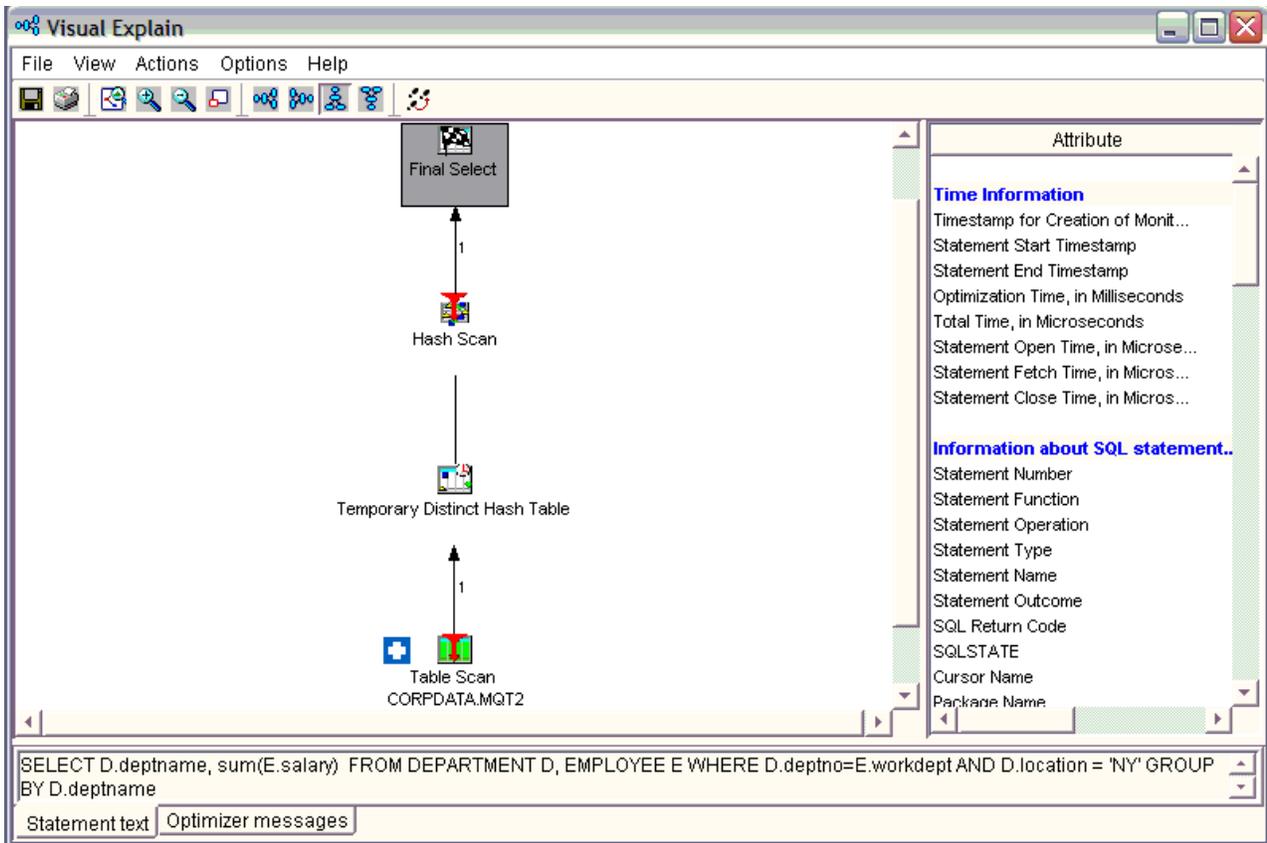
```
CREATE TABLE MQT2
AS (SELECT D.deptname, D.location, sum(E.salary) as sum_sal
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.deptno=E.workdept
GROUP BY D.Deptname, D.location)
DATA INITIALLY IMMEDIATE REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

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 and the query specify the same tables, then the MQT can potentially be used and matching continues. If the MQT references tables not referenced in the query, then the unreferenced table is examined to determine if it is a parent table in referential integrity constraint. If the foreign key is non-nullable and the two tables are joined using a primary key or foreign key equal predicate, then the MQT can still be potentially used.

Example 3

The MQT contains fewer 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 preceding query:

```
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:

```

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 could 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.

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 preceding query:

```

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. The MQT matching algorithm matches the parameter marker and to the constant 'NY' in the predicate D.Location=?. It verifies that the values of the parameter marker are the same as the constant in the MQT; therefore the MQT can be used.

The MQT matching algorithm also attempts to match where the predicates between the MQT and the query are not the same. For example, if the MQT has a predicate SALARY > 50000, and the query has the predicate SALARY > 70000, the MQT contains the rows necessary to run the query. The MQT is used in the query, but the predicate SALARY > 70000 is left as selection in the query, so SALARY must be a column of the MQT.

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 preceding query:

```

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.

Example 6

```

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

```

Create an MQT based on the preceding query:

```

| CREATE TABLE MQT6(workdept, sumSalary)
| AS (SELECT workdept, sum(salary)
|     FROM EMPLOYEE
|     GROUP BY workdept )
| DATA INITIALLY IMMEDIATE REFRESH DEFERRED
| ENABLE QUERY OPTIMIZATION
| MAINTAINED BY USER

```

In this example, the SUM(salary) aggregation is pushed down through the join to the EMPLOYEE table, allowing for a match and substitution of MQT6. A regrouping to (sum(sum(salary))) is defined at the top of the query to compensate for the grouping pushdown.

Instead of department joining to all the rows in the employee table, it now has the opportunity to join to the predetermined aggregates in MQT6. This type of MQT substitution can result in significant reduction of processing and IO.

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 group by columns that are a superset of query group by columns, the query can be rewritten to do regrouping. This regrouping reaggregates the groups of the MQT into the groups required by the query. When regrouping is required, the column functions need to be recomputed. The following table shows the supported regroup expressions.

The regrouping expression/aggregation rules are:

Table 28. 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 | Not applicable | 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) |

Table 28. Expression/aggregation rules for MQTs (continued)

| Query | MQT | Final query |
|--|--|------------------|
| 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 | Not applicable | 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 | Not applicable | 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 | Not applicable | MIN(C2) |
| GROUPING(C1) | GROUPING(C1) as grp | grp |
| GROUPING(C2) where C2 is from a table not in the MQT | Not applicable | GROUPING(C2) |

MQT matching does not support ARRAY_AGG, XMLAGG, and XMLGROUP grouping functions. AVG, STDDEV, STDDEV_SAMP, VARIANCE_SAMP 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 FETCH FIRST N ROWS is specified in the MQT, then FETCH FIRST N ROWS must also be specified in the query. Also, 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 an 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 is part of the rewritten query.

Related reference:

“MQT supported function” on page 76

Although an 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.

Determining unnecessary MQTs

You can easily determine which MQTs are being used for query optimization. However, you can now easily find all MQTs and retrieve statistics on MQT usage as a result of System i Navigator and IBM i functionality.

To assist you in tuning your performance, this function produces statistics on MQT usage in a query. To access through the System i Navigator, navigate to: **Database > Schemas > Tables**. Right-click your table and select **Show Materialized Query Tables**. You can also view MQT usage information by right-click on

Tables or Views folder and select **Show Materialized Query Tables**. This action displays usage information for MQTs created over all the tables or view in that schema.

Note: You can also view the statistics through an application programming interface (API).

In addition to all existing attributes of an MQT, two fields can help you determine unnecessary MQTs.

These fields are:

Last Query Use

States the timestamp when the MQT was last used by the optimizer to replace user specified tables in a query.

Query Use Count

Lists the number of instances the MQT was used by the optimizer to replace user specified tables in a query.

The fields start and stop counting based on your situation, or the actions you are currently performing on your system. A save and restore procedure does not reset the statistics counter if the MQT is restored over an existing MQT. If an MQT is restored that does not exist on the system, the statistics are reset.

Related information:

Retrieve member description (QUSRMBRD) command

Summary of MQT query recommendations

Follow these recommendations when using MQT queries.

- Do not include local selection or constants in the MQT because that limits the number of user-specified queries where the optimizer can use the MQT.
- 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 where the query optimizer can use the MQT. Not recommended.
- If the MQT is created with DATA INITIALLY DEFERRED, consider specifying DISABLE QUERY OPTIMIZATION to prevent the optimizer from using the MQT until it has been populated. When the MQT is populated and ready for use, the ALTER TABLE statement with ENABLE QUERY OPTIMIZATION enables the MQT.

| In addition, consider using a sparse index or EVI INCLUDE additional MQT aggregates rather than an MQT if
| you are concerned with stale data.

MQT tables need to be optimized just like non-MQT tables. It is recommended that indexes are created over the MQT columns 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 MQTs have been enabled through the QAQQINI file, and an MQT exists over at least one of the tables in the query, there is 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.

Related concepts:

“How the EVI works” on page 202

EVI works in different ways for costing and implementation.

Related reference:

“Sparse index optimization” on page 195

An SQL sparse index is like a select/omit access path. Both the sparse index and the select/omit logical file contain only keys that meet the selection specified. For a sparse index, the selection is specified with a WHERE clause. For a select/omit logical file, the selection is specified in the DDS using the COMP

operation.

Recursive query optimization

Certain applications and data are recursive by nature. Examples of such applications are a bill-of-material, reservation, trip planner, or networking planning system. Data in one results row has a natural relationship (call it a parent, child relationship) with data in another row or rows. The kinds of recursion implemented in these systems can be performed by using SQL Stored Procedures and temporary results tables. However, the use of a recursive query to facilitate the access of this hierarchical data can lead to a more elegant and better performing application.

Recursive queries can be implemented by defining either a Recursive Common Table Expression (RCTE) or a Recursive View.

Recursive query example

A recursive query is one that is defined by a Union All with an initialization fullselect that seeds the recursion. The iterative fullselect contains a direct reference to itself in the FROM clause.

There are additional restrictions as to what can be specified in the definition of a recursive query. Those restrictions can be found in SQL Programming topic.

Functions like grouping, aggregation, or distinct require a materialization of all the qualifying records before performing the function. These functions cannot be allowed within the iterative fullselect itself. The functions must be placed in the main query, allowing the recursion to complete.

The following is an example of a recursive query over a table called flights, that contains information about departure and arrival cities. The query returns all the flight destinations available by recursion from the two specified cities (New York and Chicago). It also returns the number of connections and total cost to arrive at that final destination.

This example uses the recursion process to also accumulate information like the running cost and number of connections. Four values are put in the queue entry. These values are:

- The originating departure city (either Chicago or New York) because it remains fixed from the start of the recursion
- The arrival city which is used for subsequent joins
- The incrementing connection count
- The accumulating total cost to reach each destination

Typically the data needed for the queue entry is less than the full record (sometimes much less) although that is not the case for this example.

```
CREATE TABLE flights
(
  departure CHAR (10) NOT NULL WITH DEFAULT,
  arrival CHAR (10) NOT NULL WITH DEFAULT,
  carrier CHAR (15) NOT NULL WITH DEFAULT,
  flight_num CHAR (5) NOT NULL WITH DEFAULT,
  ticket INT NOT NULL WITH DEFAULT)

WITH destinations (departure, arrival, connects, cost ) AS
(
  SELECT f.departure,f.arrival, 0, ticket
  FROM flights f
  WHERE f.departure = 'Chicago' OR
        f.departure = 'New York'
  UNION ALL
  SELECT
    r.departure, b.arrival, r.connects + 1,
    r.cost + b.ticket
```

```

FROM destinations r, flights b
WHERE r.arrival = b.departure
)
SELECT DISTINCT departure, arrival, connects, cost
FROM destinations

```

The following is the initialization fullselect of the preceding query. It seeds the rows that start the recursion process. It provides the initial destinations (arrival cities) that are a direct flight from Chicago or New York.

```

SELECT f.departure,f.arrival, 0, ticket
FROM flights f
WHERE f.departure='Chicago' OR
      f.departure='New York'

```

The following is the iterative fullselect of the preceding query. It contains a single reference in the FROM clause to the destination recursive common table expression. It also sources further recursive joins to the same flights table. The arrival values of the parent row (initially direct flights from New York or Chicago) are joined with the departure value of the subsequent child rows. It is important to identify the correct parent/child relationship on the recursive join predicate or infinite recursion can occur. Other local predicates can also be used to limit the recursion. For example, for a limit of at most 3 connecting flights, a local predicate using the accumulating connection count, `r.connects<=3`, can be specified.

```

SELECT
  r.departure, b.arrival, r.connects + 1 ,
  r.cost + b.ticket
FROM destinations r, flights b
WHERE r.arrival=b.departure

```

The main query is the query that references the recursive common table expression or view. It is in the main query where requests like grouping, ordering, and distinct are specified.

```

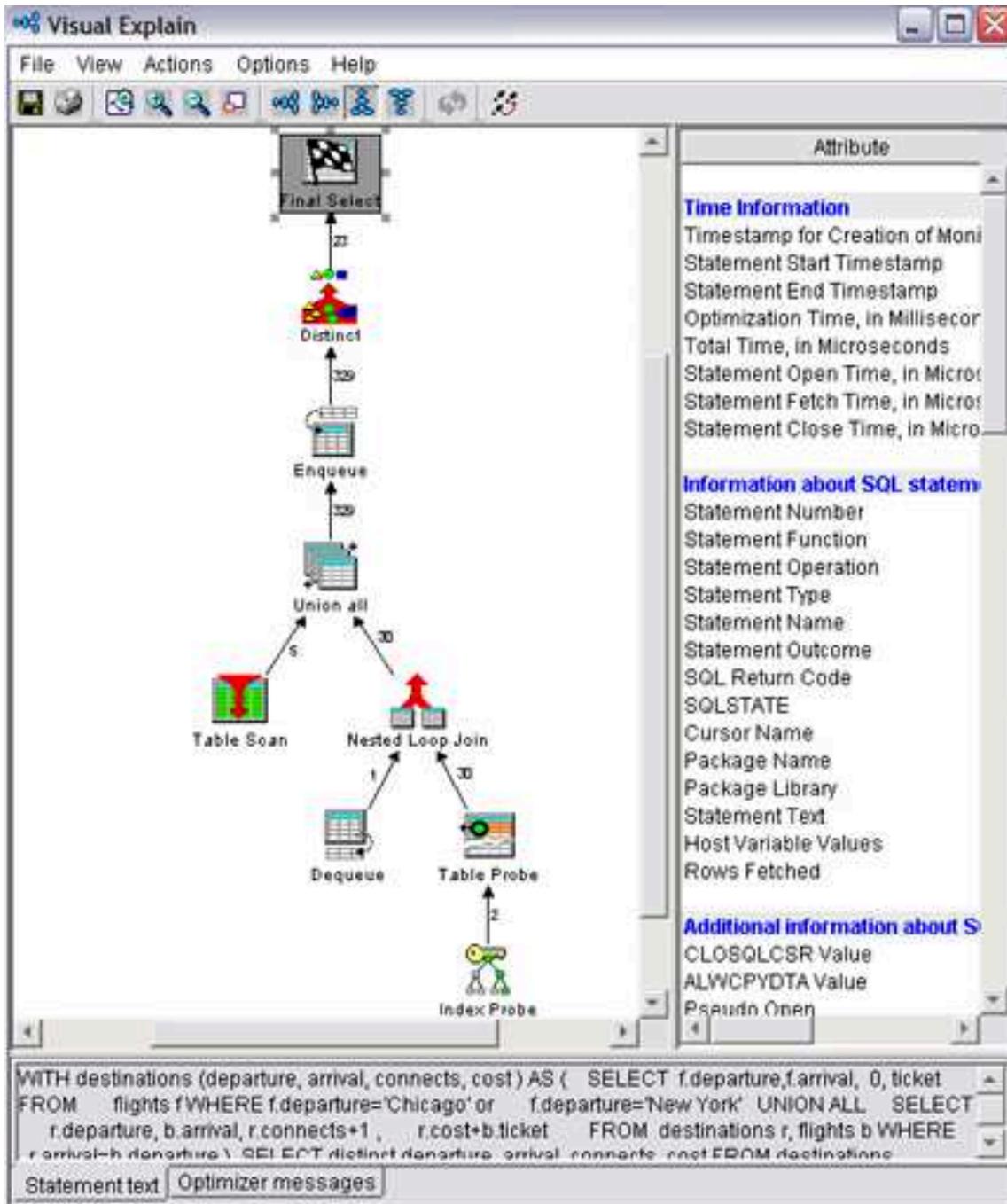
SELECT DISTINCT departure, arrival, connects, cost
FROM destinations

```

Implementation considerations

To implement a source for the recursion, a new temporary data object is provided called a queue. As rows meet the requirements of either the initialization fullselect or the iterative fullselect, they are pulled up through the union all. Values necessary to feed the continuing recursion process are captured and placed in an entry on the queue: an enqueue operation.

At query runtime, the queue data source then takes the place of the recursive reference in the common table expression or view. The iterative fullselect processing ends when the queue is exhausted of entries or a fetch N rows limitation has been met. The recursive queue feeds the recursion process and holds transient data. The join between dequeuing of these queue entries and the rest of the fullselect tables is always a constrained join, with the queue on the left.



Multiple initialization and iterative fullselects

The use of multiple initialization and iterative fullselects specified in the recursive query definition allows for a multitude of data sources and separate selection requirements to feed the recursion process.

For example, the following query allows for final destinations accessible from Chicago by both flight and train travel.

```
WITH destinations (departure, arrival, connects, cost ) AS
(
  SELECT f.departure, f.arrival, 0 , ticket
  FROM flights f
  WHERE f.departure='Chicago'
  UNION ALL
  SELECT t.departure, t.arrival, 0 , ticket
```

```

FROM trains t
WHERE t.departure='Chicago'
UNION ALL
SELECT
  r.departure,b.arrival, r.connects + 1 ,
  r.cost + b.ticket
FROM destinations r, flights b
WHERE r.arrival=b.departure
UNION ALL
SELECT
  r.departure,b.arrival, r.connects+1 ,
  r.cost+b.ticket
FROM destinations r, trains b
WHERE r.arrival=b.departure)

SELECT departure, arrival, connects,cost
FROM destinations;

```

All rows coming out of the RCTE/View are part of the recursion process and need to be fed back in. When there are multiple fullselects referencing the common table expression, the query is rewritten by the optimizer to process all non-recursive initialization fullselects first. Then, using a single queue feed, those same rows and all other row results are sent equally to the remaining iterative fullselects. No matter how you order the initialization and iterative fullselects in the definition of the RCTE/view, the initialization fullselects run first. The iterative fullselects share equal access to the contents of the queue.

The screenshot shows the Visual Explain tool interface. The main window displays a query execution plan with the following components and flow:

- Final Select** (top) receives 336 rows from an **Enqueue** operation.
- Enqueue** receives 336 rows from a **Union all** operation.
- Union all** receives 2 rows from a **Table Scan** and 1 row from another **Table Scan**.
- Union all** also receives 333 rows from a **Nested Loop Join**.
- Nested Loop Join** receives 1 row from a **Dequeue** operation and 333 rows from another **Union all**.
- Union all** (under Nested Loop Join) receives 30 rows from a **Table Probe** and 3 rows from another **Table Probe**.
- Table Probe** operations receive 2 and 1 rows respectively from the bottom of the plan.

On the right side, the performance statistics are displayed:

| Attribute | Value |
|---|-------|
| Time Information | |
| Timestamp for Creation of Monit... | 20 |
| Statement Start Timestamp | 20 |
| Statement End Timestamp | 20 |
| Optimization Time, in Milliseconds | 29 |
| Total Time, in Microseconds | 15 |
| Statement Open Time, in Micros... | 15 |
| Statement Fetch Time, in Micros... | No |
| Statement Close Time, in Micros... | No |
| Information about SQL stateme... | |
| Statement Number | 40 |
| Statement Function | Se |
| Statement Operation | Op |
| Statement Type | Dy |
| Statement Name | ST |
| Statement Outcome | Su |
| SQL Return Code | 0 |
| SQLSTATE | 00 |
| Cursor Name | CF |
| Package Name | |
| Package Library | |
| Statement Text | WI |
| Host Variable Values | 0, 1 |
| Rows Fetched | No |
| Additional information about SQ... | |
| CLOSECURSOR Value | |

At the bottom, the SQL statement text is visible:

```
WITH destinations (departure, arrival, connects, cost) AS (
  SELECT f.departure, f.arrival, 0, ticket FROM flights f
  WHERE f.departure='Chicago' UNION ALL
  SELECT t.departure, t.arrival, 0, ticket FROM trains t
  WHERE t.departure='Chicago' UNION ALL
  SELECT r.departure, b.arrival, r.connects+1, r.cost+b.ticket
  FROM destinations r, flights b
  WHERE r.arrival=b.departure UNION ALL
  SELECT r.departure, b.arrival, r.connects+1,
  r.cost+b.ticket FROM destinations r, trains b
  WHERE r.arrival=b.departure)
SELECT departure, arrival,
connects, cost FROM destinations
```

Predicate pushing

When processing most queries with non-recursive common table expressions or views, local predicates specified on the main query are pushed down so fewer records need to be materialized. Pushing local predicates from the main query into the defined recursive part of the query (through the Union ALL), however, could considerably alter the process of recursion itself. So as a rule, the Union All specified in a recursive query is currently a predicate fence. Predicates are not pushed down or up, through this fence.

The following is an example of how pushing a predicate in to the recursion limits the recursive results and alter the intent of the query.

The intent of the query is to find all destinations accessible from 'Chicago', not including the final destination of 'Dallas'. Pushing the "arrival<>'Dallas'" predicate into the recursive query alters the output of the intended results. It prevents the output of final destinations where 'Dallas' was an intermediate stop.

```
WITH destinations (departure, arrival, connects, cost ) AS
(
  SELECT f.departure,f.arrival, 0, ticket
  FROM flights f
```

```

WHERE f.departure='Chicago'
UNION ALL
SELECT
  r.departure, b.arrival, r.connects + 1 ,
  r.cost + b.ticket
FROM destinations r, flights b
WHERE r.arrival=b.departure
)
SELECT departure, arrival, connects, cost
FROM destinations
WHERE arrival != 'Dallas'

```

Conversely, the following is an example where a local predicate applied to all the recursive results is a good predicate to put in the body of the recursive definition because it could greatly decrease the number of rows materialized from the RCTE/View. The better query request here is to specify the `r.connects <=3` local predicate with in the RCTE definition, in the iterative fullselect.

```

WITH destinations (departure, arrival, connects, cost ) AS
(
  SELECT f.departure,f.arrival, 0, ticket
  FROM flights f
  WHERE f.departure='Chicago' OR
        f.departure='New York'
  UNION ALL
  SELECT
    r.departure, b.arrival, r.connects + 1 ,
    r.cost + b.ticket
  FROM destinations r, flights b
  WHERE r.arrival=b.departure
)
SELECT departure, arrival, connects, cost
FROM destinations
WHERE r.connects<=3

```

Placement of local predicates is key in recursive queries. They can incorrectly alter the recursive results if pushed into a recursive definition. Or they can cause unnecessary rows to be materialized and then rejected, when a local predicate could legitimately help limit the recursion.

Specifying SEARCH consideration

Certain applications dealing with hierarchical, recursive data could have a requirement in how data is processed: by depth or by breadth.

Using a queuing (First In First Out) mechanism to track the recursive join key values implies the results are retrieved in breadth first order. Breadth first means retrieving all the direct children of a parent row before retrieving any of the grandchildren of that same row. This retrieval is an implementation distinction, however, and not a guarantee.

Applications might want to guarantee how the data is retrieved. Some applications might want to retrieve the hierarchical data in depth first order. Depth first means that all the descendents of each immediate child row are retrieved before the descendents of the next child are retrieved.

The SQL architecture allows for the guaranteed specification of how the application retrieves the resulting data by the use of the `SEARCH DEPTH FIRST` or `BREADTH FIRST` keyword. When this option is specified, name the recursive join value, identify a set sequence column, and provide the sequence column in an outer `ORDER BY` clause. The results are output in depth or breadth first order. Note this ordering is ultimately a relationship sort and not a value-based sort.

Here is the preceding example output in depth first order.

```

WITH destinations (departure, arrival, connects, cost ) AS
(
  SELECT f.departure, f.arrival, 0 , ticket

```

```

FROM flights f
WHERE f.departure='Chicago' OR f.departure='New York'
UNION ALL
SELECT
  r.departure,b.arrival, r.connects+1 ,
  r.cost+b.ticket
FROM destinations r, flights b
WHERE r.arrival=b.departure)

```

SEARCH DEPTH FIRST BY arrival **SET** depth_sequence

```

SELECT *
FROM destinations
ORDER BY depth_sequence

```

If the ORDER BY clause is not specified in the main query, the sequencing option is ignored. To facilitate the correct sort there is additional information put on the queue entry during recursion. With BREADTH FIRST, it is the recursion level number and the immediate ancestor join value, so sibling rows can be sorted together. A depth first search is a little more data intensive. With DEPTH FIRST, the query engine needs to represent the entire ancestry of join values leading up to the current row and put that information in a queue entry. Also, because these sort values are not coming from an external data source, the sort implementation is always a temporary sorted list (no indexes possible).

Do not use the SEARCH option if you do not need your data materialized in a depth or breadth first manner. There is additional CPU and memory overhead to manage the sequencing information.

Specifying CYCLE considerations

Recognizing that data in the tables used in a recursive query might be cyclic in nature is important to preventing infinite loops.

The SQL architecture allows for the optional checking for cyclic data and discontinuing the repeating cycles at that point. This additional checking is done by the use of the CYCLE option. The correct join recursion value must be specified on the CYCLE request and a cyclic indicator must be specified. The cyclic indicator could be optionally output in the main query and can be used to help determine and correct errant cyclic data.

```

WITH destinations (departure, arrival, connects, cost , itinerary) AS
(
  SELECT f.departure, f.arrival, 1 , ticket, CAST(f.departure||f.arrival AS VARCHAR(2000))
  FROM flights f
  WHERE f.departure='New York'
  UNION ALL
  SELECT r.departure,b.arrival, r.connects+1 ,
    r.cost+b.ticket, cast(r.itinerary||b.arrival AS varchar(2000))
  FROM destinations r, flights b
  WHERE r.arrival = b.departure)
CYCLE arrival SET cyclic TO '1' DEFAULT '0' USING Cycle_Path

SELECT departure, arrival, itinerary, cyclic
FROM destinations

```

When a cycle is determined to be repeating, the output of that cyclic sequence of rows is stopped. To check for a 'repeated' value however, the query engine needs to represent the entire ancestry of the join values leading up to the current row in order to look for the repeating join value. This ancestral history is information that is appended to with each recursive cycle and put in a field on the queue entry.

To implement this history field, the query engine uses a compressed representation of the recursion values on the ancestry chain. The query engine can then do a fixed length, quicker scan through the accumulating ancestry to determine if the value has been seen before. This compressed representation is determined by the use of a distinct node in the query tree.

Do not use the CYCLE option unless you know your data is cyclic, or you want to use it specifically to help find the cycles for correction or verification purposes. There is additional CPU and memory overhead to manage and check for repeating cycles before a given row is materialized.

The screenshot shows the Visual Explain window with a query execution plan on the left and a list of attributes on the right. The execution plan starts with an Index Probe, followed by a Table Probe, then a Union All operation. The Union All branches into a Deq table and a Nested Loop Join. The Nested Loop Join involves a Deq table and another Table Probe. Above the Union All is a Distinct operation, followed by an Eq table, a Temporary List, a ListScan, and finally a Final Select operation.

The attributes listed on the right are:

- Time Information**
 - Timestamp for Creation of Monit...
 - Statement Start Timestamp
 - Statement End Timestamp
 - Optimization Time, in Milliseconds
 - Total Time, in Microseconds
 - Statement Open Time, in Micros...
 - Statement Fetch Time, in Micros...
 - Statement Close Time, in Micros...
- Information about SQL stateme...**
 - Statement Number
 - Statement Function
 - Statement Operation
 - Statement Type
 - Statement Name
 - Statement Outcome
 - SQL Return Code
 - SQLSTATE
 - Cursor Name
 - Package Name
 - Package Library
 - Statement Text
 - Host Variable Values
 - Rows Fetched
- Additional information about SQ...**
 - CLOSESQLCSR Value
 - ALWCOPYDTA Value
 - Pseudo Open
 - Pseudo Close

At the bottom, the statement text is visible:

```
WITH destinations (departure, arrival, connects, cost, itinerary) AS ( SELECT f.departure, f.arrival, 1, ticket,
cast(f.departure||f.arrival as varchar(2000))FROM flights f WHERE f.departure='New York' UNION ALL
SELECT ... )
```

SMP and recursive queries

Recursive queries can benefit as much from symmetric multiprocessing (SMP) as do other queries on the system.

Recursive queries and parallelism, however, present some unique requirements. The initialization fullselect of a recursive query is the fullselect that seeds the initial values of the recursion. It is likely to produce only a small fraction of the ultimate results that cycle through the recursion process. The query optimizer does not want each of the threads running in parallel to have a unique queue object that feeds only itself. This results in some threads having way too much work to do and others threads quickly depleting their work.

The best way to handle this work is to have all the threads share the same queue. This method allows a thread to enqueue a new recursive key value just as a waiting thread is there to dequeue that request. A shared queue allows all threads to actively contribute to the overall depletion of the queue entries until no thread is able to contribute more results.

Having multiple threads share the same queue, however, requires some management by the Query runtime so that threads do not prematurely end. Some buffering of the initial seed values might be necessary. This buffering is illustrated in the following query, where there are two fullselects that seed the recursion. A buffer is provided so that no thread hits a dequeue state and terminates before the query has seeded enough recursive values to get things going.

The following Visual Explain diagram shows the plan for the following query run with CHGQRYA DEGREE(*NBRTASKS 4). It shows how the results of the multiple initialization fullselects are buffered up. The multiple threads, illustrated by the multiple arrow lines, are acting on the enqueue and dequeue request nodes. As with all SMP queries, the multiple threads, in this case 4, put their results into a Temporary List object which becomes the output for the main query.

```
cl:chgqrya degree(*nbrtasks 4);
```

```
WITH destinations (departure, arrival, connects, cost )AS
(
  SELECT f.departure, f.arrival, 0 , ticket
  FROM flights f WHERE f.departure='Chicago'
  UNION ALL
  SELECT t.departure, t.arrival, 0 , ticket
  FROM trains t WHERE t.departure='Chicago'
  UNION ALL
  SELECT
    r.departure,b.arrival, r.connects+1 ,
    r.cost+b.ticket
  FROM destinations r, flights b
  WHERE r.arrival=b.departure
  UNION ALL
  SELECT
    r.departure,b.arrival, r.connects+1 ,
    r.cost+b.ticket
  FROM destinations r, trains b
  WHERE r.arrival=b.departure)
SELECT departure, arrival, connects,cost
FROM destinations;
```

Visual Explain

File View Actions Options Help

| Attribute | Value |
|---|-------|
| Time Information | |
| Timestamp for Creation of Monit... | 2006 |
| Statement Start Timestamp | 2006 |
| Statement End Timestamp | 2006 |
| Optimization Time, in Milliseconds | 111 |
| Total Time, in Microseconds | 2497 |
| Statement Open Time, in Micros... | 2497 |
| Statement Fetch Time, in Micros... | Not A |
| Statement Close Time, in Micros... | Not A |
| Information about SQL stateme... | |
| Statement Number | 17 |
| Statement Function | Sele |
| Statement Operation | Oper |
| Statement Type | Dyna |
| Statement Name | STM |
| Statement Outcome | Succ |
| SQL Return Code | 0 |
| SQLSTATE | 000C |
| Cursor Name | CRS |
| Package Name | |
| Package Library | |
| Statement Text | WITH |
| Host Variable Values | 0, C, |
| Rows Fetched | Not A |
| Additional information about SQ... | |
| CLOSECURSOR Value | |
| ALWCOPYDATA Value | Any T |
| Pseudo Open | No |
| Pseudo Close | No |
| Hard Close Reason Code | Not A |
| ODP Implementation | Reus |
| Dynamic Replan Reason Code | Acce |
| Timestamp When Plan Was Cre... | 0001 |
| Data Conversion Reason Code | Not a |
| Blocking Enabled | ALW |
| Delay Prep | Yes |
| Statement is Explainable | Yes |
| Naming Convention | SQL |
| Type of Dynamic Processing | Loca |
| SQL Path | "QSY |
| Information common to most m... | |
| System Name | Y046 |
| Job Name | QZD. |
| Job User | QUS |
| Job Number | 1889 |

WITH destinations (departure, arrival, connects, cost)AS (SELECT f.departure, f.arrival, 0 , ticket FROM

Statement text Optimizer messages

Adaptive Query Processing

Adaptive Query Processing analyzes actual query run time statistics and uses that information for subsequent optimizations.

With rapidly increasing amounts of data, the price of miscalculating complex plans can result in dramatic performance problems. These problems might be measured in minutes or hours instead of seconds or minutes. Traditionally, optimizer architecture has attempted to overcome potential plan problems in several ways. The most common technique is to increase the amount of time spent optimizing a query, searching for safe alternatives. While additional time reduces the likelihood of a failed plan, it does not fundamentally avoid the problem.

The DB2 optimizer relies on statistical estimates to optimize a query. These estimates can be inaccurate for a number of reasons. The reasons include a lack of statistical metadata for the query tables, complex join conditions, skewed or rapidly changing data within the tables, and others.

The SQE query engine uses a technique called Adaptive Query Processing (AQP). AQP analyzes actual query run time statistics and uses that information to correct previous estimates. These updated estimates can provide better information for subsequent optimizations.

Related reference:

“Adaptive Query Processing in Visual Explain” on page 144
You can use Visual Explain to request a new plan.

How AQP works

There are three main parts to AQP support.

- **Global Statistics Cache (GSC):** The “Global Statistics Cache” on page 6 is a system-side repository of statistical information gathered from actual query runs. When the SQE query engine observes a discrepancy between record count estimates and actual observed values, an entry might be made in the GSC. This entry provides the optimizer with more accurate statistical information for subsequent optimizations.
- **AQP Request Support:** This support runs after a query completes. The processing is done in a system task so it does not affect the performance of user applications. Estimated record counts are compared to the actual values. If significant discrepancies are noted, the AQP Request Support stores the observed statistic in the GSC. The AQP Request Support might also make specific recommendations for improving the query plan the next time the query runs.
- **AQP Handler:** The AQP Handler runs in a thread parallel to a running query and observes its progress. The AQP handler wakes up after a query runs for at least 2 seconds without returning any rows. Its job is to analyze the actual statistics from the partial query run, diagnose, and possibly recover from join order problems. These join order problems are due to inaccurate statistical estimates. The query can be reoptimized using partial observed statistics or specific join order recommendations or both. If this optimization results in a new plan, the old plan is terminated and the query restarted with the new plan, provided the query has not returned any results.

AQP looks for an unexpected starvation join condition when it analyzes join performance. Starvation join is a condition where a table late in the join order eliminates many records from the result set. In general, the query would perform better if the table that eliminates the large number of rows is first in the join order. When AQP identifies a table that causes an unexpected starvation join condition, the table is noted as the ‘forced primary table’. The forced primary table is saved for a subsequent optimization of the query.

That subsequent optimization with the forced primary recommendation can be used in two ways:

- The forced primary table is placed first in the join order, overriding the join order implied by the statistical estimates. The rest of the join order is defined using existing techniques.
- The forced primary table can be used for LPG preselection against a large fact table in the join.

| **Related reference:**

| “Adaptive Query Processing in Visual Explain” on page 144
| You can use Visual Explain to request a new plan.

| **AQP example**

| Here is an example query with an explanation of how AQP could work.

```
| SELECT * from t1, t2, t3, t4  
| WHERE t1.c1=t2.c1 AND t1.c2=t3.c2  
| AND t1.c3 = CURRENT DATE - t4.c3  
| AND t1.c5 < 50 AND t2.c6 > 40  
| AND t3.c7 < 100 AND t4.c8 - t4.c9 < 5
```

| The WHERE clause of the preceding query contains a predicate, `t1.c3 = CURRENT DATE - t4.c3`, that is difficult to estimate. The estimation difficulty is due to the derivation applied to column `t4.c3` and the derivation involving columns `t4.c8` and `t4.c9`. For the purposes of this example, the predicate `t1.c3 = CURRENT DATE - t4.c3` actually eliminates all or nearly all records in the join.

| Due to characteristics of the columns involved in that predicate, the statistical estimate has many rows returned from the join. The optimizer selects join order `t1, t3, t2, t4` based on the following record count estimates.

- | • Join `t1` to `t3` produces 33,000,000 rows.
- | • Join `t1, t3` result to `t2` produces 1,300,000 rows.
- | • Join `t1, t3, t2` result to `t4` (final result set) produces 5 million rows.

| The join order is reasonable assuming that the final result set actually produces 5 million rows, but the estimate is incorrect. The query performs poorly since tables `t1, t3, t2` are joined first, producing 1,300,000 rows. These rows are all rejected by table `t4` and the `t1.c3 = CURRENT DATE - t4.c3` predicate (join starvation).

| AQP identifies `t4` as the forced primary table. The optimizer would choose `t1` as the second table in the join order since there are no join conditions between `t4` and `t2` or `t3`. Since the join condition between tables `t4` and `t1` selects few rows, this plan is likely many orders of magnitude faster than the original plan.

| **Related reference:**

| “Adaptive Query Processing in Visual Explain” on page 144
| You can use Visual Explain to request a new plan.

| **AQP join order**

| Adaptive Query Processing analyzes actual query run time join statistics and uses that information for subsequent join optimizations.

| The SQE engine implements AQP join order recommendations in the following ways:

| **Subsequent to run**

| When each query completes, a fast check is done on key points of the query execution to compare actual selected records with the estimates. If there is a significant discrepancy, then a stand-alone task is notified to do a deeper analysis of the query execution.

| The query plan and the execution statistics are passed to the task. A separate task is used for the in-depth analysis so the user job is not impacted while the deep analysis is done. Each step of the join is analyzed, looking for characteristics of starvation join. Starvation join shows a significant reduction in the number of rows produced compared to the previous step. The definition of what is considered significant depends on a number of factors.

| If the criteria for starvation join are met, the actual number of records selected at key points of the query are compared to estimates. If there is a significant discrepancy between the actual and estimated record counts, the table at that join position is identified as a 'forced primary table'. This table is saved with the query plan in the system plan cache. When the query runs in the future, the optimizer retrieves the original plan from the system plan cache. The optimizer sees the forced primary table recommendation, and optimizes the query using this recommendation.

| The forced primary recommendation is used in two ways by the optimizer:

- | • The forced primary table is placed first in the join order by the join order optimization strategy.
- | • The forced primary table is used by the strategy for LPG optimization. The preceding example is a star join since table T1 is joined to the other tables in the query. t1.c3 is the column used to join T1 to T4. If an index exists over this join column, then it might be advantageous to do preselection against table T1 using the records selected from table T4. The forced primary table recommendation is used as a hint for the optimizer to consider this technique.

| **Concurrent to run**

| The preceding logic to identify starvation join can also run in a thread in parallel to the executing query. The AQP handler thread is created for longer running queries. The thread monitors the query execution and can run the same logic described earlier against partial data from the query execution.

| If the partial results show starvation join and significant differences with the record count estimates, the query is reoptimized in the thread. When the new plan is ready, the execution of the original plan is stopped and the new plan started. This scheme for correcting join problems 'on the fly' can only be carried out before any records are selected for the final result set.

| **Note:** AQP can help correct query performance problems, but it is not a substitute for a good database design coupled with a good indexing strategy.

| **Related reference:**

| "Adaptive Query Processing in Visual Explain" on page 144
| You can use Visual Explain to request a new plan.

| **Database Monitor additions for AQP**

| Additional information is logged in the database monitor when the AQP handler code replaces an executing plan.

| A new set of 30xx records is written to the database monitor reflecting the replaced plan. The user needs to be able to distinguish between records produced for the first plan iteration and records produced for subsequent optimization. To distinguish these records, an unused short integer column of the database monitor record is used as a 'plan iteration counter'.

| Column QQSMINTF is used for this purpose. For the original optimization of the query, the 30xx records have this field set to 1. Subsequent reoptimization done by AQP processing will increment the value by 1.

| The following is an example of how DB monitor output might look like when *a* is replaced 'on the fly'. The example query is the following two-file join with an ORDER BY clause over one of the tables:

```
| SELECT a.orderkey,b.orderkey  
| FROM rvdstar/item_fact3 a, rvdstar/item_fact b  
| WHERE a.quarter - 8 = b.quarter  
| ORDER BY b.orderkey
```

| Assume that an *order by pushdown* plan is chosen, then replaced using AQP while the query is running. The following is an example of what the DB monitor records might look like. The columns shown for the purposes of explaining the changes are QQRID, QQUCNT, QQSMINTF, and QQRCOD. The other fields in the monitor are not affected by AQP processing.

Table 29. Database monitor records for example query

| QQRID | QQUCNT | QQSMINTF | QQRCD |
|-------|--------|----------|-------|
| 3010 | 14 | - | - |
| 3006 | 14 | 1 | A0 |
| 3001 | 14 | 1 | I2 |
| 3000 | 14 | 1 | T1 |
| 3023 | 14 | 1 | - |
| 3007 | 14 | 1 | - |
| 3020 | 14 | 1 | I1 |
| 3014 | 14 | 1 | - |
| 5005 | 14 | 1 | - |
| 5002 | 14 | 1 | - |
| 5004 | 14 | 1 | - |
| 5007 | 14 | 1 | - |
| 3006 | 14 | 2 | B6 |
| 3000 | 14 | 2 | T1 |
| 3000 | 14 | 2 | T3 |
| 3023 | 14 | 2 | - |
| 3003 | 14 | 2 | F7 |
| 3007 | 14 | 2 | - |
| 3020 | 14 | 2 | I1 |
| 3014 | 14 | 2 | - |
| 5005 | 14 | 2 | - |
| 5002 | 14 | 2 | - |
| 5004 | 14 | 2 | - |
| 1000 | 14 | 2 | - |
| 5007 | 14 | 2 | - |
| 3019 | 14 | - | - |
| 1000 | 14 | - | - |

Notes on the preceding table:

- There is a full set of optimizer-generated records that reflect the first choice of the optimizer: an *order by pushdown* plan. These records have the QQSMINTF column value set to 1. There is a 3001 record indicating an index was used to provide the ordering. There are 3000 and 3023 records indicating a Table Scan of the second table and a temporary hash table built to aid join performance. The remaining records, including the 3014 and the 500x records, have QQSMINTF set to 1 to reflect their association with the original *order by pushdown* plan.
- There is a second full set of optimizer-generated records that reflect the second choice of the optimizer: a *sorted temporary* plan to implement the ORDER BY. These records have the QQSMINTF column value set to 2. This time there are two 3000 records indicating table scan was used to access both tables. There is a 3023 record indicating a temporary hash table was built and a 3003 record indicating the results were sorted. The remaining records, including the 3014 and the 500x records, have QQSMINTF set to 2 to reflect their association with the replacement plan.
- Both sets of optimizer records have the same unique count (QQUCNT value).

- There is a 3006 (Access Plan Rebuilt) record generated for each replacement plan (QQSMINTF > 0). The QQRCOD (reason code) value is set to a new value, 'B6'. The 'B6' value indicates the access plan was rebuilt due to AQP processing. In the example, there is a 3006 record with QQSMINTF = 1 and a QQRCOD value of 'A0'. The 1 indicates that the original optimization built the plan for the first time. There might not be a 3006 record associated with the original optimization if the optimizer was able to reuse a plan from the plan cache.
- The 1000, 3010 and 3019 records are produced by XPF at open or close time. These records are not generated by the optimizer so there are no changes due to AQP. There are one set of the records, as in previous releases, regardless of whether AQP replaced the plan. The QQSMINTF value is *NULL* for these records.
- The replacement plan is the plan that runs to completion and returns the results. To retrieve the DB monitor records from the plan that actually returns the records, it is necessary to query the DB monitor file using a subquery. Retrieve the records where the QQSMINTF value is equal to the maximum QQSMINTF value for a given QQUCNT.

Related concepts:

“Database monitor formats” on page 251

This section contains the formats used to create the database monitor SQL tables and views.

Related reference:

“Monitoring your queries using the Database Monitor” on page 121

Start Database Monitor (STRDBMON) command gathers information about a query in real time and stores this information in an output table. This information can help you determine whether your system and your queries are performing well, or whether they need fine-tuning. Database monitors can generate significant CPU and disk storage overhead when in use.

“Adaptive Query Processing in Visual Explain” on page 144

You can use Visual Explain to request a new plan.

“QAQQINI query options” on page 162

There are different options available for parameters in the QAQQINI file.

Optimizing query performance using query optimization tools

Query optimization is an iterative process. You can gather performance information about your queries and control the processing of your queries.

DB2 for IBM i – Health Center

Use the DB2 for IBM i Health Center to capture information about your database. You can view the total number of objects, the size limits of selected objects, the design limits of selected objects, environmental limits, and activity level.

Navigator view of Health Center

The System i Navigator provides a robust graphical interface to capture, view, and interact with the Health Center.

To start the health center, follow these steps:

1. In the System i Navigator window, expand the system that you want to use.
2. Expand **Databases**.
3. Right-click the database that you want to work with and select **Health Center**.

You can change your preferences by clicking **Change** and entering filter information. Click **Refresh** to update the information.

To save your health center history, do the following:

1. In the System i Navigator window, expand the system you want to use.
2. Expand **Databases**.

- | 3. Right-click the database that you want to work with and select **Health Center**.
- | 4. On the health center dialog, select the area that you want to save. For example, if you want to save the current overview, click **Save** on the Overview tab. Size limits and Design limits are not saved.
- | 5. Specify a schema and table to save the information. You can view the contents of the selected table by clicking **View Contents**. If you select to save information to a table that does not exist, the system creates the table for you.

| **Health Center SQL procedures**

| The Health Center is implemented upon several DB2 for i SQL procedures.

| IBM i users can call the Health Center SQL procedures directly.

| **QSYS2.Health_Database_Overview ():**

| The QSYS2.Health_Database_Overview() procedure returns counts of all the different types of DB2 for i objects within the target schema or schemas. The counts are broken down by object type and subtype.

| **Procedure definition:**

```
| CREATE PROCEDURE QSYS2.HEALTH_DATABASE_OVERVIEW(
|     IN ARCHIVE_OPTION INTEGER,
|     IN OBJECT_SCHEMA VARCHAR(258),
|     IN NUMBER_OF_ITEMS_ARCHIVE INTEGER,
|     IN OVERVIEW_SCHEMA VARCHAR(258),
|     IN OVERVIEW_TABLE VARCHAR(258))
|     DYNAMIC RESULT SETS 1
|     LANGUAGE C
|     SPECIFIC QSYS2.HEALTH_DATABASE_OVERVIEW
|     NOT DETERMINISTIC
|     MODIFIES SQL DATA
|     CALLED ON NULL INPUT
|     EXTERNAL NAME 'QSYS/QSQHEALTH(OVERVIEW) '
|     PARAMETER STYLE SQL;
```

| Service Program Name: QSYS/QSQHEALTH

| Default Public Authority: *USE

| Threadsafe: Yes

| **IBM i release**

| This procedure was added to IBM i in V5R4M0.

| **Parameters**

| **Archive_Option**

| (Input) The type of operation to perform for the DB2 for i Health Center overview detail.

| The supported values are:

- | • 1 = Query only, no archive action is taken
- | • 2 = Archive only
- | • 3 = Create archive and archive
- | • 4 = Query the archive

| **Note:** Option 1 produces a new result set. Options 2 and 3 simply use the results from the last Query option. Option 3 fails if the archive exists.

| **Object_Schema**

| (Input) The target schema or schemas for this operation. A single schema name can be entered. The
| '%' character can be used to direct the procedure to process all schemas with names that start with
| the same characters which appear before the '%'. When this parameter contains only the '%' character,
| the procedure processes all schemas within the database.

| **Number_Of_Items_Archive**

| (Input) The number of rows to archive.

| The archive can be used to recognize trends over time. To have meaningful historical comparisons,
| choose the row count size carefully. This argument is ignored if the Archive_Option is 1.

| **Overview_Table**

| (Input) The table that contains the database overview archive.

| This argument is ignored if the Archive_Option is 1.

| **Authorities**

| To query an existing archive, *USE object authority is required for the Overview_Schema and
| Overview_Table. To create an archive, *CHANGE object authority is required for the Overview_Schema.
| To add to an existing archive, *CHANGE object authority is required for the Overview_Table and *USE
| object authority is required for the Overview_Schema.

| **Result Set**

| When Archive_Option is 1 or 4, a single result set is returned.

| The format of the result is as follows.

| QSYS2.Health_Database_Overview () result set format:

```
| "TIMESTAMP" TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,  
| SCHEMAS BIGINT NOT NULL ,  
| GRP01 CHAR(1) DEFAULT NULL ,  
| TABLES BIGINT NOT NULL ,  
| PARTITIONED_TABLES FOR COLUMN TABLESRT BIGINT NOT NULL ,  
| DISTRIBUTED_TABLES FOR COLUMN TABLES_DST BIGINT NOT NULL ,  
| MATERIALIZED_QUERY_TABLES FOR COLUMN TABLES_MAT BIGINT NOT NULL ,  
| PHYSICAL_FILES FOR COLUMN TABLESHY BIGINT NOT NULL ,  
| SOURCE_FILES FOR COLUMN TABLES_SRC BIGINT NOT NULL ,  
| GRP02 CHAR(1) DEFAULT NULL ,  
| VIEWS BIGINT NOT NULL ,  
| LOGICAL_FILES FOR COLUMN VIEWS_LGL BIGINT NOT NULL ,  
| GRP03 CHAR(1) DEFAULT NULL ,  
| BINARY_RADIX_INDEXES FOR COLUMN INDEXES_BI BIGINT NOT NULL ,  
| EVI_INDEXES FOR COLUMN INDEXES_EV BIGINT NOT NULL ,  
| GRP04 CHAR(1) DEFAULT NULL ,  
| PRIMARY_KEY_CONSTRAINTS FOR COLUMN CSTSRI BIGINT NOT NULL ,  
| UNIQUE_CONSTRAINTS FOR COLUMN CSTS_UNQ BIGINT NOT NULL ,  
| CHECK_CONSTRAINTS FOR COLUMN CSTS_CHK BIGINT NOT NULL ,  
| REFERENTIAL_CONSTRAINTS FOR COLUMN CSTS_RI BIGINT NOT NULL ,  
| GRP05 CHAR(1) DEFAULT NULL ,  
| EXTERNAL_TRIGGERS FOR COLUMN TRGS_EXT BIGINT NOT NULL ,  
| SQL_TRIGGERS FOR COLUMN TRGS_SQL BIGINT NOT NULL ,  
| INSTEAD_OF_TRIGGERS FOR COLUMN TRGS_INSTD BIGINT NOT NULL ,  
| GRP06 CHAR(1) DEFAULT NULL ,  
| ALIASES BIGINT NOT NULL ,  
| DDM_FILES BIGINT NOT NULL ,  
| GRP07 CHAR(1) DEFAULT NULL ,  
| EXTERNALPROCEDURES FOR COLUMN PROCS_EXT BIGINT NOT NULL ,  
| SQLPROCEDURES FOR COLUMN PROCS_SQL BIGINT NOT NULL ,  
| GRP08 CHAR(1) DEFAULT NULL ,  
| EXTERNAL_SCALAR_FUNCTIONS FOR COLUMN FUNCS_EXTS BIGINT NOT NULL ,  
| EXTERNAL_TABLE_FUNCTIONS FOR COLUMN FUNCS_EXTT BIGINT NOT NULL ,  
| SOURCE_SCALAR_FUNCTIONS FOR COLUMN FUNCS_SRCS BIGINT NOT NULL ,  
| SOURCE_AGGREGATE_FUNCTIONS FOR COLUMN FUNCS_SRCR BIGINT NOT NULL ,  
| SQL_SCALAR_FUNCTIONS FOR COLUMN FUNCS_SQLS BIGINT NOT NULL ,  
| SQL_TABLE_FUNCTIONS FOR COLUMN FUNCS_SQLT BIGINT NOT NULL ,  
| GRP09 CHAR(1) DEFAULT NULL ,
```

```

| SEQUENCES BIGINT NOT NULL ,
| SQLACKAGES FOR COLUMN SQLPKGS    BIGINT NOT NULL ,
| USER_DEFINED_DISTINCT_TYPES FOR COLUMN UDTS BIGINT NOT NULL ,
| JOURNALS BIGINT NOT NULL ,
| JOURNAL_RECEIVERS FOR COLUMN JRRCV BIGINT NOT NULL ,
| "SCHEMA" VARCHAR(258) ALLOCATE(10) NOT NULL
|
| LABEL ON COLUMN <result set>
| ( "TIMESTAMP" IS 'Timestamp' ,
|   SCHEMAS IS 'Schemas' ,
|   GRP01 IS 'Tables' ,
|   TABLES IS 'Non-partitioned tables' ,
|   PARTITIONED_TABLES IS 'Partitioned tables' ,
|   DISTRIBUTED_TABLES IS 'Distributed tables' ,
|   MATERIALIZED_QUERY_TABLES IS 'Materialized query tables' ,
|   PHYSICAL_FILES IS 'Physical files' ,
|   SOURCE_FILES IS 'Source files' ,
|   GRP02 IS 'Views' ,
|   VIEWS IS 'Views' ,
|   LOGICAL_FILES IS 'Logical files' ,
|   GRP03 IS 'Indexes' ,
|   BINARY_RADIX_INDEXES IS 'Binary radix indexes' ,
|   EVI_INDEXES IS 'Encoded vector indexes' ,
|   GRP04 IS 'Constraints' ,
|   PRIMARY_KEY_CONSTRAINTS IS 'PRIMARY KEY constraints' ,
|   UNIQUE_CONSTRAINTS IS 'UNIQUE constraints' ,
|   CHECK_CONSTRAINTS IS 'CHECK constraints' ,
|   REFERENTIAL_CONSTRAINTS IS 'Referential constraints' ,
|   GRP05 IS 'Triggers' ,
|   EXTERNAL_TRIGGERS IS 'External triggers' ,
|   SQL_TRIGGERS IS 'SQL triggers' ,
|   INSTEAD_OF_TRIGGERS IS 'INSTEAD OF triggers' ,
|   GRP06 IS 'Aliases' ,
|   ALIASES IS 'Aliases' ,
|   DDM_FILES IS 'DDM files' ,
|   GRP07 IS 'Procedures' ,
|   EXTERNALPROCEDURES IS 'External procedures' ,
|   SQLPROCEDURES IS 'SQL procedures' ,
|   GRP08 IS 'Functions' ,
|   EXTERNAL_SCALAR_FUNCTIONS IS 'External scalar functions' ,
|   EXTERNAL_TABLE_FUNCTIONS IS 'External table functions' ,
|   SOURCE_SCALAR_FUNCTIONS IS 'Source scalar functions' ,
|   SOURCE_AGGREGATE_FUNCTIONS IS 'Source aggregate functions' ,
|   SQL_SCALAR_FUNCTIONS IS 'SQL scalar functions' ,
|   SQL_TABLE_FUNCTIONS IS 'SQL table functions' ,
|   GRP09 IS 'Miscellaneous' ,
|   SEQUENCES IS 'Sequences' ,
|   SQLACKAGES IS 'SQL packages' ,
|   USER_DEFINED_DISTINCT_TYPES IS 'User-defined distinct types' ,
|   JOURNALS IS 'Journals' ,
|   JOURNAL_RECEIVERS IS 'Journal receivers' ,
|   "SCHEMA" IS 'Schema mask' ) ;

```

Error Messages

Table 30. Error messages

| Message ID | Error Message Text |
|------------|--|
| SQL0462 W | This warning appears in the job log if the procedure encounters objects for which the user does not have *USE object authority. The warning is provided as an indication that the procedure was unable to process all available objects. |

Usage Notes

None

Related Information

None

Examples

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 390.

Example 1

Retrieve the overview for the entire database.

```
CALL QSYS2.Health_Database_Overview(1, '%', NULL, NULL, NULL);
```

Example results in System i Navigator:

| Metric - 10/22/09 7:33:39 AM | Value |
|------------------------------|--------|
| Schemas | 778 |
| Tables | |
| Non-partitioned tables | 4,541 |
| Partitioned tables | 32 |
| Distributed tables | 59 |
| Materialized query tables | 7 |
| Physical files | 11,235 |
| Source files | 1,646 |
| Views | |
| Views | 5,129 |
| Logical files | 1,824 |
| Indexes | |
| Constraints | |
| Triggers | |
| Aliases | |
| Procedures | |
| Functions | |
| Miscellaneous | |

Example 2

Archive all rows in the overview to an SQL table named MYLIB/ARCHIVE1.

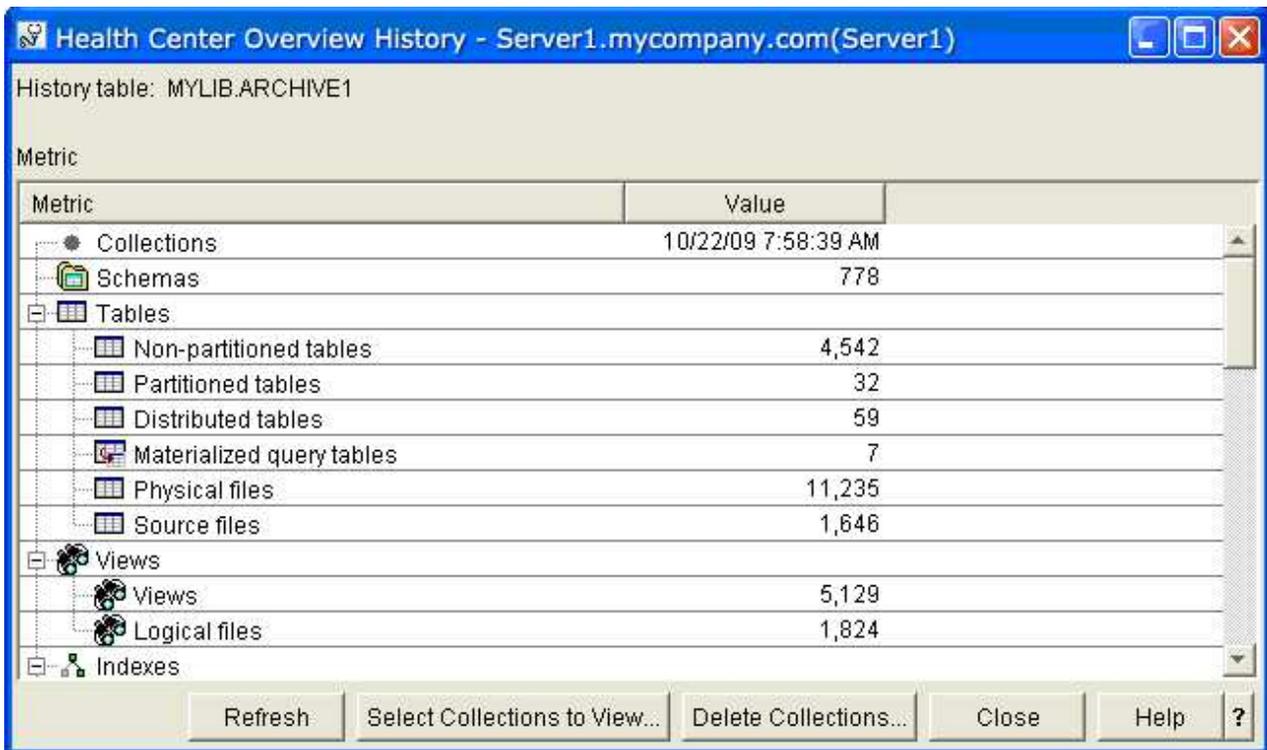
```
CALL QSYS2.Health_Database_Overview(3, '%', 2147483647, 'MYLIB', 'ARCHIVE1')
```

Example 3

Retrieve the overview from MYLIB/ARCHIVE1.

```
CALL QSYS2.Health_Database_Overview(4, '%', NULL, 'MYLIB', 'ARCHIVE1')
```

Example results in System i Navigator:



| Metric | Value |
|---------------------------|---------------------|
| Collections | 10/22/09 7:58:39 AM |
| Schemas | 778 |
| Tables | |
| Non-partitioned tables | 4,542 |
| Partitioned tables | 32 |
| Distributed tables | 59 |
| Materialized query tables | 7 |
| Physical files | 11,235 |
| Source files | 1,646 |
| Views | |
| Views | 5,129 |
| Logical files | 1,824 |
| Indexes | |

QSYS2.Health_Activity ():

The QSYS2.Health_Activity () procedure returns summary counts of database and SQL operations over a set of objects within one or more schemas.

Procedure definition:

```
CREATE PROCEDURE QSYS2.HEALTH_ACTIVITY(  
  IN ARCHIVE_OPTION INTEGER,  
  IN REFRESH_CURRENT_VALUES INTEGER,  
  IN OBJECT_SCHEMA VARCHAR(258),  
  IN OBJECT_NAME VARCHAR(258),  
  IN NUMBER_OBJECTS_ACTIVITY_TO_ARCHIVE INTEGER,  
  IN NUMBER_OF_ACTIVITY_ARCHIVE INTEGER,  
  IN ACTIVITY_SCHEMA VARCHAR(258),  
  IN ACTIVITY_TABLE VARCHAR(258))  
  DYNAMIC RESULT SETS 1  
  LANGUAGE C  
  SPECIFIC QSYS2.HEALTH_ACTIVITY  
  NOT DETERMINISTIC
```

```
|      MODIFIES SQL DATA
|      CALLED ON NULL INPUT
|      EXTERNAL NAME 'QSYS/QSQHEALTH(ACTIVITY) '
|      PARAMETER STYLE SQL;
```

| Service Program Name: QSYS/QSQHEALTH

| Default Public Authority: *USE

| Threadsafe: Yes

| **IBM i release**

| This procedure was added to IBM i 6.1.

| **Parameters**

| **Archive_Option**

| (Input) The type of operation to perform for the DB2 for i Health Center overview detail.

| The supported values are:

- | • 1 = Query only, no archive action is taken
- | • 2 = Archive only
- | • 3 = Create archive and archive
- | • 4 = Query the archive

| **Note:** Option 1 produces a new result set. Options 2 and 3 simply use the results from the last Query option. Option 3 fails if the archive exists.

| **Refresh_Current_Values**

| (Input) This option directs how the archive operation is done. This option is only valid with archive options 2 and 3.

| The supported values are:

- | • 0 = No. Indicates that we capture the activity on the entire set of specified schemas and objects.
- | • 1 = Yes. Indicates that we only refresh the activity of the objects previously captured (based on the short names).
- | • 2 = None. Use the results from the prior call. A call must have been performed in this job before using this option

| **Object_Schema**

| (Input) The target schema or schemas for this operation. A single schema name can be entered. The '%' character can be used to direct the procedure to process all schemas with names that start with the same characters which appear before the '%'. When this parameter contains only the '%' character, the procedure processes all schemas within the database.

| This name also affects the items refreshed if Refresh_Current_Values = 1.

| **Object_Name**

| (Input) The target object name for this operation. Only the '%' character is treated as a wildcard since an underscore is a valid character in a name. The name must be delimited, if necessary, and case sensitive.

| This name also affects the items refreshed if Refresh_Current_Values = 1.

| **Number_Objects_Activity_to_Archive**

| (Input) The number of objects to save for each activity.

| **Number_Of_Activity_Archive**

| (Input) The number of rows to save per object activity.

| The archive can be used to recognize trends over time. To have meaningful historical comparisons,
| choose the row count size carefully. This argument is ignored if the Archive_Option is 1 or 4.

| **Activity_Schema**

| (Input) The table that contains the database activity archive.

| This argument is ignored if the Archive_Option is 1.

| **Activity_Table**

| The table that contains the database activity archive.

| This argument is ignored if the Archive_Option is 1.

| **Authorities**

| To query an existing archive, *USE object authority is required for the Activity_Schema and
| Activity_Table. To create an archive, *CHANGE object authority is required for the Activity_Schema. To
| add to an existing archive, *CHANGE object authority is required for the Activity_Table and *USE object
| authority is required for the Activity_Schema.

| When Archive_Option is 1 or 3, *USE object authority is required for the Object_Schema and for any
| objects which are indicated by Object_Name. When an object is encountered and the caller does not have
| *USE object authority, an SQL0462 warning is placed in the job log. The object is skipped and not
| included in the procedure result set.

| **Result Set**

| When Archive_Option is 1 or 4, a single result set is returned.

| The format of the result is as follows. All these items were added for IBM i 6.1.

| QSYS2.Health_Activity() result set format:

```
| "TIMESTAMP" TIMESTAMP NOT NULL,  
| ACTIVITY VARCHAR(2000) ALLOCATE(20) DEFAULT NULL,  
| CURRENT_VALUE FOR COLUMN "VALUE" BIGINT DEFAULT NULL,  
| OBJECT_SCHEMA FOR COLUMN BSHEMA VARCHAR(128)ALLOCATE(10) DEFAULT NULL,  
| OBJECT_NAME FOR COLUMN BNAME VARCHAR(128) ALLOCATE(20) DEFAULT NULL,  
| OBJECT_TYPE FOR COLUMN BTYPE VARCHAR(24) ALLOCATE(10) DEFAULT NULL,  
| SYSTEM_OBJECT_SCHEMA FOR COLUMN SYS_DNAME VARCHAR(10) ALLOCATE(10)DEFAULT NULL,  
| SYSTEM_OBJECT_NAME FOR COLUMN SYS_ONAME VARCHAR(10) ALLOCATE(10) DEFAULT NULL,  
| PARTITION_NAME FOR COLUMN MBRNAME VARCHAR(10) ALLOCATE(10) DEFAULT NULL,  
| ACTIVITY_ID FOR COLUMN ACTIV00001 INTEGER DEFAULT NULL  
  
| LABEL ON COLUMN <result set>  
| ("TIMESTAMP" IS 'Timestamp',  
| ACTIVITY IS 'Activity',  
| CURRENT_VALUE IS 'Current Value',  
| OBJECT_SCHEMA IS 'Object Schema',  
| OBJECT_NAME IS 'Object Name',  
| OBJECT_TYPE IS 'Object Type',  
| SYSTEM_OBJECT_SCHEMA IS 'System Object Schema',  
| SYSTEM_OBJECT_NAME IS 'System Object Name',  
| PARTITION_NAME IS 'Partition Name',  
| ACTIVITY_ID IS 'Activity ID');
```

| **Limit Detail**

| The supported Database Health Center Activity can be seen on any machine by executing this query. The
| supported value column contains zeros because this category of Health Center information is not tied to a
| limit.

| SELECT * FROM QSYS2.SQL_SIZING WHERE SIZING_ID BETWEEN 18000 AND 18199;

| **Note:** The **bold** rows were added in IBM i 7.1.

Table 31. Summary counts of database and SQL operations within a schema.

| SIZING_ID | SIZING_NAME | SUPPORTED_VALUE |
|--------------|----------------------------------|-----------------|
| 18100 | INSERT OPERATIONS | 0 |
| 18101 | UPDATE OPERATIONS | 0 |
| 18102 | DELETE OPERATIONS | 0 |
| 18103 | LOGICAL READS | 0 |
| 18104 | PHYSICAL READS | 0 |
| 18105 | CLEAR OPERATIONS | 0 |
| 18106 | INDEX BUILDS/REBUILDS | 0 |
| 18107 | DATA SPACE REORGANIZE OPERATIONS | 0 |
| 18108 | DATA SPACE COPY OPERATIONS | 0 |
| 18109 | FULL OPENS | 0 |
| 18110 | FULL CLOSES | 0 |
| 18111 | DAYS USED | 0 |
| 18112 | INDEX QUERY USE | 0 |
| 18113 | INDEX QUERY STATISTICS USE | 0 |
| 18114 | INDEX LOGICAL READS | 0 |
| 18115 | INDEX RANDOM READS | 0 |
| 18116 | SQL STATEMENT COMPRESSION COUNT | 0 |
| 18117 | SQL STATEMENT CONTENTION COUNT | 0 |
| 18118 | RANDOM READS | 0 |
| 18119 | SEQUENTIAL READS | 0 |

Error Messages

Table 32. Error messages

| Message ID | Error Message Text |
|------------|--|
| SQL0462 W | This warning appears in the job log if the procedure encounters objects for which the user does not have *USE object authority. The warning is provided as an indication that the procedure was unable to process all available objects. |

Usage Notes

None

Related Information

None

Example

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 390.

Retrieve the activity information for all objects within the QSYS2 schema, using a maximum of 10 objects per each activity.

```
CALL QSYS2.Health_Activity(1, 0, 'QSYS2', '%', 10, NULL, NULL, NULL);
```

Example results in System i Navigator:

The screenshot shows the 'Health Center - Server1(Server1)' window with the 'Activity' tab selected. The 'Settings for analysis' section is configured with 'Use the following filters' selected, 'Schema: QSYS2', 'Object: All names', and 'Objects per activity: 10'. The 'Activity - 10/26/09 10:18:18 AM' section displays a table of activity data.

| Activity - 10/26/09 10:18:18 AM | Value | Status |
|-----------------------------------|--------|--------|
| Insert operations | | |
| Update operations | | |
| Delete operations | | |
| Logical reads | | |
| Physical reads | | |
| Clear operations | | |
| Full opens | | |
| Full closes | | |
| Days used | | |
| Index query use | | |
| SQL QSYS2.SYSTXTINDI (SYSTE00001) | 21,155 | Normal |
| SQL QSYS2.SYSTXTINDI (SYSTE00001) | 14,756 | Normal |
| SQL QSYS2.QASQRESL (SYSRO00001) | 10,109 | Normal |
| SQL QSYS2.SYSTXTCOLI (SYSTE00001) | 9,872 | Normal |
| QSYS2.QASQSPDP (QASQDRDP) | 9,818 | Normal |
| QSYS2.QASQDRDP (QASQDRDP) | 9,818 | Normal |
| SQL QSYS2.QASEQOBJ (SYSSEQOBJ) | 9,724 | Normal |
| SQL QSYS2.SYSTXTINDI (SYSTE00001) | 7,796 | Normal |
| QSYS2.QASEQOBJ (QASEQOBJ) | 814 | Normal |
| QSYS2.SYSTXTCOLI (SYSTXTCOLI) | 439 | Normal |
| Index query statistics use | | |
| Index logical reads | | |

At the bottom of the window, there are buttons for 'Refresh', 'View History...', 'Save...', 'Change Status Threshold...', 'Close', and 'Help ?'.

| QSYS2.Health_Design_Limits ():

| The QSYS2.Health_Design_Limits () procedure returns detailed counts of design limits over a set of objects within one or more schemas. Design limits correspond to architectural constructs, such as 'Maximum number of columns in a table or view'.

| Procedure definition:

```
| CREATE PROCEDURE QSYS2.HEALTH_DESIGN_LIMITS(  
|     ARCHIVE_OPTION INTEGER,  
|     IN_REFRESH_CURRENT_VALUES INTEGER,  
|     IN_OBJECT_SCHEMA VARCHAR(258),  
|     IN_OBJECT_NAME VARCHAR(258),  
|     IN_NUMBER_OBJECTS_LIMIT_TO_ARCHIVE INTEGER,  
|     IN_NUMBER_OF_LIMITS_ARCHIVE INTEGER,  
|     IN_LIMIT_SCHEMA VARCHAR(258),  
|     IN_LIMIT_TABLE VARCHAR(258),  
|     DYNAMIC_RESULT_SETS 1  
|     LANGUAGE C  
|     SPECIFIC QSYS2.HEALTH_DESIGN_LIMITS  
|     NOT DETERMINISTIC  
|     MODIFIES SQL DATA  
|     CALLED ON NULL INPUT  
|     EXTERNAL NAME 'QSYS/QSQHEALTH(DESIGN)'  
|     PARAMETER STYLE SQL;
```

| Service Program Name: QSYS/QSQHEALTH

| Default Public Authority: *USE

| Threadsafe: Yes

| IBM i release

| This procedure was added to IBM i V5R4M0.

| Parameters

| Archive_Option

| (Input) The type of operation to perform for the DB2 for i Health Center activity detail.

| The supported values are:

- | • 1 = Query only, no archive action is taken
- | • 2 = Archive only
- | • 3 = Create archive and archive
- | • 4 = Query the archive

| **Note:** Option 1 produces a new result set. Options 2 and 3 simply use the results from the last Query option. Option 3 fails if the archive exists.

| Refresh_Current_Values

| (Input) This option directs how the archive operation is done. This option is only valid with archive options 2 and 3.

| The supported values are:

- | • 0 = No. Indicates that we capture the activity on the entire set of specified schemas and objects.
- | • 1 = Yes. Indicates that we only refresh the activity of the objects previously captured (based on the short names).
- | • 2 = None. Use the results from the prior call. A call must have been performed in this job before using this option

| **Object_Schema**

| (Input) The target schema or schemas for this operation. A single schema name can be entered. The
| '%' character can be used to direct the procedure to process all schemas with names that start with
| the same characters which appear before the '%'. When this parameter contains only the '%' character,
| the procedure processes all schemas within the database.

| This name also affects the items refreshed if Refresh_Current_Values = 1.

| **Object_Name**

| (Input) The target object name for this operation. Only the '%' character is treated as a wildcard since
| an underscore is a valid character in a name. The name must be delimited, if necessary, and case
| sensitive.

| This name also affects the items refreshed if Refresh_Current_Values = 1.

| **Number_Objects_Limit_to_Archive**

| (Input) The number of objects to save for each design limit.

| **Number_Of_Limits_Archive**

| (Input) The number of rows to save per object design limit.

| The archive can be used to recognize trends over time. To have meaningful historical comparisons,
| choose the row count size carefully. This argument is ignored if the Archive_Option is 1 or 4.

| **Limit_Schema**

| (Input) The schema that contains the database limit archive.

| This argument is ignored if the Archive_Option is 1.

| **Limit_Table**

| The table that contains the database limit archive.

| This argument is ignored if the Archive_Option is 1.

| **Authorities**

| To query an existing archive, *USE object authority is required for the Limit_Schema and Limit_Table. To
| create an archive, *CHANGE object authority is required for the Limit_Schema. To add to an archive,
| *CHANGE object authority is required for the Limit_Table.

| When Archive_Option is 1 or 3, *USE object authority is required for the Object_Schema and for any
| objects which are indicated by Object_Name. When an object is encountered and the caller does not have
| *USE object authority, an SQL0462 warning is placed in the job log. The object is skipped and not
| included in the procedure result set.

| **Result Set**

| When Archive_Option is 1 or 4, a single result set is returned.

| The format of the result is as follows. All these items were added for IBM i V5R4M0.

| QSYS2.Health_Design_Limits() result set format:

```
| "TIMESTAMP" TIMESTAMP NOT NULL,  
| LIMIT VARCHAR(2000) ALLOCATE(20) DEFAULT NULL,  
| CURRENT_VALUE FOR COLUMN "VALUE" BIGINT DEFAULT NULL,  
| PERCENT DECIMAL(5, 2) DEFAULT NULL,  
| OBJECT_SCHEMA FOR COLUMN BSCHEMA VARCHAR(128) ALLOCATE(10) DEFAULT NULL,  
| OBJECT_NAME FOR COLUMN BNAME VARCHAR(128) ALLOCATE(20) DEFAULT NULL,  
| OBJECT_TYPE FOR COLUMN BTYPE VARCHAR(24) ALLOCATE(10) DEFAULT NULL,  
| SYSTEM_OBJECT_SCHEMA FOR COLUMN SYS_DNAME VARCHAR(10) ALLOCATE(10) DEFAULT NULL,  
| SYSTEM_OBJECT_NAME FOR COLUMN SYS_ONAME VARCHAR(10) ALLOCATE(10) DEFAULT NULL,  
| PARTITION_NAME FOR COLUMN MBRNAME VARCHAR(10) ALLOCATE(10) DEFAULT NULL  
| MAXIMUM_VALUE FOR COLUMN "MAXVALUE" BIGINT DEFAULT NULL  
| LIMIT_ID INTEGER DEFAULT NULL
```

```

| LABEL ON COLUMN <result set>
| ( "TIMESTAMP" IS 'Timestamp',
|   LIMIT IS 'Limit',
|   CURRENT_VALUE IS 'Current          Value',
|   PERCENT IS 'Percent',
|   OBJECT_SCHEMA IS 'Object          Schema',
|   OBJECT_NAME IS 'Object            Name',
|   OBJECT_TYPE IS 'Object            Type',
|   SYSTEM_OBJECT_SCHEMA IS 'System   Object          Schema',
|   SYSTEM_OBJECT_NAME IS 'System     Object          Name',
|   PARTITION_NAME IS 'Partition      Name',
|   MAXIMUM_VALUE IS 'Maximum         Value',
|   LIMIT_ID IS 'Limit                ID');

```

Limit Detail

The supported Database Health Center Design limits can be seen on any machine by executing this query:

```
SELECT * FROM QSYS2.SQL_SIZING WHERE SIZING_ID BETWEEN 16000 AND 16999;
```

Table 33. Design limits over objects within a schema.

| SIZING_ID | SIZING_NAME | SUPPORTED_VALUE |
|-----------|---|---------------------|
| 16100 | MAXIMUM NUMBER OF MEMBERS | 327670 |
| 16101 | MAXIMUM NUMBER OF RECORD FORMATS | 32 |
| 16800 | MAXIMUM JOURNAL RECEIVER SIZE | 1.09951E+12 (~1 TB) |
| 16801 | TOTAL SQL STATEMENTS | 0 |
| 16802 | TOTAL ACTIVE SQL STATEMENTS | 0 |
| 16803 | MAXIMUM SQL PACKAGE SIZE | 520093696 (~500 MB) |
| 16804 | MAXIMUM LARGE SQL PACKAGE SIZE | 1056964608 (~1 GB) |
| 16805 | MAXIMUM SQL PROGRAM ASSOCIATED SPACE SIZE | 16777216 |

Error Messages

Table 34. Error messages

| Message ID | Error Message Text |
|------------|--|
| SQL0462 W | This warning appears in the job log if the procedure encounters objects for which the user does not have *USE object authority. The warning is provided as an indication that the procedure was unable to process all available objects. |

Usage Notes

None

Related Information

None

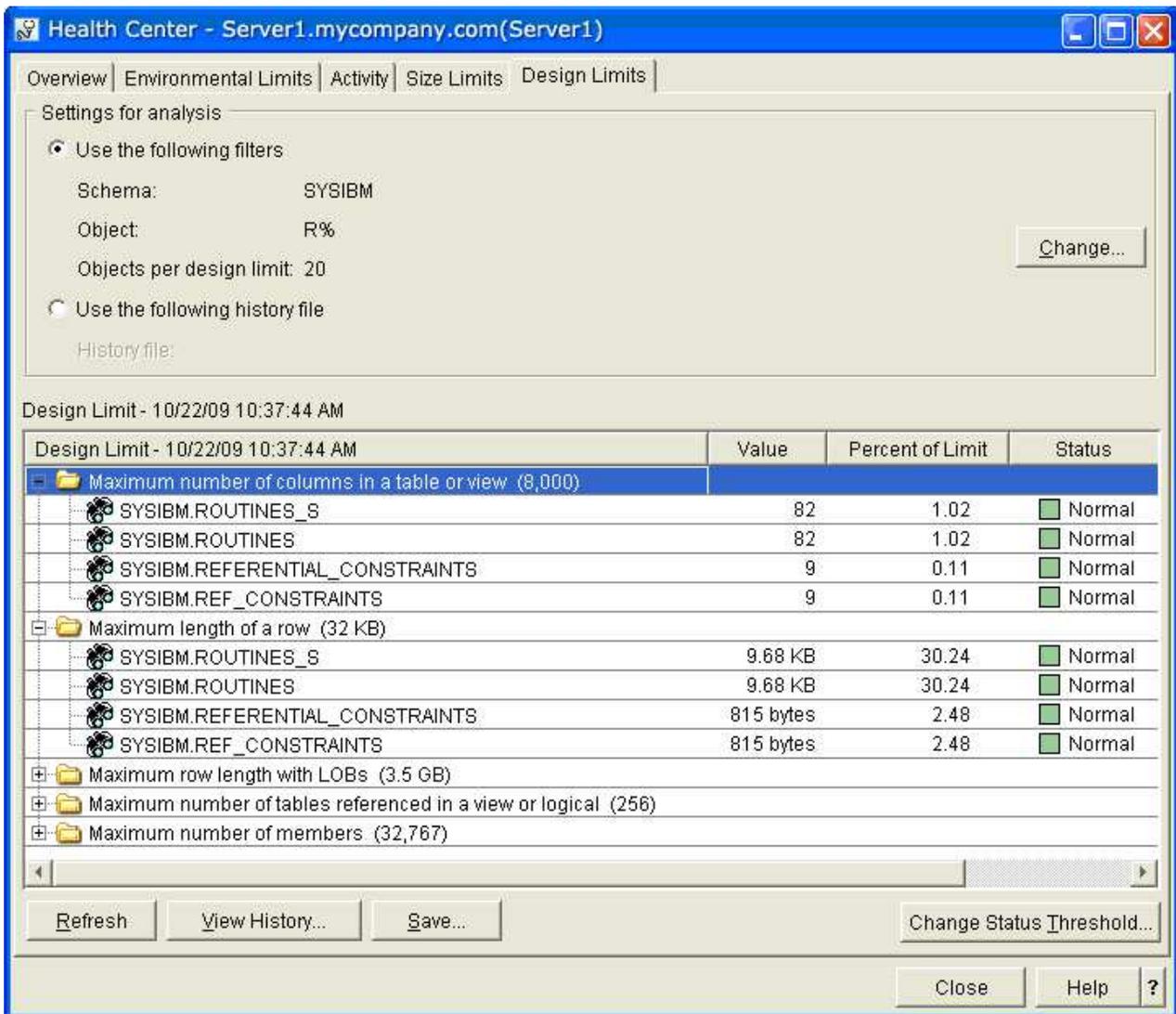
Example

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 390.

Retrieve the design limit information for all object names which start with the letter R, within the SYSIBM schema, using a maximum of 20 objects per each design limit.

```
CALL QSYS2.Health_Design_Limits(1, 0, 'SYSIBM', 'R%', 20, NULL, NULL, NULL);
```

Example results in System i Navigator:



QSYS2.Health_Size_Limits ():

The QSYS2.Health_Size_Limits () procedure returns detailed size information for database objects within one or more schemas. Size limits help you understand trends towards reaching a database limit such as 'Maximum size of the data in a table partition'.

Procedure definition:

```
CREATE PROCEDURE QSYS2.HEALTH_SIZE_LIMITS(
  IN ARCHIVE_OPTION INTEGER,
  IN REFRESH_CURRENT_VALUES INTEGER,
  IN OBJECT_SCHEMA VARCHAR(258),
  IN OBJECT_NAME VARCHAR(258),
  IN NUMBER_OBJECTS_LIMIT_TO_ARCHIVE INTEGER,
  IN NUMBER_OF_LIMITS_ARCHIVE INTEGER,
  IN LIMIT_SCHEMA VARCHAR(258),
  IN LIMIT_TABLE VARCHAR(258))
  DYNAMIC RESULT SETS 1
  LANGUAGE C
  SPECIFIC QSYS2.HEALTH_SIZE_LIMITS
```

```
| NOT DETERMINISTIC
| MODIFIES SQL DATA
| CALLED ON NULL INPUT
| EXTERNAL NAME 'QSYS/QSQHEALTH(SIZE) '
| PARAMETER STYLE SQL;
```

| Service Program Name: QSYS/QSQHEALTH

| Default Public Authority: *USE

| Threadsafe: Yes

| **IBM i release**

| This procedure was added to IBM i V5R4M0.

| **Parameters**

| **Archive_Option**

| (Input) The type of operation to perform for the DB2 for i Health Center activity detail.

| The supported values are:

- | • 1 = Query only, no archive action is taken
- | • 2 = Archive only
- | • 3 = Create archive and archive
- | • 4 = Query the archive

| **Note:** Option 1 produces a new result set. Options 2 and 3 simply use the results from the last Query option. Option 3 fails if the archive exists.

| **Refresh_Current_Values**

| (Input) This option directs how the archive operation is done. This option is only valid with archive options 2 and 3.

| The supported values are:

- | • 0 = No. Indicates that we capture the activity on the entire set of specified schemas and objects.
- | • 1 = Yes. Indicates that we only refresh the activity of the objects previously captured (based on the short names).
- | • 2 = None. Use the results from the prior call. A call must have been performed in this job before using this option

| **Object_Schema**

| (Input) The target schema or schemas for this operation. A single schema name can be entered. The '%' character can be used to direct the procedure to process all schemas with names that start with the same characters which appear before the '%'. When this parameter contains only the '%' character, the procedure processes all schemas within the database.

| This name also affects the items refreshed if Refresh_Current_Values = 1.

| **Object_Name**

| (Input) The target object name for this operation. Only the '%' character is treated as a wildcard since an underscore is a valid character in a name. The name must be delimited, if necessary, and case sensitive.

| This name also affects the items refreshed if Refresh_Current_Values = 1.

| **Number_Objects_Limit_to_Archive**

| (Input) The number of objects to save for each size limit.

| **Number_Of_Limits_Archive**

| (Input) The number of rows to save per object size limit.

| The archive can be used to recognize trends over time. To have meaningful historical comparisons, choose the row count size carefully. This argument is ignored if the Archive_Option is 1 or 4.

| **Limit_Schema**

| (Input) The schema that contains the database activity archive.

| This argument is ignored if the Archive_Option is 1.

| **Limit_Table**

| The table that contains the database activity archive.

| This argument is ignored if the Archive_Option is 1.

| **Authorities**

| To query an existing archive, *USE object authority is required for the Limit_Schema and Limit_Table. To create an archive, *CHANGE object authority is required for the Limit_Schema. To add to an archive, *CHANGE object authority is required for the Limit_Table.

| When Archive_Option is 1 or 3, *USE object authority is required for the Object_Schema and for any objects which are indicated by Object_Name. When an object is encountered and the caller does not have *USE object authority, an SQL0462 warning is placed in the job log. The object is skipped and not included in the procedure result set.

| **Result Set**

| When Archive_Option is 1 or 4, a single result set is returned.

| The format of the result is as follows. All these items were added for IBM i V5R4M0.

| QSYS2.Health_Size_Limits() result set format:

```
| "TIMESTAMP" TIMESTAMP NOT NULL,  
| LIMIT VARCHAR(2000) ALLOCATE(20) DEFAULT NULL,  
| CURRENT_VALUE FOR COLUMN "VALUE" BIGINT DEFAULT NULL,  
| PERCENT DECIMAL(5, 2) DEFAULT NULL, OBJECT_SCHEMA FOR COLUMN BSCHEMA VARCHAR(128) ALLOCATE(10) DEFAULT NULL,  
| OBJECT_NAME FOR COLUMN BNAME VARCHAR(128) ALLOCATE(20) DEFAULT NULL,  
| OBJECT_TYPE FOR COLUMN BTYPE VARCHAR(24) ALLOCATE(10) DEFAULT NULL,  
| SYSTEM_OBJECT_SCHEMA FOR COLUMN SYS_DNAME VARCHAR(10) ALLOCATE(10) DEFAULT NULL,  
| SYSTEM_OBJECT_NAME FOR COLUMN SYS_ONAME VARCHAR(10) ALLOCATE(10) DEFAULT NULL,  
| MAXIMUM_VALUE FOR COLUMN "MAXVALUE" BIGINT DEFAULT NULL,  
| LIMIT_ID INTEGER DEFAULT NULL,  
| PARTITION_NAME FOR COLUMN MBRNAME VARCHAR(10) ALLOCATE(10) DEFAULT NULL,  
| "SCHEMA" VARCHAR(258) ALLOCATE(10) DEFAULT NULL,  
| OBJECT VARCHAR(258) ALLOCATE(10) DEFAULT NULL,  
| "REFRESH" INTEGER DEFAULT NULL
```

```
| LABEL ON COLUMN <result set>  
| ( "TIMESTAMP" IS 'Timestamp',  
|   LIMIT IS 'Limit',  
|   CURRENT_VALUE IS 'Current Value',  
|   PERCENT IS 'Percent',  
|   OBJECT_SCHEMA IS 'Object Schema',  
|   OBJECT_NAME IS 'Object Name',  
|   OBJECT_TYPE IS 'Object Type',  
|   SYSTEM_OBJECT_SCHEMA IS 'System Object Schema',  
|   SYSTEM_OBJECT_NAME IS 'System Object Name',  
|   MAXIMUM_VALUE IS 'Maximum Value',  
|   LIMIT_ID IS 'Limit ID',  
|   PARTITION_NAME IS 'Partition Name',  
|   "SCHEMA" IS 'Schema Mask',  
|   OBJECT IS 'Object Mask',  
|   "REFRESH" IS 'Refresh');
```

Limit Detail

The supported Database Health Center Size limits can be seen on any machine by executing this query:

```
SELECT * FROM QSYS2.SQL_SIZING WHERE SIZING_ID BETWEEN 15000 AND 15999;
```

Note: MAXIMUM NUMBER OF OVERFLOW ROWS was added in IBM i 7.1.

Table 35. Size limit information for database objects within a schema.

| SIZING_ID | SIZING_NAME | SUPPORTED_VALUE |
|-----------|--|-----------------|
| 15000 | MAXIMUM NUMBER OF ALL ROWS | 4.29E+09 |
| 15001 | MAXIMUM NUMBER OF VALID ROWS | 4.29E+09 |
| 15002 | MAXIMUM NUMBER OF DELETED ROWS | 4.29E+09 |
| 15003 | MAXIMUM TABLE PARTITION SIZE | 1.7E+12 |
| 15004 | MAXIMUM NUMBER OF OVERFLOW ROWS | 4.29E+09 |
| 15101 | MAXIMUM ROW LENGTH | 32766 |
| 15102 | MAXIMUM ROW LENGTH WITH LOBS | 3.76E+09 |
| 15103 | MAXIMUM NUMBER OF PARTITIONS | 256 |
| 15150 | MAXIMUM NUMBER OF REFERENCED TABLES | 256 |
| 15300 | MAXIMUM NUMBER OF TRIGGERS | 300 |
| 15301 | MAXIMUM NUMBER OF CONSTRAINTS | 300 |
| 15302 | MAXIMUM LENGTH OF CHECK CONSTRAINT | 2097151 |
| 15400 | MAXIMUM *MAX4GB INDEX SIZE | 4.29E+09 |
| 15401 | MAXIMUM *MAX1TB INDEX SIZE | 1.1E+12 |
| 15402 | MAXIMUM NUMBER OF INDEX ENTRIES | 0 |
| 15500 | MAXIMUM KEY COLUMNS | 120 |
| 15501 | MAXIMUM KEY LENGTH | 32767 |
| 15502 | MAXIMUM NUMBER OF PARTITIONING KEYS | 120 |
| 15700 | MAXIMUM NUMBER OF FUNCTION PARAMETERS | 255 |
| 15701 | MAXIMUM NUMBER OF PROCEDURE PARAMETERS | 1024 |

Error Messages

Table 36. Error messages

| Message ID | Error Message Text |
|------------|--|
| SQL0462 W | This warning appears in the job log if the procedure encounters objects for which the user does not have *USE object authority. The warning is provided as an indication that the procedure was unable to process all available objects. |

Usage Notes

None

Related Information

None

Example

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 390.

Retrieve the size limit information for all object names which start with the letter S, within the SYSIBM schema, using a maximum of five objects per each design limit.

```
CALL QSYS2.Health_Size_Limits(1, 0, 'SYSIBM', 'S%', 5, NULL, NULL, NULL);
```

Example results in System i Navigator:

The screenshot shows the 'Health Center - Server1(Server1)' window with the 'Size Limits' tab selected. The 'Settings for analysis' section is configured with the following options:

- Use the following filters
- Schema: SYSIBM
- Object: S%
- Objects per size limit: 5
- Use the following history file
- History file:

The 'Size Limit - 10/22/09 12:59:12 PM' table displays the following data:

| Size Limit - 10/22/09 12:59:12 PM | Value | Percent of Limit | Status |
|---|-------|------------------|--------|
| Maximum number of all rows in a partition (4,294,967,288) | | | |
| SYSIBM.SYSCHARSETS (SYSCH00001) | 86 | 0.00 | Normal |
| SYSIBM.SQLYTYPEINFO (SQLYT00001) | 28 | 0.00 | Normal |
| SYSIBM.SYSPRIVILEGES (SYSPR00001) | 7 | 0.00 | Normal |
| SYSIBM.SYSTABLETYPES (SYSTA00001) | 5 | 0.00 | Normal |
| SYSIBM.SYSDUMMY1 (SYSDUMMY1) | 1 | 0.00 | Normal |
| Maximum number of valid rows in a partition (4,294,967,288) | | | |
| Maximum size of the data in a table partition (1.55 TB) | | | |
| Total SQL statements | | | |
| Total active SQL statements | | | |
| Maximum SQL program associated space size (16 MB) | | | |

At the bottom of the window, there are buttons for 'Refresh', 'View History...', 'Save...', and 'Change Status Threshold...'.

QSYS2.Health_Environmental_Limits ():

The QSYS2.Health_Environmental_Limits() procedure returns detail on the top 10 jobs on the system, for different SQL or application limits. The jobs do not have to be in existence. The top 10 information is maintained within DB2 for i and gets reset when the machine is IPLed, the IASP is varied ON, or when the QSYS2.Reset_Environmental_Limits() procedure is called.

Procedure definition:

```
CREATE PROCEDURE QSYS2.HEALTH_ENVIRONMENTAL_LIMITS(  
  IN ARCHIVE_OPTION INTEGER,  
  IN NUMBER_OF_LIMITS_ARCHIVE INTEGER,
```

```

|         IN LIMIT_SCHEMA VARCHAR(258),
|         IN LIMIT_TABLE VARCHAR(258))
|         DYNAMIC RESULT SETS 1
|         LANGUAGE C
|         SPECIFIC QSYS2.HEALTH_ENVIRONMENTAL_LIMITS
|         NOT DETERMINISTIC
|         MODIFIES SQL DATA
|         CALLED ON NULL INPUT
|         EXTERNAL NAME 'QSYS/QSQHEALTH(ENVIRON) '
|         PARAMETER STYLE SQL;

```

| Service Program Name: QSYS/QSQHEALTH

| Default Public Authority: *USE

| Threadsafe: Yes

| **IBM i release**

| This procedure was added to IBM i 6.1.

| **Parameters**

| **Archive_Option**

| (Input) The type of operation to perform for the DB2 for i Health Center activity detail.

| The supported values are:

- | • 1 = Query only, no archive action is taken
- | • 2 = Archive only
- | • 3 = Create archive and archive
- | • 4 = Query the archive

| **Note:** Option 1 produces a new result set. Options 2 and 3 simply use the results from the last Query option. Option 3 fails if the archive exists.

| **Number_Of_Limits_Archive**

| (Input) The number of rows to save per object health limit.

| The archive can be used to recognize trends over time. To have meaningful historical comparisons, choose the row count size carefully. This argument is ignored if the Archive_Option is 1 or 4.

| **Limit_Schema**

| (Input) The schema that contains the database activity archive.

| This argument is ignored if the Archive_Option is 1.

| **Limit_Table**

| The table that contains the database activity archive.

| This argument is ignored if the Archive_Option is 1.

| **Authorities**

| To query an existing archive, *USE object authority is required for the Limit_Schema and Limit_Table. To create an archive, *CHANGE object authority is required for the Limit_Schema. To add to an archive, *CHANGE object authority is required for the Limit_Table.

| When Archive_Option is 1 or 3, *USE object authority is required for the Object_Schema and for any objects which are indicated by Object_Name. When an object is encountered and the caller does not have *USE object authority, an SQL0462 warning is placed in the job log. The object is skipped and not included in the procedure result set.

| **Result Set**

| When Archive_Option is 1 or 4, a single result set is returned.

| The format of the result is as follows. All these items were added for IBM i 6.1.

| QSYS2.Health_Environmental_Limits() result set format:

```
| "TIMESTAMP" TIMESTAMP NOT NULL,
| LIMIT VARCHAR(2000) ALLOCATE(20) DEFAULT NULL,
| HIGHWATER_MARK_VALUE FOR COLUMN HIMARK BIGINT DEFAULT NULL,
| WHEN_VALUE_WAS_RECORDED FOR COLUMN TIMEHIT TIMESTAMP NOT NULL,
| PERCENT DECIMAL(5, 2) DEFAULT NULL,
| JOB_NAME VARCHAR(28) ALLOCATE(20) DEFAULT NULL,
| "CURRENT_USER" FOR COLUMN CUSER VARCHAR(128) ALLOCATE(10) DEFAULT NULL,
| JOB_TYPE VARCHAR(26) ALLOCATE(20) DEFAULT NULL,
| MAXIMUM_VALUE FOR COLUMN MAXVAL BIGINT DEFAULT NULL,
| JOB_STATUS VARCHAR(13) DEFAULT NULL,
| CLIENT_WRKSTNNAME FOR COLUMN "WRKSTNNAME" VARCHAR(255) DEFAULT NULL,
| CLIENT_APPLNAME FOR COLUMN "APPLNAME" VARCHAR(255) DEFAULT NULL,
| CLIENT_ACCTNG FOR COLUMN "ACCTNG" VARCHAR(255) DEFAULT NULL,
| CLIENTROGRAMID FOR COLUMN "PROGRAMID" VARCHAR(255) DEFAULT NULL,
| CLIENT_USERID FOR COLUMN "USERID" VARCHAR(255) DEFAULT NULL,
| WHEN_LIMITS_ESTABLISHED FOR COLUMN TIMESET TIMESTAMP NOT NULL,
| INTERFACE_NAME FOR COLUMN INTNAME VARCHAR(127) ALLOCATE(10) DEFAULT NULL,
| INTERFACE_TYPE FOR COLUMN INTTYPE VARCHAR(63) ALLOCATE(10) DEFAULT NULL,
| INTERFACE_LEVEL FOR COLUMN INTLEVEL VARCHAR(63) ALLOCATE(10) DEFAULT NULL,
| LIMIT_ID INTEGER DEFAULT NULL

| LABEL ON COLUMN <result set>
| ( "TIMESTAMP" IS 'Timestamp',
|  LIMIT IS 'Limit',
|  HIGHWATER_MARK_VALUE IS 'Largest          Value',
|  WHEN_VALUE_WAS_RECORDED IS 'Timestamp          When          Recorded',
|  PERCENT IS 'Percent',
|  JOB_NAME IS 'Job          Name',
|  "CURRENT_USER" IS 'Current          User',
|  JOB_TYPE IS 'Job          Type',
|  MAXIMUM_VALUE IS 'Maximum          Value',
|  JOB_STATUS IS 'Job          Status',
|  CLIENT_WRKSTNNAME IS 'Client          Workstation          Name',
|  CLIENT_APPLNAME IS 'Client          Application          Name',
|  CLIENT_ACCTNG IS 'Client          Accounting          Code',
|  CLIENTROGRAMID IS 'Client          Program          Identifier',
|  CLIENT_USERID IS 'Client          User          Identifier',
|  WHEN_LIMITS_ESTABLISHED IS 'Timestamp          Limits          Established',
|  INTERFACE_NAME IS 'Interface          Name',
|  INTERFACE_TYPE IS 'Interface          Type',
|  INTERFACE_LEVEL IS 'Interface          Level',
|  LIMIT_ID IS 'Limit          ID' );
```

| **Limit Detail**

| The supported Database Health Center Environmental limits can be seen on any machine by executing this query:

```
| SELECT * FROM QSYS2.SQL_SIZING WHERE SIZING_ID BETWEEN 18200 AND 18299;
```

| **Note:** The **bold** row was added in IBM i 7.1.

| *Table 37. SQL environmental limits.*

| SIZING_ID | SIZING_NAME | SUPPORTED_VALUE |
|-----------|--|-----------------|
| 18200 | MAXIMUM NUMBER OF LOB or XML LOCATORS PER JOB | 16000000 |
| 18201 | MAXIMUM NUMBER OF LOB or XML LOCATORS PER SERVER JOB | 209000 |
| 18202 | MAXIMUM NUMBER OF ACTIVATION GROUPS | 0 |
| 18203 | MAXIMUM NUMBER OF DESCRIPTORS | 0 |
| 18204 | MAXIMUM NUMBER OF CLI HANDLES | 160000 |

Table 37. SQL environmental limits. (continued)

| SIZING_ID | SIZING_NAME | SUPPORTED_VALUE |
|-----------|---|-----------------|
| 18205 | MAXIMUM NUMBER OF SQL OPEN CURSORS | 21754 |
| 18206 | MAXIMUM NUMBER OF SQL PSEUDO OPEN CURSORS | 0 |
| 18207 | MAXIMUM LENGTH OF SQL STATEMENT | 2097152 |

Error Messages

None

Usage Notes

None

Related Information

None

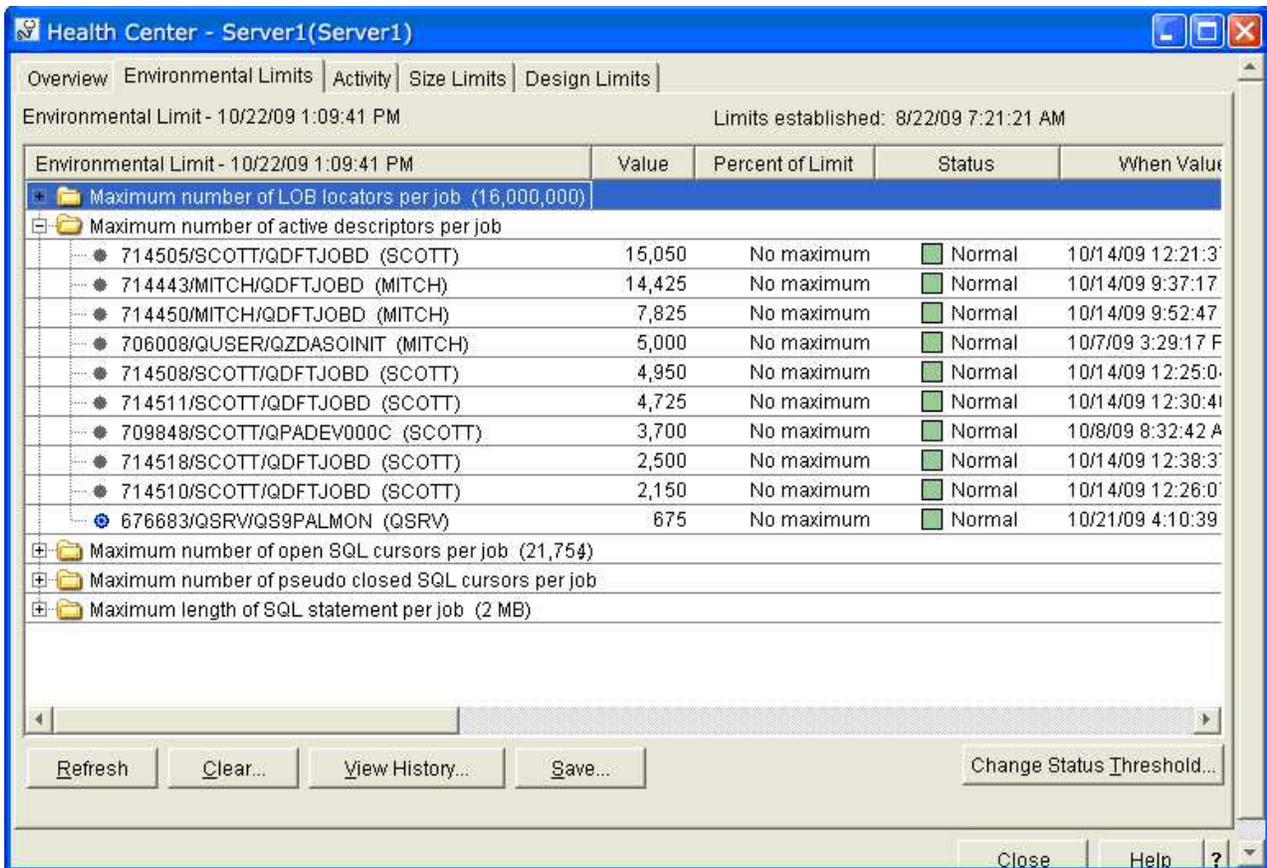
Example

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 390.

Retrieve the SQL environmental limits for the current database.

```
CALL QSYS2.Health_Environmental_Limits(1, 0, NULL, NULL);
```

Example results in System i Navigator:



QSYS2.Reset_Environmental_Limits ():

The QSYS2.Reset_Environmental_Limits () procedure clears out the environment limit cache for the database. If IASPs are being used, this procedure clears the environment limit cache for the IASP within which it is called.

Procedure definition:

```
CREATE PROCEDURE QSYS2.RESET_ENVIRONMENTAL_LIMITS(
LANGUAGE C
SPECIFIC QSYS2.RESET_ENVIRONMENTAL_LIMITS
NOT DETERMINISTIC
MODIFIES SQL DATA
CALLED ON NULL INPUT
EXTERNAL NAME 'QSYS/QSQSSUDF(RESETENV)'
PARAMETER STYLE SQL;
```

Service Program Name: QSYS/QSQHEALTH

Default Public Authority: *USE

Threadsafe: Yes

IBM i release

This procedure was added to IBM i 6.1.

Parameters

None.

Authorities

This procedure requires the user to have *JOBCTL user special authority or be authorized to the QIBM_DB_SQLADM Function through Application Administration in System i Navigator. The Change Function Usage (CHGFCNUSG) command can also be used to allow or deny use of the function.

For example:

```
CHGFCNUSG FCNID(QIBM_DB_SQLADM) USER(XXXXX) USAGE(*ALLOWED)
```

Result Set

None.

Error Messages

Table 38. Error messages

| Message ID | Error Message Text |
|------------|------------------------------|
| SQL0552 | Not authorized to PROCEDURE. |

Usage Notes

None

Related Information

None

Example

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 390.

Reset the SQL environmental limits for the current database.

```
CALL QSYS2.RESET_ENVIRONMENTAL_LIMITS;
```

Monitoring your queries using the Database Monitor

Start Database Monitor (STRDBMON) command gathers information about a query in real time and stores this information in an output table. This information can help you determine whether your system and your queries are performing well, 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, for every query on the system, or for a group of queries on the system. When a job is monitored by multiple monitors, each monitor is logging rows to a different output table. You can identify rows in the output database table by its unique identification number.

When you start a monitor using the **Start Database Monitor (STRDBMON)** command, the monitor is automatically registered with System i Navigator and appears in the System i Navigator monitor list.

Note: Database monitors also contain SQL statement text and variable values. If the variable values or SQL statements contain sensitive data you should create database monitors in a library that is not publicly authorized to prevent exposure to the sensitive data.

What kinds of statistics you can gather

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

- System and job name
- SQL statement and subselect 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 system resources.
- Take a 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, except for runtime or multi-step query information.

Related information:

Start Debug (STRDBG) command

Print SQL Information (PRTSQLINF) command

Start Database Monitor (STRDBMON) command

Start Database Monitor (STRDBMON) command

The **Start Database Monitor (STRDBMON)** command starts the collection of database performance statistics for a specified job, for all jobs on the system or for a selected set of jobs. The statistics are placed in a user-specified database table and member. If the table or member do not exist, one is created based on the QAQQDBMN table in library QSYS. If the table and member do exist, the record format of the specified table is verified to insure it is the same.

For each monitor started using the STRDBMON command, the system generates a monitor ID that can be used to uniquely identify each individual monitor. The monitor ID can be used on the ENDDDBMON

command to uniquely identify which monitor is to be ended. The monitor ID is returned in the informational message CPI436A which is generated for each occurrence of the STRDBMON command. The monitor ID can also be found in column QQC101 of the QQQ3018 database monitor record.

Informally there are two types of monitors. A private monitor is a monitor over one, specific job (or the current job). Only one (1) monitor can be started on a specific job at a time. For example, STRDBMON JOB(*) followed by another STRDBMON JOB(*) within the same job is not allowed. A public monitor is a monitor which collects data across multiple jobs. There can be a maximum of 10 public monitors active at any one time. For example, STRDBMON JOB(*ALL) followed by another STRDBMON JOB(*ALL) is allowed providing the maximum number of public monitors does not exceed 10. You could have 10 public monitors and 1 private monitor active at the same time for any specific job.

If multiple monitors specify the same output file, only one copy of the database statistic records is written to the file for each job. For example, STRDBMON OUTFILE(LIB/TABLE1) JOB(*) and STRDBMON OUTFILE(LIB/TABLE1) JOB(*ALL) target the same output file. For the current job, there are not two copies of the database statistic records—one copy for the private monitor and one copy for the public monitor. There is only one copy of the database statistic records.

If the monitor is started on all jobs (a public monitor), any jobs waiting on queues or started during the monitoring period are included in the monitor data. If the monitor is started on a specific job (a private monitor) that job must be active in the system when the command is issued. Each job in the system can be monitored concurrently by one private monitor and a maximum of 10 public monitors.

The STRDBMON command allows you to collect statistic records for a specific set or subset of the queries running on any job. This filtering can be performed over the job name, user profile, query table names, query estimated run time, TCP/IP address, or any combination of these filters. Specifying a STRDBMON filter helps minimize the number of statistic records captured for any monitor.

Example 1: Starting Public Monitoring

```
STRDBMON  OUTFILE(QGPL/FILE1)  OUTMBR(MEMBER1 *ADD)
JOB(*ALL)  FRCRCD(10)
```

This command starts database monitoring for all jobs on the system. The performance statistics are added to the member named MEMBER1 in the file named FILE1 in the QGPL library. 10 records are held before being written to the file.

Example 2: Starting Private Monitoring

```
STRDBMON  OUTFILE(*LIBL/FILE3)  OUTMBR(MEMBER2)
JOB(134543/QPGMR/DSP01)  FRCRCD(20)
```

This command starts database monitoring for job number 134543. The job name is DSP01 and was started by the user named QPGMR. The performance statistics are added to the member named MEMBER2 in the file named FILE3. 20 records are held before being written to the file.

Example 3: Starting Private Monitoring to a File in a Library in an Independent ASP

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(134543/QPGMR/DSP01)
```

This command starts database monitoring for job number 134543. The job name is DSP01 and was started by the user named QPGMR. The performance statistics are added to the member name DBMONFILE (since OUTMBR was not specified) in the file named DBMONFILE in the library named LIB41. This library could exist in more than one independent auxiliary storage pool (ASP); the library in the name space of the originator's job is always used.

Example 4: Starting Public Monitoring For All Jobs That Begin With 'QZDA

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(*ALL/*ALL/QZDA*)
```

This command starts database monitoring for all jobs that whose job name begins with 'QZDA'. The performance statistics (monitor records) are added to member DBMONFILE (since OUTMBR was not specified) in file DBMONFILE in library LIB41. This library could exist in more than one independent auxiliary storage pool (ASP); the library in the name space of the originator's job is always used.

Example 5: Starting Public Monitoring and Filtering SQL Statements That Run Over 10 Seconds

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(*ALL)  RUNTHLD(10)
```

This command starts database monitoring for all jobs. Monitor records are created only for those SQL statements whose estimated run time meets or exceeds 10 seconds.

Example 6: Starting Public Monitoring and Filtering SQL Statements That Have an Estimated Temporary Storage Over 200 MB

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(*ALL)  STGTHLD(200)
```

This command starts database monitoring for all jobs. Monitor records are created only for those SQL statements whose estimated temporary storage meets or exceeds 200 MB.

Example 7: Starting Private Monitoring and Filtering Over a Specific File

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(*)  
FTRFILE(LIB41/TABLE1)
```

This command starts database monitoring for the current job. Monitor records are created only for those SQL statements that use file LIB41/TABLE1.

Example 8: Starting Private Monitoring for the Current User

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(*)  FTRUSER(*CURRENT)
```

This command starts database monitoring for the current job. Monitor records are created only for those SQL statements that are executed by the current user.

Example 9: Starting Public Monitoring For Jobs Beginning With 'QZDA' and Filtering Over Run Time and File

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(*ALL/*ALL/QZDA*)  
RUNTHLD(10)  FTRUSER(DEVLPR1)  FTRFILE(LIB41/TTT*)
```

This command starts database monitoring for all jobs whose job name begins with 'QZDA'. Monitor records are created only for those SQL statements that meet all the following conditions:

- The estimated run time, as calculated by the query optimizer, meets, or exceeds 10 seconds
- Was executed by user 'DEVLPR1'.
- Use any file whose name begins with 'TTT' and resides in library LIB41.

Example 10: Starting Public Monitoring and Filtering SQL Statements That Have Internet Address '9.10.111.77'.

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(*ALL)  
FTRINTNETA('9.10.111.77')
```

This command starts database monitoring for all jobs. Monitor records are created only for TCP/IP database server jobs that are using the client IP version 4 address of '9.10.111.77'.

Example 11: Starting Public Monitoring and Filtering SQL Statements That Have a Port Number of 8471

```
STRDBMON  OUTFILE(LIB41/DBMONFILE)  JOB(*ALL)  FTRLCLPORT(8471)
```

This command starts database monitoring for all jobs. Monitor records are created only for TCP/IP database server jobs that are using the local port number 8471.

Example 12: Starting Public Monitoring Based on Feedback from the Query Governor

```
CHGSYSVAL QQRYSIMT(200)
STRDBMON  OUTFILE(LIB41/DBMONFILE) JOB(*ALL) FTRQRYGOVR(*COND)
```

This command starts database monitoring for all jobs whose estimated run time is expected to exceed 200 seconds, based on the response to the query governor. In this example, data is collected only if the query is canceled or a return code of 2 is returned by a query governor exit program. The query can be canceled by a user response to the inquiry message CPA4259, issued because the query exceeded the query governor limits. It can also be canceled by the program logic inside the registered query governor exit program.

Example 13: Collecting database monitor for Interactive SQL use

```
STRDBMON OUTFILE(QGPL/STRSQLMON1) OUTMBR(*FIRST *REPLACE)
JOB(*ALL/*ALL/*ALL) TYPE(*DETAIL)
FTRCLTPGM(STRSQL)
```

This command uses the database monitor pre-filter by Client Special Register Program ID to collect monitor records for all the SQL statements executed by Interactive SQL (STRSQL command) usage.

Related information:

Start Database Monitor (STRDBMON) command

End Database Monitor (ENDDDBMON) command

The **End Database Monitor (ENDDDBMON)** command ends the collection of database performance statistics for a specified job, all jobs on the system, or a selected set of jobs (for example, a generic job name).

To end a monitor, you can specify the job or the monitor ID or both. If only the JOB parameter is specified, the monitor that was started using the same exact JOB parameter is ended - if there is only one monitor which matches the specified JOB. If more than one monitor is active which matches the specified JOB, then the user uniquely identifies which monitor is to be ended by use of the MONID parameter.

When only the MONID parameter is specified, the specified MONID is compared to the monitor ID of the monitor for the current job and to the monitor ID of all active public monitors (monitors that are open across multiple jobs). The monitor matching the specified MONID is ended.

The monitor ID is returned in the informational message CPI436A. This message is generated for each occurrence of the STRDBMON command. Look in the job log for message CPI436A to find the system generated monitor ID, if needed. The monitor ID can also be found in column QQC101 of the QQQ3018 database monitor record.

Restrictions

- If a specific job name and number or JOB(*) was specified on the **Start Database Monitor (STRDBMON)** command, the monitor can only be ended by specifying the same job name and number or JOB(*) on the ENDDDBMON command.
- If JOB(*ALL) was specified on the **Start Database Monitor (STRDBMON)** command, the monitor can only be ended by specifying ENDDDBMON JOB(*ALL). The monitor cannot be ended by specifying ENDDDBMON JOB(*).

When monitoring is ended for all jobs, all the jobs on the system are triggered to close the database monitor output table. However, the ENDDDBMON command can complete before all the monitored jobs have written their final statistic records to the log. Use the **Work with Object Locks (WRKOBJLCK)** command to determine that all the monitored jobs no longer hold locks on the database monitor output table before assuming that the monitoring is complete.

Example 1: End Monitoring for a Specific Job

```
ENDDBMON JOB(*)
```

This command ends database monitoring for the current job.

Example 2: End Monitoring for All Jobs

```
ENDDBMON JOB(*ALL)
```

This command ends the monitor open across all jobs on the system. If more than one monitor with JOB(*ALL) is active, then the MONID parameter must also be specified to uniquely identify which specific public monitor to end.

Example 3: End Monitoring for an Individual Public Monitor with MONID Parameter

```
ENDDBMON JOB(*ALL) MONID(061601001)
```

This command ends the monitor that was started with JOB(*ALL) and that has a monitor ID of 061601001. Because there were multiple monitors started with JOB(*ALL), the monitor ID must be specified to uniquely identify which monitor that was started with JOB(*ALL) is to be ended.

Example 4: End Monitoring for an Individual Public Monitor with MONID Parameter

```
ENDDBMON MONID(061601001)
```

This command performs the same function as the previous example. It ends the monitor that was started with JOB(*ALL) or JOB(*) and that has a monitor ID of 061601001.

Example 5: End Monitoring for All JOB(*ALL) Monitors

```
ENDDBMON JOB(*ALL/*ALL/*ALL) MONID(*ALL)
```

This command ends all monitors that are active across multiple jobs. It does not end any monitors open for a specific job or the current job.

Example 6: End Monitoring for a Generic Job

```
ENDDBMON JOB(QZDA*)
```

This command ends the monitor that was started with JOB(QZDA*). If more than one monitor with JOB(QZDA*) is active, then the MONID parameter must also be specified to uniquely identify which individual monitor to end.

Example 7: End Monitoring for an Individual Monitor with a Generic Job

```
ENDDBMON JOB(QZDA*) MONID(061601001)
```

This command ends the monitor that was started with JOB(QZDA*) and has a monitor ID of 061601001. Because there were multiple monitors started with JOB(QZDA*), the monitor ID must be specified to uniquely identify which JOB(QZDA*) monitor is to be ended.

Example 8: End Monitoring for a Group of Generic Jobs

```
ENDDBMON JOB(QZDA*) MONID(*ALL)
```

This command ends all monitors that were started with JOB(QZDA*).

Related information:

End Database Monitor (ENDDBMON) command

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 (**Start Database Monitor (STRDBMON)**) is written out based upon a set of logical formats which are defined in the database monitor formats. These views correlate closely to the debug messages and the **Print SQL Information (PRSQLINF)** messages.

The database monitor formats section also identifies which physical columns are used for each view and what information it contains. You can use the views to identify the information that can be extracted from the monitor. These rows are defined in several different views which are not shipped with the system and must be created by the user, if wanted. The views can be created with the SQL DDL. The column descriptions are explained in the tables following each figure.

Related concepts:

“Database monitor formats” on page 251

This section contains the formats used to create the database monitor SQL tables and views.

Database monitor examples

The System i Navigator interface provides a powerful tool for gathering and analyzing performance monitor data using database monitor. However, you might want to do your own analysis of the database monitor files.

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 **Start Database Monitor (STRDBMON)** and **End Database Monitor (ENDDDBMON)** commands. 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) TYPE(*DETAIL). If this table does not exist, the command creates one from the skeleton table in QSYS/QAQQDBMN.
2. Run your application
3. **ENDDDBMON**
4. Create views over LIB/PERFDATA using the SQL DDL. Creating the views is not mandatory. All the information resides in the base table that was specified on the **STRDBMON** command. The views simply provide an easier way to view the data.

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

Related information:

Start Database Monitor (STRDBMON) command

End Database Monitor (ENDDDBMON) command

Application with table scans example:

Determine which queries in your SQL application are implemented with table scans. The complete information can be obtained by joining two views: 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 (B.End_Timestamp - B.Start_Timestamp) AS TOT_TIME, A.System_Table_Schema, A.System_Table_Name,
A.Index_Advised, A.Table_Total_Rows, C.Number_Rows_Returned, A.Estimated_Rows_Selected,
B.Statement_Text_Long
FROM LIB.QQQ3000 A, LIB.QQQ1000 B, LIB.QQQ3019 C
WHERE A.Join_Column = B.Join_Column
AND A.Join_Column = C.Join_Column
```

Sample output of this query is shown in the following table. Key to this example are the join criteria:

```
WHERE A.Join_Column = B.Join_Column
AND A.Join_Column = C.Join_Column
```

Much 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 views and then joining the views together allows you to piece together all the data for a query or set of queries. Column QQJFLD uniquely identifies all queries within a job; column QQUCNT is unique at the query level. The combination of the two, when referenced in the context of the logical views, connects the query implementation to the query statement information.

Table 39. 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. Without the SQL information row, it is 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 monitor 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 Open_Id of row QQQ3014.

Queries with table scans example:

Like 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 (D.End_TimeStamp - D.Start_TimeStamp) AS TOT_TIME, A.System_Table_Schema, A.System_Table_Name,
A.Table_Total_Rows, A.Index_Advised,
B.Open_Id, B.Open_Time,
C.Clock_Time_to_Return_All_Rows, C.Number_Rows_Returned,
D.Result_Rows, D.Statement_Text_Long
FROM LIB.QQQ3000 A INNER JOIN LIB.QQQ3014 B
ON (A.Join_Column = B.Join_Column)
LEFT OUTER JOIN LIB.QQQ3019 C
ON (A.Join_Column = C.Join_Column)
LEFT OUTER JOIN LIB.QQQ1000 D
ON (A.Join_Column = D.Join_Column)
```

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

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

Table 40. Output for All Queries that Performed Table Scans

| Lib Name | Table Name | Total Rows | Index Advised | Query OPNID | ODP Open Time | Clock Time | Recs Rtnd | Rows Rtnd | TOT_TIME | Statement Text |
|----------|------------|------------|---------------|-------------|---------------|------------|-----------|-----------|----------|--|
| LIB1 | TBL1 | 20000 | Y | | 1.1 | 4.7 | 10 | 10 | 6.2 | SELECT * FROM LIB1/TBL1 WHERE FLD1 = 'A' |

Table 40. Output for All Queries that Performed Table Scans (continued)

| Lib Name | Table Name | Total Rows | Index Advised | Query OPNID | ODP Open Time | Clock Time | Recs Rtned | Rows Rtned | TOT_TIME | Statement Text |
|----------|------------|------------|---------------|-------------|---------------|------------|------------|------------|----------|---|
| 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.

Table scan detail example:

Your next step could 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 could perform better with an index to access the data. For the queries where an index is advised, the rows selected by the query are low in comparison to the total number of table rows. This selectivity is another indication that a table scan might not be optimal. Finally, a long execution time might highlight queries that could be improved by performance tuning.

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

```
SELECT A.System_Table_Schema, A.System_Table_Name,
       A.Index_Advised, A.Index_Advised_Columns,
       A.Index_Advised_Columns_Count, B.Open_Id,
       C.Statement_Text_Long
FROM LIB.QQQ3000 A INNER JOIN LIB.QQQ3014 B
ON (A.Join_Column = B.Join_Column)
LEFT OUTER JOIN LIB.QQQ1000 C
ON (A.Join_Column = C.Join_Column)
WHERE A.Index_Advised = 'Y'
```

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

Table 41. 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 | |

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. Consider how often these queries are run and the overhead of maintaining an additional index over the table when deciding whether to create the suggested index.

If you create a permanent index over FLD1, FLD2 the next sequence of steps is as follows:

1. Start the performance monitor again
2. Rerun 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 the examples assume that data has been collected in LIB/PERFDATA and the documented views have been created.

1. How many queries are performing dynamic replans?

```
SELECT COUNT(*)
FROM LIB.QQ1000
WHERE Dynamic_Replan_Reason_Code <> 'NA'
```

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

```
SELECT Dynamic_Replan_Reason_Code, Statement_Text_Long
FROM LIB.QQ1000
WHERE Dynamic_Replan_Reason_Code <> 'NA'
```

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

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

```
SELECT COUNT(*)
FROM LIB.QQ3002
WHERE System_Table_Schema = 'LIB1'
AND System_Table_Name = 'TBL1'
```

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

```
SELECT A.System_Table_Schema, A.System_Table_Name,
A.Index_Advised_Columns, B.Statement_Text_Long
FROM LIB.QQ3002 A, LIB.QQ1000 B
WHERE A.Join_Column = B.Join_Column
AND A.System_Table_Schema = 'LIB1'
AND A.System_Table_Name = '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.System_Table_Schema, A.System_Table_Name, A.Index_Advised_Columns,
B.Open_Id, C.Statement_Text_Long
FROM LIB.QQ3002 A INNER JOIN LIB.QQ3014 B
ON (A.Join_Column = B.Join_Column)
LEFT OUTER JOIN LIB.QQ1000 C
ON (A.Join_Column = C.Join_Column)
WHERE A.System_Table_Schema LIKE '%'
AND A.System_Table_Name = '%'
```

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

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

```

SELECT CASE Statement_Function
      WHEN 'O' THEN 'Other'
      WHEN 'S' THEN 'Select'
      WHEN 'L' THEN 'DDL'
      WHEN 'I' THEN 'Insert'
      WHEN 'U' THEN 'Update'
      ELSE 'Unknown'
END, COUNT(*)
FROM LIB.QQ1000
GROUP BY Statement_Function
ORDER BY 2 DESC

```

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

```

SELECT (End_Timestamp - Start_Timestamp), Job_User,
      Current_User_Profile, Statement_Text_Long
FROM LIB.QQ1000
ORDER BY 1 DESC

```

8. Which queries are the most time consuming?

```

SELECT (A.Open_Time + B.Clock_Time_to_Return_All_Rows),
      A.Open_Id, C.Statement_Text_Long
FROM LIB.QQ3014 A LEFT OUTER JOIN LIB.QQ3019 B
      ON (A.Join_Column = B.Join_Column)
LEFT OUTER JOIN LIB.QQ1000 C
      ON (A.Join_Column = C.Join_Column)
ORDER BY 1 DESC

```

Note: This example assumes that detail data was collected (STRDBMON TYPE(*DETAIL)).

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

```

SELECT A.*
FROM LIB.PERFDATA A, LIB.QQ1000 B
WHERE A.QQJFLD = B.Join_Column

```

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. Physical column QQRCOD = 'T1' means that a table scan was performed because no indexes exist over the queried table.

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

```

SELECT COUNT(*)
FROM LIB.QQ3004
WHERE Reason_Code = 'F6'

```

11. Which SQL queries were implemented with nonreusable ODPs?

```

SELECT B.Statement_Text_Long
FROM LIB.QQ3010 A, LIB.QQ1000 B
WHERE A.Join_Column = B.Join_Column
      AND A.ODP_Implementation = 'N';

```

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

```

SELECT Estimated_Processing_Time, Open_Id
FROM LIB.QQ3014
WHERE Stopped_By_Query_Governor = 'Y'

```

Note: This example assumes that detail data was collected (STRDBMON TYPE(*DETAIL)).

13. Which queries estimated time exceeds actual time?

```

SELECT A.Estimated_Processing_Time,
      (A.Open_Time + B.Clock_Time_to_Return_All_Rows),
      A.Open_Id, C.Statement_Text_Long
FROM LIB.QQ3014 A LEFT OUTER JOIN LIB.QQ3019 B
      ON (A.Join_Column = B.Join_Column)
LEFT OUTER JOIN LIB.QQ1000 C

```

```

ON (A.Join_Column = C.Join_Column)
WHERE A.Estimated_Processing_Time/1000 >
(A.Open_Time + B.Clock_Time_to_Return_All_Rows)

```

Note: This example assumes that detail data was collected (STRDBMON TYPE(*DETAIL)).

14. Should you apply a PTF for queries containing UNIONS? Yes, if any queries are performing UNIONS. Do any of the queries perform this function?

```

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

```

Note: If the result is greater than 0, apply the PTF.

15. You are a system administrator and an upgrade to the next release is planned. You want to compare data from the two releases.
- 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 need to compare the statement text between the rows in the two tables to correlate the data.

Using System i Navigator with detailed monitors

You can work with detailed monitors from the System i Navigator interface. The detailed SQL performance monitor is the System i Navigator version of the STRDBMON database monitor, found on the native interface.

| You can start this monitor by right-clicking SQL Performance Monitors under the Database portion of the System i Navigator tree and selecting **New > Monitor**. This monitor saves detailed data in real time to a hard disk. It does not need to be paused or ended in order to analyze the results. You can also choose to run a Visual Explain based on the data gathered by the monitor. Since this monitor saves data in real time, it might have a performance impact on your system.

Starting a detailed monitor

You can start a detailed monitor from the System i Navigator interface.

| You can start this monitor by right-clicking **SQL Performance Monitors** under the Database portion of the System i Navigator tree and selecting **New > SQL Performance Monitor**.

| When you create a detailed monitor, you can filter the information that you want to capture.

Initial number of records:

| Select to specify the number of records initially allocated for the monitor. The 'Initial number of records' option is used to pre-allocate storage to the database monitor out file. When collecting large amounts of monitor records, this option improves the collection performance by avoiding automatic storage extensions that occur as a file grows in size.

Minimum estimated query runtime:

| Select to include queries that exceed a specified amount of time. Select a number and then a unit of time.

Minimum estimated temporary storage:

| Select to include queries that exceed a certain amount of temporary storage. Specify a size in MB.

Job name:

| Select to filter by a specific job name. Specify a job name in the field. You can specify the entire ID or use a wildcard. For example, 'QZDAS*' finds all jobs where the name starts with 'QZDAS.'

| **Job user:**

| Select to filter by a job user. Specify a user ID in the field. You can specify the entire ID or use a
| wildcard. For example, 'QUSER*' finds all user IDs where the name starts with 'QUSER.'

| **Current user:**

| Select to filter by the current user of the job. Specify a user ID in the field. You can specify the
| entire ID or use a wildcard. For example, 'QSYS*' finds all users where the name starts with
| 'QSYS.'

| **Client location:**

| Select to filter by Internet access. The input needs to be in IPv4 or IPv6 form.

- | 1. IP version 4 address in dotted decimal form. Specify an Internet Protocol version 4 address in
| the form nnn.nnn.nnn.nnn where each nnn is a number in the range 0 through 255.
- | 2. IP version 6 address in colon hexadecimal form. Specify an internet protocol version 6 address
| in the form xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx, where each xxxx is a hex number in the
| range 0 through FFFF. IP version 6 includes the IPv4-mapped IPv6 address form (for example,
| ::FFFF:1.2.3.4). For IP version 6, the compressed form of the address is allowed.
- | 3. IP host domain name. Specify an internet host domain name of up to 254 characters in length.

| **Local port:**

| Select to filter by port number. You can select a port from the list or else enter your own port
| number.

| Ports in the list include:

- | • 446 - DRDA/DDM
- | • 447 - DRDA/DDM
- | • 448 - Secure DRDA/DDM (SSL)
- | • 4402 - QXDAEDRSQL server
- | • 8471 - Database server
- | • 9471 - Secure database server (SSL)

| **Query Governor limits:**

| Select to search for queries that have exceeded or are expected to exceed the query governor
| limits set for the system. Choose from the following options:

- | • Always collect information when exceeded
- | • Conditional collection of information when exceeded

| **Client registers:**

| Select to filter by the client register information.

| **Statements that access these objects:**

| Select to filter by only queries that use certain tables. Click **Browse** to select tables to include. To
| remove a table from the list, select the table and click **Remove**. A maximum of 10 table names
| can be specified.

| **Activity to monitor:**

| Select to collect monitor output for user-generated queries or for both user-generated and
| system-generated queries.

You can choose which jobs you want to monitor or choose to monitor all jobs. You can have multiple instances of monitors running on your system at one time. You can create up to 10 detailed monitors to monitor all jobs. When collecting information for all jobs, the monitor will collect on previously started jobs or new jobs that are started after the monitor is created. You can edit this list by selecting and removing jobs from the **Selected jobs** list.

Analyzing detailed monitor data

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.

| | Value | Summary Available | Statements Available |
|------------------------------|-------|-------------------|----------------------|
| Overview | | | |
| SQL Statements | 51 | ✓ | ✓ |
| Users | 1 | ✓ | |
| Jobs | 1 | ✓ | |
| Threads | 1 | ✓ | |
| Average Table Rows | 14 | | |
| Average Rows Returned | 0 | | |
| Average Runtime | 0.001 | | |
| Average Parallel Degree Used | 1 | | |
| Maximum Parallel Degree | 1 | | |
| SGE | 1 | ✓ | ✓ |
| COE | 0 | ✓ | ✓ |
| System Naming | 39 | ✓ | ✓ |
| SQL Naming | 11 | ✓ | ✓ |
| Full Opens | 1 | ✓ | ✓ |
| Pseudo-Opens | 0 | ✓ | ✓ |
| Average MOTs Used | 0 | ✓ | ✓ |
| Average Indexes Used | 1 | ✓ | ✓ |
| Full Indexes Created | 0 | ✓ | ✓ |
| Sparse Indexes Created | 0 | ✓ | ✓ |
| Index From Index Created | 0 | ✓ | ✓ |
| Index Creates Advised | 2 | ✓ | ✓ |
| Advised Statistics | 0 | ✓ | ✓ |
| Temporary Tables | 0 | ✓ | ✓ |
| Sorts | 1 | ✓ | ✓ |
| Access Plans Rebuilt | 2 | ✓ | ✓ |
| Sort Sequence | 0 | ✓ | ✓ |
| Call Statements | 3 | ✓ | ✓ |
| Error | 0 | ✓ | ✓ |

On the Analysis Overview dialog, you can view overview information or else choose one of the following categories:

- How much work was requested?
- What options were provided to the optimizer?
- What implementations did the optimizer use?
- What types of SQL statements were requested?
- Miscellaneous information
- I/O information

From the Actions menu, you can choose one of the following summary predefined reports:

User summary

Contains a row of summary information for each user. Each row summarizes all SQL activity for that user.

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.

These jobs are perhaps candidates for performance tuning. You could then 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 mostly 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 many INSERTs are occurring, you might use an OVRDBF command to increase the blocking factor or the QDBENCWT API.

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. 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.

In addition, when a green check is displayed under Summary column, you can select that row and click **Summary** to view information about that row type. Click **Help** for more information about the summary report. To view information organized by statements, click **Statements**.

Comparing monitor data

You can use System i Navigator to compare data sets in two or more monitors.

System i Navigator provides you with two types of comparison. The first is a simple compare that provides a high-level comparison of the monitors or snapshots. The second is a detailed comparison. The simple compare provides you with enough data about the monitors or snapshots to help you determine if a detailed comparison is helpful.

To launch a simple compare, go to **System i Navigator > system name > SQL performance monitors**. Right-click one or more monitors and select **Compare**.

To launch a detailed comparison, select the Detailed Comparison tab.

On the Detailed Comparison dialog, you can specify information about the data sets that you want to compare.

Name The name of the monitors that you want to compare.

Schema mask

Select any names that you want the comparison to ignore. For example, consider the following scenario: You have an application running in a test schema and it is optimized. Now you move it to the production schema and you want to compare how it executes there. The statements in the comparison are identical except that the statements in the test schema use "TEST" and the statements in the production schema use "PROD". You can use the schema mask to ignore "TEST" in the first monitor and "PROD" in the second monitor. Then the statements in the two monitors appear identical.

Statements that ran longer than

The minimum runtime for statements to be compared.

Minimum percent difference

The minimum difference in key attributes of the two statements being compared that determines

if the statements are considered equal or not. For example, if you select 25% as the minimum percent different, only matching statements whose key attributes differ by 25% or more are returned.

When you click **Compare**, both monitors are scanned for matching statements. Any matches found are displayed side-by-side for comparison of key attributes of each implementation.

On the Comparison output dialog, you view statements that are included in the monitor by clicking **Show Statements**. You can also run Visual Explain by selecting a statement and clicking **Visual Explain**.

Any matches found are displayed side-by-side for comparison of key attributes of each implementation.

Viewing statements in a monitor

You can view SQL statements that are included in a detailed monitor.

Right-click any detailed monitor in the SQL performance monitor window and select **Show statements**.

The filtering options provide a way to focus in on a particular area of interest:

| **Minimum runtime for the longest execution of the statement:**

Select to include statements that exceed a certain amount of time. Select a number and then a unit of time.

| **Statements that ran on or after this date and time:**

Select to include statements run at a specified date and time. Select a date and time.

| **Statements that reference the following objects:**

| Select to include statements that use or reference certain objects. Click **Browse** to select objects to include.

| **Statements that contain the following text:**

| Select to include only those statements that contain a specific type of SQL statement. For example, specify **SELECT** if you only want to include statements that are using **SELECT**. The search is case insensitive for ease of use. For example, the string 'SELECT' finds the same entries as the search string 'select'.

| Multiple filter options can be specified. In a multi-filter case, the candidate entries for each filter are computed independently. Only those entries that are present in all the candidate lists are shown. For example, if you specified options **Minimum runtime for the longest execution of the statement** and **Statements that ran on or after this date and time**, you will be shown statements with the minimum runtime that ran on or after the specified date and time.

Related reference:

“Index advisor” on page 137

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.

Importing a monitor

You can import monitor data that has been collected using some other interface by using System i Navigator.

Monitors that are created using the **Start Database Monitor (STRDBMON)** command are automatically registered with System i Navigator. They are also included in the list of monitors displayed by System i Navigator.

To import monitor data, right-click **SQL Performance monitors** and select **Import**. Once you have imported a monitor, you can analyze the data.

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 optimizer is able to perform radix index probe over any combination of the primary key columns, plus one additional secondary key column. Therefore it is important that the first secondary key column is the most selective secondary key column. The optimizer uses radix index scan with any of the remaining secondary key columns. While radix index scan is not as fast as radix index probe, it can still reduce the number of keys selected. It is recommended that secondary key columns that are fairly selective are included.

Determine the true selectivity of any secondary key columns and whether you include those key columns in the index. When building the index, make the primary key columns the left-most key columns, followed by any of the secondary key columns chosen, prioritized by selectivity.

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. When suggesting indexes, the CQE optimizer only considers the selection criteria. It does not include join, ordering, and grouping criteria. The SQE optimizer includes selection, join, ordering, and grouping criteria when suggesting indexes.

You can access index advisor information in many different ways. These ways include:

- The index advisor interface in System i Navigator
- SQL performance monitor Show statements
- Visual Explain interface
- Querying the Database monitor view 3020 - Index advised.

Related reference:

“Overview of information available from Visual Explain” on page 143

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

“Database monitor view 3020 - Index advised (SQE)” on page 326

Displays the SQL logical view format for database monitor QQQ3020.

“Viewing statements in a monitor” on page 136

You can view SQL statements that are included in a detailed monitor.

Displaying index advisor information

You can display index advisor information from the optimizer using System i Navigator.

System i Navigator displays information found in the QSYS2/SYSIXADV system table.

To display index advisor information, follow these steps:

1. In the System i Navigator window, expand the system that you want to use.
2. Expand **Databases**.
3. Right-click the database that you want to work with and select **Index Advisor > Index Advisor**.

You can also find index advisor information for a specific schema or a specific table by right-clicking on a schema or table object.

Once you have displayed the information, you have several options. You can create an index from the list, remove the index advised from the list, or clear the list entirely. You can also right-click on an index and select **Show SQL**, launching a Run SQL Scripts session with the index creation statement.

- | Depending on if you are viewing the index advice at the database level or the schema level your list could be large. Once you have the list displayed, follow these steps to subset your list:

1. Go to the **View** menu option, and select **Customize this view > Include ...**
2. Enter the information you would like to filter the list by.
3. Press the **OK** button to get the refreshed list of index advice.

Database manager indexes advised system table:

This topic describes the indexes advised system table.

Table 42. SYSIXADV system table

| Column name | System column name | Data type | Description |
|-------------------------|--------------------|----------------|---|
| TABLE_NAME | TBNAME | VARCHAR(258) | Table over which an index is advised |
| TABLE_SCHEMA | DBNAME | VARCHAR(128) | SQL schema containing the table |
| SYSTEM_TABLE_NAME | SYS_TNAME | CHAR(10) | System table name on which the index is advised |
| PARTITION_NAME | TBMEMBER | CHAR(10) | Partition detail for the index |
| KEY_COLUMNS_ADVISED | KEYSADV | VARCHAR(16000) | Column names for the advised index |
| LEADING_COLUMN_KEYS | LEADKEYS | VARCHAR(16000) | Leading, Order Independent keys. the keys at the beginning of the KEY_COLUMNS_ADVISED field which could be reordered and still satisfy the index being advised. |
| INDEX_TYPE | INDEX_TYPE | CHAR(14) | Radix (default) or EVI |
| LAST_ADVISED | LASTADV | TIMESTAMP | Last time this row was updated |
| TIMES_ADVISED | TIMESADV | BIGTINT | Number of times this index has been advised |
| ESTIMATED_CREATION_TIME | ESTTIME | INT | Estimated number of seconds for index creation |
| REASON_ADVISED | REASON | CHAR(2) | Coded reason why index was advised |
| LOGICAL_PAGE_SIZE | PAGESIZE | INT | Recommended page size for index |
| MOST_EXPENSIVE_QUERY | QUERYCOST | INT | Execution time in seconds of the query |
| AVERAGE_QUERY_ESTIMATE | QUERYEST | INT | Average execution time in seconds of the query |
| TABLE_SIZE | TABLE_SIZE | BIGINT | Number of rows in table when the index was advised |
| NLSS_TABLE_NAME | NLSSNAME | CHAR(10) | NLSS table to use for the index |
| NLSS_TABLE_SCHEMA | NLSSDBNAME | CHAR(10) | Schema name of the NLSS table |
| MTI_USED | MTIUSED | BIGINT | The number of times that this specific Maintained Temporary Index (MTI) has been used by the optimizer. The optimizer stops using a matching MTI once a permanent index is created. |
| MTI_CREATED | MTICREATED | INTEGER | The number of times that this specific Maintained Temporary Index (MTI) has been created by the optimizer. MTIs do not persist across system IPLs. |

Table 42. SYSIXADV system table (continued)

| Column name | System column name | Data type | Description |
|------------------------------|--------------------|-------------|--|
| LAST_MTI_USED | LASTMTIUSE | TIMESTAMP | The timestamp representing the last time this specific Maintained Temporary Index (MTI) was used by the optimizer to improve the performance of a query. The MTI Last Used field can be blank. The blank field indicates that an MTI which exactly matches this advice has never been used by the queries which generated this index advice. |
| AVERAGE_QUERY_ESTIMATE_MICRO | QRYMICRO | BIGINT | Average execution time in microseconds of the query which drove the index advice |
| EVI_DISTINCT_VALUES | EVIVALS | INTEGER | Recommended value to use when creating the advised EVI index. This value is <i>n</i> within the WITH <i>n</i> DISTINCT VALUES clause on the CREATE INDEX SQL statement. |
| INCLUDE_COLUMNS | INCLCOL | CLOB(10000) | EVI INCLUDE expressions for index creation. |
| FIRST_ADVISED | FIRSTADV | TIMESTAMP | When this row was inserted. |
| SYSTEM_TABLE_SCHEMA | SYS_DNAME | CHAR(10) | System name of the table schema. |

Index advisor column descriptions

Displays the columns that are used in the Index advisor window.

Table 43. Columns used in Index advisor window

| Column name | Description |
|-----------------------------------|--|
| Table for Which Index was Advised | The optimizer is advising creation of a permanent index over this table. This value is the long name for the table. The advice was generated because the table was queried and no existing permanent index could be used to improve the performance of the query. |
| Schema | Schema or library containing the table. |
| System Schema | System name of the schema. |
| System Name | System table name on which the index is advised |
| Partition | Partition detail for the index. Possible values: <ul style="list-style-type: none"> • <blank>, which means For all partitions • For Each Partition • specific name of the partition |
| Keys Advised | Column names for the advised index. The order of the column names is important. The names are listed in the same order as in the CREATE INDEX SQL statement. An exception is when the leading, order independent key information indicates that the ordering can be changed. |
| Leading Keys Order Independent | Leading, Order Independent keys. the keys at the beginning of the KEY_COLUMNS_ADVISED field which could be reordered and still satisfy the index being advised. |
| Index Type Advised | Radix (default) or EVI |
| Last Advised for Query Use | The timestamp representing the last time this index was advised for a query. |

Table 43. Columns used in Index advisor window (continued)

| Column name | Description |
|--------------------------------|--|
| Times Advised for Query Use | The cumulative number of times this index has been advised. This count ceases to increase once a matching permanent index is created. The row of advice remains in this table until the user removes it |
| Estimated Index Creation Time | Estimated time in seconds to create this index. |
| Reason advised | Reason why index was advised. Possible values are: Row selection Ordering/Grouping Row selection and Ordering/Grouping |
| Logical Page Size Advised (KB) | Recommended page size to be used on the PAGESIZE keyword of the CREATE INDEX SQL statement when creating this index. |
| Most Expensive Query Estimate | Execution time in seconds of the longest running query which generated this index advice. |
| Average of Query Estimates | Average execution time in seconds of all queries that generated this index advice. |
| Rows in Table when Advised | Number of rows in table for the last time this index was advised. |
| NLSS Table Advised | The sort sequence table in use by the query which generated the index advice. For more detail on sort sequences: |
| NLSS Schema Advised | The schema of the sort sequence table. |
| MTI Used | The number of times that this specific Maintained Temporary Index (MTI) has been used by the optimizer. |
| MTI Created | The number of times that this specific Maintained Temporary Index (MTI) has been created by the optimizer. MTIs do not persist across system IPLs. |
| MTI Last Used | The timestamp representing the last time this specific Maintained Temporary Index (MTI) was used by the optimizer to improve the performance of a query. The MTI Last Used field can be blank. A blank field indicates that an MTI which exactly matches this advice has never been used by the queries which generated this index advice. |
| EVI Distinct Values | Recommended value to use when creating the advised EVI index. This value is n within the WITH n DISTINCT VALUES clause on the CREATE INDEX SQL statement. |
| First Advised | The date/time when a row is first added to the Index Advisor table for this advice. |

Querying database monitor view 3020 - Index advised

The index advisor information can be found in the Database Monitor view 3020 - Index advised (SQE).

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.

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 can significantly reduce the number of keys selected based on the corresponding query selection. Secondary key columns are columns that might or might not significantly reduce the number of keys selected.

Column QQIDXK contains the number of suggested primary key columns that are listed in column QQIDX. These primary key columns 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 four and QQIDX specifies seven key columns, then the first four key columns are the primary key columns. The remaining three key columns are the suggested secondary key columns.

Condensing index advice

Many times, the index advisor advises several different indexes for the same table. You can condense these advised indexes into the best matches for your queries.

1. In the **System i Navigator** window, expand the system you want to use.
2. Expand **Databases**.
3. Right-click the database that you want to work with and select **Index Advisor > Condense Advised Indexes**.

Depending on if you are viewing the condensed index advice at the database level or the schema level your list could be large. Once you have the list displayed, follow these steps to subset your list:

1. Go to the **View** menu option, and select **Customize this view > Include ...**
2. Enter the information you would like to filter the list by.
3. Select **OK** to get the refreshed list of condensed index advice.

| Table for Which Index was Advised | Schema | System Name | Partition | Keys Advised | Advised Index Type | Last Advised for Query Use | Times Advised for Query Use | Estimated Index Creation | Logical Page Size Advised | Most Expensive Query Estimate | Average of Query Estimates | Rows in Table when Advised | NLSS Table Advised | MTI Used | MTI Creat. | MTI Last Used |
|-----------------------------------|--------|-------------|------------|--------------------|-----------------------------|----------------------------|-----------------------------|--------------------------|---------------------------|-------------------------------|----------------------------|----------------------------|--------------------|----------|------------|-------------------|
| MQT1 | SAMPLE | SAMPLE | MQT1 | JOB | Binary Radix | 9/1/09 3:57:14 PM | 10 | 00:00:01 | 64 | 1 | 1 | 42 | *HEX | 4 | 3 | 9/1/09 2:58:10 PM |
| EMPLOYEE | SAMPLE | SAMPLE | EMPLOYEE | JOB, WORKDEPT | Binary Radix | 9/1/09 4:06:54 PM | 8 | 00:00:01 | 64 | 1 | 1 | 14 | *HEX | 0 | 0 | |
| DEPARTMENT | SAMPLE | SAMPLE | DEPARTMENT | LOCATION, DEPTNAME | Binary Radix | 9/1/09 3:53:05 PM | 8 | 00:00:01 | 64 | 1 | 1 | 14 | *HEX | 0 | 0 | |
| DEPARTMENT | SAMPLE | SAMPLE | DEPARTMENT | LOCATION, DEPTNO | Binary Radix | 9/1/09 3:53:05 PM | 8 | 00:00:01 | 64 | 1 | 1 | 14 | *HEX | 0 | 0 | |
| DEPARTMENT | SAMPLE | SAMPLE | DEPARTMENT | DEPTNAME, LOCATION | Binary Radix | 9/1/09 3:53:12 PM | 6 | 00:00:01 | 64 | 1 | 1 | 14 | *HEX | 0 | 0 | |
| MQT2 | SAMPLE | SAMPLE | MQT2 | LOCATION, DEPTNAME | Binary Radix | 9/1/09 3:53:15 PM | 4 | 00:00:01 | 64 | 1 | 1 | 8 | *HEX | 0 | 0 | |
| MQT2 | SAMPLE | SAMPLE | MQT2 | DEPTNAME, SUM_SAL | Encoded vector (not unique) | 9/1/09 3:53:15 PM | 4 | 00:00:01 | 64 | 1 | 1 | 8 | *HEX | 0 | 0 | |

Viewing your queries with Visual Explain

You can use the **Visual Explain** tool with System i 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.

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.

Visual Explain can be used to graphically display the implementations of queries stored in the detailed SQL performance monitor. However, it does not work with tables resulting from the memory-resident monitor.

Starting Visual Explain

There are two ways to invoke the Visual Explain tool. The first, and most common, is through System i Navigator. The second is through the Visual Explain (QQQVEXPL) API.

You can start Visual Explain from any of the following windows in System i Navigator:

- Enter an SQL statement in the **Run SQL Scripts** window. Select the statement and choose **Explain** or **Run and Explain** from the **Visual Explain** menu.
- Expand the list of available SQL Performance Monitors. Right-click a detailed SQL Performance Monitor and choose the **Show Statements** option. Select filtering information and select the statement in the List of Statements window. Right-click and select **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 SQL Details for Jobs function by right-clicking **Databases** and select **SQL Details for Jobs**. Click **Apply**. Select a job from the list and right-click and select **Show Details**. When the SQL is displayed in the lower pane, you can start Visual Explain by right-clicking on **Statement** and selecting **Visual Explain**.
- Right-click SQL Plan Cache and select **Show Statements**. Select filtering information and select the statement in the List of Statements window. Right-click and select **Visual Explain**.
- Expand the list of available SQL Plan Cache Snapshots. Right-click a snapshot and select **Show Statements**. Select filtering information and select the statement in the List of Statements window. Right-click and select **Visual Explain**.
- Expand the list of SQL Plan Cache Event Monitors. Right-click an event monitor and select **Show Statements**. Select filtering information and select the statement in the List of Statements window. Right-click and select **Visual Explain**.

You have three options when running Visual Explain from Run SQL Scripts.

Visual Explain only

This option allows you to explain the query without actually running it. The data displayed represents the estimate of the query optimizer.

Note: Some queries might receive an error code 93 stating that they are too complex for displaying in Visual Explain. You can circumvent this error by selecting the "Run and Explain" option.

Run and Explain

If you select Run and Explain, the query is run by the system before the diagram is displayed. This option might take a significant amount of time, but the information displayed is more complete and accurate.

Explain while running

For long running queries, you can choose to start Visual Explain while the query is running. By refreshing the Visual Explain diagram, you can view the progress of the query.

In addition, a database monitor table that was not created as a result of using System i Navigator can be explained through System i Navigator. First you must import the database monitor table into System i Navigator. To import, right-click the SQL Performance Monitors and choose the **Import** option. Specify a name for the performance monitor (name it is known by within System i 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 System i Navigator.

You can save your Visual Explain information as an SQL Performance monitor. This monitor 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.

Related information:

Visual Explain (QQQVEXPL) API

Overview of information available from Visual Explain

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

The information includes:

- 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 query implementation, the optimizer can determine if statistics need to be created or refreshed, or if an index might 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.

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.

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 the implementation of the SQL statement. It excludes some preliminary, or intermediate operations that are not essential for understanding the main flow of query implementation. The full view might 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**. In order to see all the detail for a **Full** view, change the Graph Detail to **Full**, close out Visual Explain, and run the query again. The setting for Graph Detail persists.

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

Refresh the Visual Explain diagram

For long running queries, you can refresh the visual explain graph with runtime statistical information before the query is complete. Refresh also updates the appropriate information in the attributes section of the icon shown on the right of the screen. In order to use the **Refresh** option, you need to select **Explain while Running** from the Run SQL Scripts window.

To refresh the diagram, select **Refresh** from the **View** menu. Or click the **Refresh** button in the toolbar.

Related reference:

“Index advisor” on page 137

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.

| Adaptive Query Processing in Visual Explain

| You can use Visual Explain to request a new plan.

| There might be times when you are asked to performance tune a query while the query is still running.
| For instance, a query might be taking a long time to finish. After viewing the plan in Visual Explain, you
| decide to create the recommended index to improve the speed of the query. So you create the index and
| then want to signal the database optimizer to consider a new plan based on the new index.

| Here are the steps to request the database engine to consider a new plan while running in Visual Explain:

- | 1. Open Run SQL Scripts.
- | 2. Type in a query.
- | 3. Go to the **Visual Explain** menu and select **Explain While Running**.
- | 4. The Visual Explain window is displayed.
- | 5. Next, go to the **Actions** menu and select **Request New Plan**.

Visual Explain - Server1(Server1)

File View Actions Options Help

Request that the database engine consider a new plan for this statement

Final Select

Nested Loop Join

Fetch N Rows

Union all

Logic

Table Probe

Index Probe

Table Probe

Index Probe

Complicated

| Attribute | Value |
|---|---------------|
| Time Information | |
| Timestamp for Creation of Monit... | 2009 |
| Statement Start Timestamp | 2009 |
| Statement End Timestamp | 2009 |
| Total Estimated Run Time (ms) | 186 |
| Actual Runtime Information | |
| Optimization Time (ms) | 88 |
| Run Time (ms) | 105 |
| Statement Open Time (ms) | Not available |
| Statement Fetch Time (ms) | 105 |
| Statement Close Time (ms) | Not available |
| Rows Fetched | 51 |
| Total Times Query Was Run | 2 |
| Total Time For All Runs (ms) | 211 |
| Synchronous Database Reads | 4 |
| Asynchronous Database Reads | 17 |
| Page Faults | 41 |
| Information about SQL statement... | |
| Statement Number | 34,2 |
| Statement Function | Sele... |

select * from qsys2.syscolumns where dbname = 'QSYS2'

Statement text

A message box appears.

Select **Yes** to restart the query.

Request New Plan

This action may cause the statement to restart from the beginning. While the database will handle this restart silently, any progress made for the statement will be lost.

Would you like to continue?

Yes No

The database optimizer considers any changes to the query environment, and determines whether it is appropriate to generate a new plan. It might be possible that the database optimizer decides it is better to continue using the existing plan.

Note: This capability could also be available when selecting Visual Explain of a statement in the SQL Details for a Job window, or the SQL Plan Cache Show Statements window.

Related reference:

- | “Adaptive Query Processing” on page 95
- | Adaptive Query Processing analyzes actual query run time statistics and uses that information for subsequent optimizations.

Optimizing performance using the Plan Cache

The SQL Plan Cache contains a wealth of information about the SQE queries being run through the database. Its contents are viewable through the System i Navigator GUI interface. Certain portions of the plan cache can also be modified.

In addition, procedures are provided to allow users to programmatically work with the plan cache. These procedures can be invoked using the SQL CALL statement.

The Plan Cache interface provides a window into the database query operations on the system. The interface to the Plan Cache resides under the **System i Navigator > system name > Database**.

Within the SQL Plan Cache folder are two folders, SQL Plan Cache Snapshots and SQL Plan Cache Event Monitors.

Clicking the SQL Plan Cache Snapshots folder shows a list of any snapshots gathered so far. A snapshot is a database monitor file generated from the plan cache at the time a 'New Snapshot' is requested. It can be treated much the same as the SQL Performance Monitors list. The same analysis capability exists for snapshots as exists for traditional SQL performance monitors.

Clicking the SQL Plan Cache Event Monitors shows a list of any events that have been defined. Plan Cache event monitors, when defined, generate database monitor information from plans as they are being removed from the cache. The list includes currently active events as well as ones that have completed. Like a snapshot, the event monitor is a database monitor file. Consequently, the same analysis capability available to SQL performance monitors and snapshots can be used on the event file.

The plan cache is an actively changing cache. Therefore, it is important to realize that it contains timely information. If information over long periods of time is of interest, an event monitor could be defined to ensure that information is captured on any plans that are removed from the cache over time.

Alternatively, you could consider implementing a method of performing periodic snapshots of the plan cache to capture trends and heavy usage periods. See the discussion on IBM supplied, callable SQL procedures later in this section on plan cache.

Note: SQL plan cache snapshots and SQL plan cache event monitors also contain SQL statement text and variable values. If the variable values or SQL statements contain sensitive data you should create SQL plan cache snapshots and SQL plan cache event monitors in a library that is not publicly authorized to prevent exposure to the sensitive data.

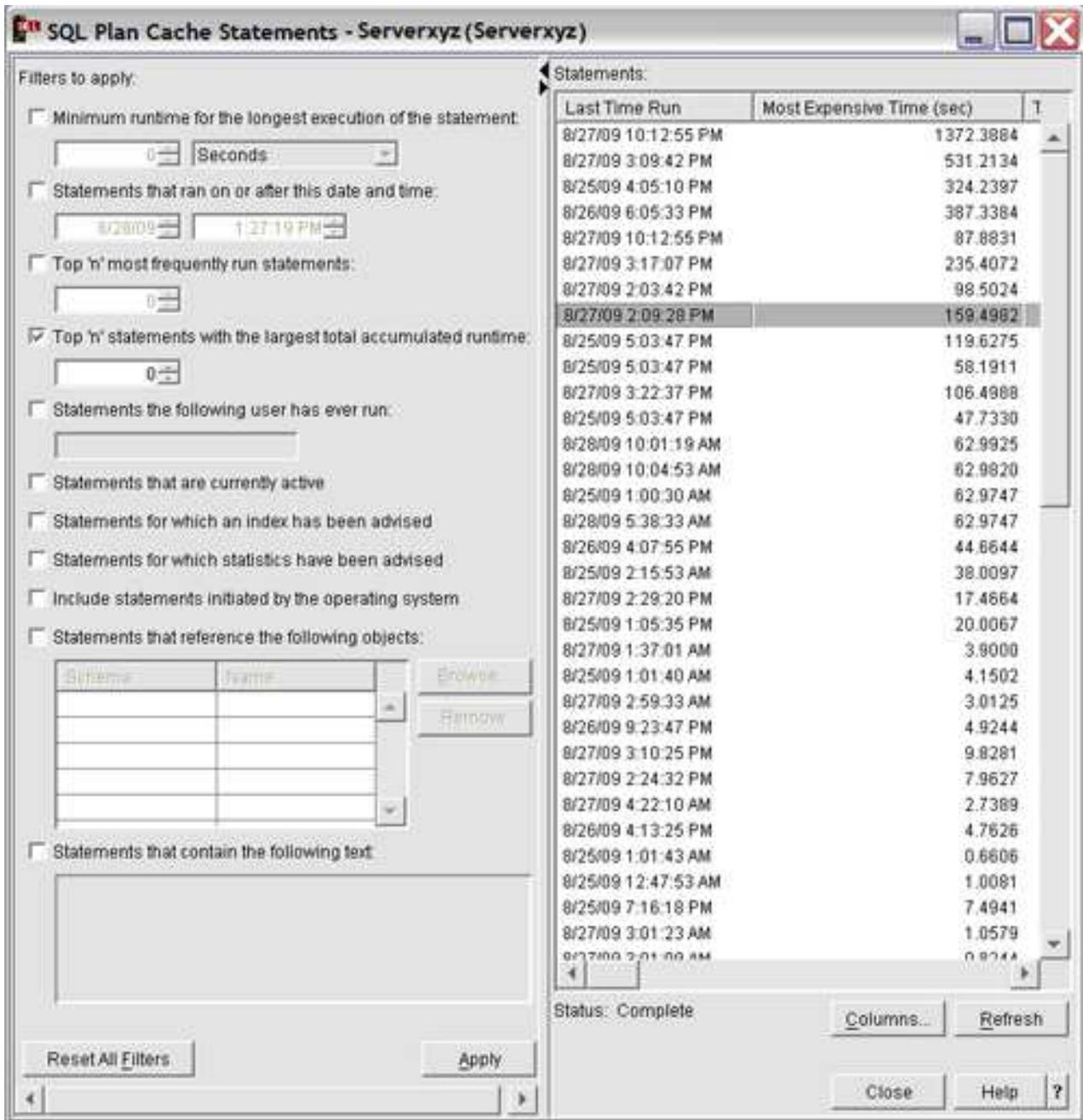
Related concepts:

“Plan cache” on page 6

The plan cache is a repository that contains the access plans for queries that were optimized by SQE.

SQL Plan Cache - Show Statements

By right-clicking the SQL Plan Cache icon, a series of options are shown which allow different views of current plan cache of the database. The **SQL Plan Cache > Show Statements** option opens a screen with filtering capability. This screen provides a direct view of the current plan cache on the system.



Press the **Apply** or **Refresh** button to display the current Plan Cache statements. The information shown includes the SQL query text, last time the query ran, most expensive single instance run, total processing time consumed, total number of times run, and information about the user and job that first created the plan entry.

The information also includes several per run averages, including average runtime, average result set size and average temporary storage usage. There is an adjusted average processing time which is the average discounting any anomalous runs.

The display also shows how many times, if any, that the database engine reused the results of a prior run, avoiding rerunning the entire statement. There is also a Save Results button (not shown) that allows you to save the statement list, for example, to a .csv file or spreadsheet.

| Finally, the numeric identifier and plan score are also displayed. For more detail on the columns displayed, see “SQL Plan Cache column descriptions” on page 150

Statement Options

By highlighting one or more plans and right clicking, a menu with several possible actions appears.

Visual Explain

| Shows a visual depiction of the access plan and provides more detailed performance analysis.
| Note only one statement can be highlighted when performing this action.

Show Longest Runs

| Shows details of up to 10 of the longest running instances of that statement. Within the Longest Runs list, you can right click a statement and select **Visual Explain**, **Work With SQL Statement**, **Work With SQL Statement and Variables**, **Save to New...** snapshot or **Remove**. Snapshots are useful for capturing the information for that specific run in Visual Explain. Removing old or superfluous runs makes room to capture future runs. Only one statement can be highlighted when performing these actions. Any runs removed only affect which runs are shown in the list. The total time, total number of runs, and other information for the statement are still calculated including the runs removed from the list.

Show Active Jobs

| Displays a list of jobs on the system that are currently using that statement or statements.

Show User History

| Shows a list of all user IDs that have run that statement along with the last time they ran it.

Work with SQL Statement

| Displays a scripting window containing the SQL statement. The scripting window is useful for working with and tuning the statement directly, or for just viewing the statement in its own window. Only one statement can be highlighted when performing this action.

Work with SQL Statements and Variables

| Displays a scripting window containing the SQL Statement and any parameter markers entered with their specific values for that run of the SQL statement.

Save to New...

| Allows you to create a snapshot of the selected statements.

| **Plan** Right-click to show options for modifying the plan:

| **Change Plan Score** allows you to set the score to a specific value. The plan score is used to determine when a plan might be removed from the cache. A lower score plan is removed before a higher score plan. By setting the plan score high, the plan remains in the cache for a longer time. Setting the plan score to a low value causes the plan to be pruned sooner than might otherwise have occurred.

| **Delete** allows you to remove the plan immediately from the cache. Note under normal circumstances there might not be a need to modify the attributes of a plan. Normal database processing ages and prunes plans appropriately. These modifying options are provided mostly as tools for minute analysis and for general interest.

| The User and Job Name for each statement on the Statements screen is the user and job that created the initial plan with full optimization. This user is not necessarily the same as the last user to run that statement. The Longest Runs screen, however, does show the particular user and job for that individual run.

Filtering Options

| The screen provides filtering options which allow the user to more quickly isolate specific criteria of interest. No filters are required to be specified (the default), though adding filtering shortens the time it

l takes to show the results. The list of statements that is returned is ordered so that the statement
l consuming the most total processing time is shown at the top. You can reorder the results by clicking the
l column heading for which you want the list ordered. Repeated clicking toggles the order from ascending
l to descending.

l The filtering options provide a way to focus in on a particular area of interest:

l **Minimum runtime for the longest execution of the statement:**

l Show statements with at least one long individual statement instance runtime.

l **Statements that ran on or after this date and time:**

l Show statements that have been run recently.

l **Top 'n' most frequently run statements:**

l Show statements run most often.

l **Top 'n' statements with the largest total accumulated runtime:**

l Show the top resource consumers. Shows the first 'n' top statements by default when no filtering
l is given. Specifying a value for 'n' improves the performance of getting the first screen of
l statements, though the total statements displayed is limited to 'n'.

l **Statements the following user has ever run:**

l Show statements a particular user has run. The user and job name shown reflect the originator of
l the cached statement. This user is not necessarily the same as the user specified on the filter
l (there could be multiple users running the statement).

l **Statements that are currently active**

l Show statements that are still running or are in pseudo-close mode. The user and job name
l shown reflect the originator of the cached statement. This user is not necessarily the same as the
l user specified on the filter (there could be multiple users running the statement).

l **Note:** An alternative for viewing the active statement for a job is to right-click the Database icon
l and select **SQL Details for Jobs...**

l **Statements for which an index has been advised**

l Show only those statements where the optimizer advised an index to improve performance.

l **Statements for which statistics have been advised**

l Show only those statements where a statistic not yet collected might have been useful to the
l optimizer. The optimizer automatically collects these statistics in the background. This option is
l normally not that interesting unless, for whatever reason, you want to control the statistics
l collection yourself.

l **Include statements initiated by the operating system**

l Show the 'hidden' statements initiated by the database to process a request. By default the list
l only includes user-initiated statements.

l **Statements that reference the following objects:**

l Show statements that reference the tables or indexes specified.

l **Statements that contain the following text:**

l Show statements that include the text specified. This option is useful for finding particular types
l of statements. For example, statements with a FETCH FIRST clause can be found by specifying
l 'fetch'. The search is not case sensitive for ease of use. For example, the string 'FETCH' finds the
l same statements as the search string 'fetch'. This option provides a wildcard search capability on
l the SQL text itself.

l Multiple filter options can be specified. The candidate statements for each filter are computed
l independently. Only those statements that are present in all the candidate lists are shown. For example,
l you could specify options **Top 'n' most frequently run statements** and **Statements the following user
l has ever run**. The display shows those most frequently run statements in the cache that have been run by

the specified user. It does not show the most frequently run statements by the user (unless those statements are also the most frequently run statements in the entire cache).

SQL Plan Cache column descriptions

Displays the columns that are used in the SQL Plan Cache Statements window.

Table 44. Columns used in SQL Plan Cache Statements window

| Column name | Description |
|--|---|
| Last Time Run | Displays the last time that this statement was run. |
| Most Expensive Time (sec) | The time taken for the longest run of this statement. |
| Total Processing Time (sec) | The sum total time that all runs of this statement took to process in seconds. |
| Total Times Run | The total number of times that this statement ran. |
| Average Processing Time (sec) | The average time per run that this statement took to process in seconds. |
| Statement | The statement text. |
| Plan Creation User Name | The name of the user id that created the plan. |
| Job Name | The name of the job that created the plan. |
| Job User | The name of the user id that owned the job that created the plan. |
| Job Number | The job number of the job that created the plan. |
| Adjusted Average Processing Time (sec) | The average time per run that this statement took to process in seconds where anomalous runs are removed from the average calculation. This time provides a realistic average for a statement by ignoring a single (or few) run that was atypical to the normal condition of the statement. |
| Average Result Set Rows | The average number of result set rows that are returned when this statement is run. |
| Average Temp Storage Used (MB) | The average amount of temporary storage used when this statement is run. |
| Plan Score | The rating of this plan relative to other plans in the cache. A plan with a higher rating relative to other plans remains in the cache for a longer time. A plan with a lower rating relative to other plans is removed from the cache sooner than the other plans. |
| Plan Identifier | A unique numeric identifier of the plan. |
| Total Cached Results Used | The number of times a result set from a prior run of the statement was reused on a subsequent run of the statement. |
| Optimization Time (sec) | The amount of time that it took to optimize this statement. |
| System Name | The system name. |
| Relational Database name | Relational database name |

SQL plan cache properties

The **SQL Plan Cache > Properties** option shows high-level information about the cache. This information includes cache size, number of plans, number of full opens and pseudo-opens that have occurred.

This information can be used to view overall database activity. If tracked over time, it provides trends to help you better understand the database utilization peaks and valleys throughout the day and week.

You can edit the Plan Cache Size Threshold property of your plan cache by right-clicking a property and selecting Edit Value. Under normal circumstances, this properties value is sufficient and altering is not necessary. If it is altered, take care to assess the performance consequences of the change. Note the value is restored to its default on an IPL.

| Description | Value |
|--|---------------------------|
| Time Of Summary | 2009-08-28-13.35.43.71142 |
| Active Query Summary | |
| Number of Currently Active Queries | 316 |
| Number of Queries Run Since Start | 328058 |
| Number of Query Full Opens Since Start | 59198 |
| Plan Usage Summary | |
| Current Number of Plans in Cache | 7005 |
| Current Plan Cache Size | 364 MB |
| Plan Cache Size Threshold | 512 MB |

Buttons: Refresh, Close, Help, ?

You might be able to edit some of the properties of your plan cache by right-clicking a property and selecting **Edit Value**.

Creating SQL plan cache snapshots

The **New > Snapshot** option allows for the creation of a snapshot from the plan cache.

Unlike the snapshot option under Show Statements, it allows you to create a snapshot without having to first view the queries.

New SQL Plan Cache Snapshot - Serv...

Name:

Schema:

Include all plan cache entries

Include plan cache entries that meet the following criteria

Filters to apply:

Minimum runtime for the longest execution of the statement:
 Seconds

Statements that ran on or after this date and time:

Top 'n' most frequently run statements:

Top 'n' statements with the largest total accumulated runtime:

Statements the following user has ever run:

Statements that are currently active

Statements for which an index has been advised

Statements for which statistics have been advised

Include statements initiated by the operating system

Statements that reference the following objects:

| Schema | Name | |
|-------------------------------------|----------------------|--|
| <input type="text"/> | <input type="text"/> | <input type="button" value="Browse..."/> |
| <input type="text" value="Schema"/> | <input type="text"/> | <input type="button" value="Remove"/> |

Statements that contain the following text:

The same filtering options are provided here as on the Show Statements screen.

The stored procedure, `qsys2.dump_plan_cache`, provides the simplest, programmatic way to create a database monitor file output (snapshot) from the plan cache. The `dump_plan_cache` procedure takes two parameters, library name and file name, for identifying the resulting database monitor file. If the file does not exist, it is created. For example, to dump the plan cache to a database performance monitor file in library QGPL:

```
CALL qsys2.dump_plan_cache('QGPL','SNAPSHOT1');
```

| **SQL plan cache event monitor**

| The SQL plan cache event monitor records changes in your plan cache.

| You can access the SQL plan cache event monitor through the System i Navigator interface or by calling the procedures directly.

| The SQL plan cache event monitor captures monitor records of plans as they are removed from the plan cache. The event monitor is useful for ensuring that all performance information potentially available in the cache is captured even if plans are removed from the cache. Combining the event monitor output with a plan cache snapshot provides a composite view of the cache from when the event monitor was started until the snapshot is taken.

| The event monitor allows the same filtering options as described for **Show statements** and **NewSnapshot**. The exceptions are that the *Top 'n' most frequently run statements* and the *Top 'n' statements with largest total accumulated runtime* are not shown. Once a statement is removed from the cache, it is no longer compared to other plans. Therefore, these two "Top n" filters do not make sense for pruned plans.

Accessing the SQL plan cache with SQL stored procedures

The System i Navigator provides a visual interface into the plan cache. However, the plan cache is also accessible through stored procedures which can be called using the SQL CALL statement.

These procedures allow for programmatic access to the plan cache and can be used, for example, for scheduling plan cache captures or pre-starting an event monitor.

qsys2.dump_plan_cache('lib', 'file')

This procedure creates a snapshot (database monitor file) of the contents of the cache. It takes two parameters, library name and file name, for identifying the resulting database monitor file. If the file does not exist, it is created. The file name is restricted to 10 characters.

For example, to dump the plan cache to a database performance monitor file called SNAPSHOT1 in library QGPL:

```
CALL qsys2.dump_plan_cache('QGPL','SNAPSHOT1');
```

qsys2.start_plan_cache_event_monitor('lib', 'file')

This procedure starts an event monitor to intercept plans as they are removed from the cache and generate performance information into the specified database monitor file. It takes two parameters, library name and file name, for identifying the resulting database monitor file.

If the file does not exist, it is created. Initially the file is created and populated with the starting record id 3018 (column QQRID = 3018). Control returns to the caller but the event monitor stays active. Library QTEMP is not allowed. The file name is restricted to 10 characters.

The event monitor stays active until one of the following occurs:

- it is ended by one of the end event monitor procedure calls.
- it is ended using the System i Navigator interface.
- an IPL (Initial Program Load) of the system occurs.

- the specified database monitor file is deleted or otherwise becomes unavailable.

For example, to start an event monitor and place plan information into a database performance monitor file called PRUNEDP1 in library QGPL:

```
CALL qsys2.start_plan_cache_event_monitor('QGPL','PRUNEDP1');
```

qsys2.start_plan_cache_event_monitor('lib', 'file', monitorID)

This procedure starts an event monitor to capture plans as they are removed from the cache and generate performance information into a database monitor file. It takes three parameters, library name, file name, and monitorID. The library name and file name identify the resulting database monitor file.

If the file does not exist, it is created. Initially the file is created and populated with the starting record id 3018. The monitorID is a CHAR(10) output parameter set by the database to contain the 10 character identification of the event monitor that was started. Control returns to the procedure caller but the event monitor stays active. Library QTEMP is not allowed. The file name is restricted to 10 characters.

The event monitor stays active until one of the following occurs:

- it is ended by one of the end event monitor procedure calls.
- it is ended using the System i Navigator interface.
- an IPL (Initial Program Load) of the system occurs.
- the specified database monitor file is deleted or otherwise becomes unavailable.

For example, start an event monitor to place plan information into a database performance monitor file called PRUNEDPLANS1 in library QGPL. Capture the monitor id into host variable HVmonid for use later:

```
CALL qsys2.start_plan_cache_event_monitor('QGPL','PRUNEDP1', :HVmonid);
```

qsys2.end_all_plan_cache_event_monitors()

This procedure can be used to end all active plan cache event monitors started either through the GUI or use the start_plan_cache_event_monitor procedures. It takes no parameters.

```
CALL qsys2.end_all_plan_cache_event_monitors();
```

qsys2.end_plan_cache_event_monitor('monID')

This procedure can be used to end the specific event monitor identified by the given monitor id value. This procedure works with the start_plan_cache_event_monitor to end a particular event monitor.

Example:

```
CALL qsys2.end_plan_cache_event_monitor('PLANC00001');
```

qsys2.change_plan_cache_size(sizeinMeg)

This procedure can be used to change the size of the Plan Cache. The integer parameter specifies the size in megabytes that the plan cache is set to. The size is reset to the database default at the subsequent IPL (Initial Program Load). If the value given is zero, the plan cache is reset to its default value.

Example:

```
CALL qsys2.change_plan_cache_size(512);
```

qsys2.dump_plan_cache_properties('lib', 'file')

This procedure creates a file containing the properties of the cache. It takes two parameters, library name and file name, for identifying the resulting properties file. If the file does not exist, it is created. The file name is restricted to 10 characters. The file definition matches the archive file qsys2/qdboppcgen.

For example, to dump the plan cache properties to a file called PCPROP1 in library QGPL:

```
CALL qsys2.dump_plan_cache_properties('QGPL','PCPROP1');
```

Verifying the performance of SQL applications

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

The commands that can help you verify performance:

Display Job (DSPJOB)

You can use the **Display Job (DSPJOB)** command with the OPTION(*OPNF) parameter to show the indexes and tables used by an application 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, the number of rows 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.

Print SQL Information (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, access plans used, and the command parameters used to precompile the source member.

Start Database Monitor (STRDBMON)

You can use the **Start Database Monitor (STRDBMON)** command to capture to a file information about every SQL statement that runs.

Change Query Attribute (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.

Start Debug (STRDBG)

You can use the **Start Debug (STRDBG)** command to put a job into debug mode, and optionally add as many as 20 programs, 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.

Related information:

Display Job (DSPJOB) command

Print SQL Information (PRTSQLINF) command

Start Database Monitor (STRDBMON) command

Change Query Attributes (CHGQRYA) command

Start Debug (STRDBG) command

Examining 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 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 QRYOPLIB 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 affect all users and queries that use this table. Once you change the **MESSAGES_DEBUG** parameter, all queries that use this QAQQINI table write debug messages to their respective job logs. 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.

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 a message.

Print SQL Information

The **Print SQL Information (PRTSQLINF)** command returns information about the embedded SQL statements in a program, SQL package (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. How the query is implemented, indexes used, join order, whether a sort is used, 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.

- The CREATE PROCEDURE and CREATE FUNCTION statement text used to create external procedures or User Defined Functions.

This output is like 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.

You can issue **PRTSQLINF** in a couple of ways. First, you can run the **PRTSQLINF** command against a saved access plan. You must execute or at least prepare the query (using the SQL 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. It could well change after the first run. **PRTSQLINF**'s requirement of a saved access plan means that the command cannot be used with **OPNQRYF**.

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

Related information:

Print SQL Information (PRTSQLINF) command

Query optimization tools: Comparison

Use this table to find the information each tool can provide, when it analyzes your queries, and the tasks it can do to improve your queries.

Table 45. Optimization tool comparison

| PRTSQLINF | STRDBG or CHGQRYA | File-based monitor (STRDBMON) | Memory-Based Monitor | Visual Explain |
|--|---|--|---|---|
| Available without running query (after access plan has been created) | Only available when the query is run | Only available when the query is run | Only available when the query is run | Only available when the query is explained |
| Displayed for all queries in SQL program, whether executed or not | Displayed only for those queries which are executed | Displayed only for those queries which are executed | Displayed only for those queries which are executed | Displayed only for those queries that are explained |
| Information 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 System i Navigator Database and API interface |
| Messages are printed to spool file | Messages are 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 System i 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 |

Changing the attributes of your queries

You can modify different types of query attributes for a job with the **Change Query Attributes (CHGQRYA)** CL command. You can also use the System i Navigator Change Query Attributes interface.

Related concepts:

“Plan cache” on page 6

The plan cache is a repository that contains the access plans for queries that were optimized by SQE.

“Objects processed in parallel” on page 50

The DB2 Symmetric Multiprocessing feature provides the optimizer with additional methods for retrieving data that include parallel processing. Symmetrical multiprocessing is a form of parallelism achieved on a single system where multiple CPU and I/O processors sharing memory and disk work simultaneously toward a single result.

Related information:

Change Query Attributes (CHGQRYA) command

Controlling 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. This modification is done through the **Change Query Attributes (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:

- | • ALLOW_ADAPTIVE_QUERY_PROCESSING
- | • ALLOW_ARRAY_VALUE_CHANGES
- | • ALLOW_TEMPORARY_INDEXES
- | • APPLY_REMOTE
- | • ASYNC_JOB_USAGE
- | • CACHE_RESULTS
- | • COLLATE_ERRORS
- | • COMMITMENT_CONTROL_LOCK_LIMIT
- | • DETERMINISTIC_UDF_SCOPE
- | • FIELDPROC_ENCODED_COMPARISON
- | • FORCE_JOIN_ORDER
- | • IGNORE_LIKE_REDUNDANT_SHIFTS
- | • LIMIT_PREDICATE_OPTIMIZATION
- | • LOB_LOCATOR_THRESHOLD
- | • MATERIALIZED_QUERY_TABLE_REFRESH_AGE
- | • MATERIALIZED_QUERY_TABLE_USAGE
- | • MEMORY_POOL_PREFERENCE
- | • MESSAGES_DEBUG
- | • NORMALIZE_DATA
- | • OPEN_CURSOR_CLOSE_COUNT
- | • OPEN_CURSOR_THRESHOLD
- | • OPTIMIZATION_GOAL
- | • OPTIMIZE_STATISTIC_LIMITATION

- | • PARALLEL_DEGREE
- | • PARAMETER_MARKER_CONVERSION
- | • QUERY_TIME_LIMIT
- | • REOPTIMIZE_ACCESS_PLAN
- | • SQLSTANDARDS_MIXED_CONSTANT
- | • SQL_CONCURRENT_ACCESS_RESOLUTION
- | • SQL_DECFLOAT_WARNINGS
- | • SQL_FAST_DELETE_ROW_COUNT
- | • SQL_MODIFIES_SQL_DATA
- | • SQL_PSEUDO_CLOSE
- | • SQL_STMT_COMPRESS_MAX
- | • SQL_STMT_REUSE
- | • SQL_SUPPRESS_WARNINGS
- | • SQL_TRANSLATE_ASCII_TO_JOB
- | • SQL_XML_DATA_CCSID
- | • STAR_JOIN
- | • STORAGE_LIMIT
- | • SYSTEM_SQL_STATEMENT_CACHE
- | • TEXT_SEARCH_DEFAULT_TIMEZONE
- | • UDF_TIME_OUT
- | • VARIABLE_LENGTH_OPTIMIZATION

Specifying the QAQQINI file with CHGQRYA:

Use the **Change Query Attributes (CHGQRYA)** command with the QRYOPTLIB (query options library) parameter to specify which schema currently contains or contains the query options file QAQQINI.

The query options file is retrieved from the schema specified on the QRYOPTLIB parameter for each query. It remains in effect for the duration of the job or user session, or until the QRYOPTLIB parameter is changed by the **Change Query Attributes (CHGQRYA)** command.

If the **Change Query Attributes (CHGQRYA)** command is not issued, or is issued without the QRYOPTLIB parameter specified, QUSRSYS is searched for the QAQQINI file. If a query options file is not found, no attributes are modified. Since the system ships without an INI file in QUSRSYS, you might 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.

Related information:

Change Query Attributes (CHGQRYA) command

Creating the QAQQINI query options file:

Each system 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 **Create Duplicate Object (CRTDUPOBJ)** command. Create a copy of the QAQQINI file in the schema specified on the **Change Query Attributes (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 could be corrupted. An error is 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 is displayed six times in the job log when the **CRTDUPOBJ** CL is used to create the file. These messages are not an error; they are only informational messages.

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 highly recommended that the file QAQQINI, in QSYS, not be modified. This file is the original template that is duplicated into QUSRSYS or a user specified library for use.

Related information:

Change Query Attributes (CHGQRYA) command
Create Duplicate Object (CRTDUPOBJ) command

QAQQINI file override support:

If you find working with the QAQQINI query options file cumbersome, consider using the `qsys2.override_qaqqini()` procedure. Instead of creating, managing, and using a QAQQINI *FILE object directly, this procedure can be called to work with a temporary version of the INI file. It uses user-specified options and values. The support relies upon the QTEMP library, so any changes affect only the job which calls the procedure.

- | The CHGQRYA command requires *JOBCTL authority.
- | *JOBCTL or QIBM_DB_SQLADM function usage is required for the following three QAQQINI options.
- | See Authority Options for SQL Analysis and Tuning for more information. These three options are more restrictive than the other options because they can affect the performance of other jobs.
- | • QUERY_TIME_LIMIT
- | • STORAGE_LIMIT
- | • PARALLEL_DEGREE
- | **Note:** QUERY_TIME_LIMIT can be changed by anyone as long as the new value is 0.
- | All other options can be changed by anyone.

Example Usage

Step 1: Establish QAQQINI override (QTEMP.QAQQINI is created based upon the current QAQQINI being used in the job)

```
--
call qsys2.override_qaqqini(1, '', '');
```

Step 2: -- Choose QAQQINI parameters and values to override. (can be called repetitively)

```

-- Avoid UDF timeouts
call qsys2.override_qaqqini(2, 'UDF_TIME_OUT', '*MAX');

-- Force full opens of cursors
call qsys2.override_qaqqini(2, 'OPEN_CURSOR_THRESHOLD', '-1');

-- Avoid using the System Wide Statement Cache
call qsys2.override_qaqqini(2, 'SYSTEM_SQL_STATEMENT_CACHE', '*NO');

-- Force any saved access plans to be rebuilt
call qsys2.override_qaqqini(2, 'REBUILD_ACCESS_PLAN', '*YES');

```

Setup complete, now execute the test

Step 3: -- Discard override values and resume using the previous QAQQINI file (this step is optional)

```

--
call qsys2.override_qaqqini(3, '', '');

```

QAQQINI query options file format:

The QAQQINI file is shipped in the schema QSYS. It has a predefined format and has been pre-populated with the default values for the rows.

Query Options File:

| | | | |
|---|-----------|-------|--|
| A | | | UNIQUE |
| A | R QAQQINI | | TEXT('Query options + file') |
| A | QQPARAM | 256A | VARLEN(10) + TEXT('Query+ option parameter') + COLHDG('Parameter') |
| A | QQVAL | 256A | VARLEN(10) + TEXT('Query option + parameter value') + COLHDG('Parameter Value') |
| A | QQTEXT | 1000G | VARLEN(100) + TEXT('Query + option text') + ALWNULL + COLHDG('Query Option' + 'Text') + CCSID(13488) + DFT(*NULL) |
| A | K QQPARAM | | |

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 prints out the optimizer debug messages:

```

UPDATE MyLib/QAQQINI SET QQVAL='*YES'
WHERE QQPARAM='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 QQPARAM='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 authority allows users to view the query options file, but not change it. Changing the values of the QAQQINI file affects all queries run on the system. Allow only the system or database administrator to have *CHANGE authority to the QAQQINI query options file.

The query options file, which resides in the library specified on the **Change Query Attributes (CHGQRYA)** CL command QRYOPTLIB parameter, is always used by the query optimizer. It is used even if the user has no authority to the query options library and file. This authority provides the system administrator with an additional security mechanism.

When the QAQQINI file resides in the library QUSRSYS the query options affects all the query users on the system. To prevent anyone from inserting, deleting, or updating the query options, the system administrator must remove update authority from *PUBLIC to the file. This update authority prevents users from changing the data in the file.

A copy of the QAQQINI file can also reside in a user library. If that library is specified on the QRYOPTLIB parameter of the **Change Query Attributes (CHGQRYA)** command, the query options affect all the queries run for that user job. To prevent the query options from being retrieved from a particular library the system administrator can revoke authority to the **Change Query Attributes (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 is issued:

```
Trigger program or external routine detected an error.
```

QAQQINI query options:

There are different options available for parameters in the QAQQINI file.

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

Table 46. Query Options Specified on QAQQINI Command

| Parameter | Value | Description |
|--|----------|---|
| ALLOW_ADAPTIVE_QUERY_PROCESSING Specifies whether Adaptive Query Processing (AQP) processing is done for a query. Adaptive query processing uses runtime statistics to look for poor performing queries and potentially replace the poor plan with an improved plan. | *DEFAULT | The default value is set to *YES. |
| | *YES | Allows Adaptive query processing to occur for this query. The existing QAQQINI options that affect AQP are the following: <ul style="list-style-type: none"> • If the REOPTIMIZE_ACCESS_PLAN QAQQINI option is set to *ONLY_REQUIRED, AQP does not reoptimize the original plan. *ONLY_REQUIRED indicates the user does not want the query reoptimized unless there is a functional reason to do so. *ONLY_REQUIRED takes precedence over AQP. • Join order requirements specified by the user in the FORCE_JOIN_ORDER QAQQINI option take precedence over AQP. If the user specifies the primary table in the join order, any AQP primary recommendations will be placed after the primary table if they are different. |
| | *NO | Adaptive query processing cannot be used for this query. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|---|----------|--|
| ALLOW_ARRAY_VALUE_CHANGES Specifies whether changes to the values of array elements are visible to the query while the query is running. | *DEFAULT | The default value is set to *NO. |
| | *NO | <p>Do not allow changes to values in arrays referenced in the query to be visible after the query is opened.</p> <p>All values which could be referenced in a query are copied during query open processing. Any changes to values in arrays after the query is opened are not visible.</p> <p>Produces queries with predictable and reproducible results, but might have a performance penalty when working with large arrays or large array elements. The penalty is less if all the references to arrays are simple non-column values, for example, :ARRAY[1] or :ARRAY[:hv2].</p> <p>Use of column values from a table to index the ARRAY, or using the UNNEST() function results in copies of the entire array being made. These copies have the largest performance penalty.</p> |
| | *YES | <p>Allow changes to values in arrays to be visible to the query while the query is running. The arrays are not copied during the open processing of the query. If the array values are changed during the processing of queries, the results of the query might be unpredictable.</p> <p>Performance might be improved for queries which reference large arrays in complex array index lookup operations, such as :Array[column-name], or when using UNNEST. Large arrays include arrays that have thousands of elements, or elements with a large size. Array index lookups using simple index values, such as :ARRAY[1] or :ARRAY[:hv2], see minimal performance improvements.</p> <p>Performance of some queries might be negatively impacted. For example, later queries that could reuse the results if they were cached to avoid recalculation where the cached result is applicable.</p> <p>Procedures that can run with *YES and still expect predictable results have the following characteristics:</p> <ol style="list-style-type: none"> 1. Contain no cursor declarations. 2. Receive arrays as input parameters: <ul style="list-style-type: none"> • and do not contain SET statements which reference arrays on the left side of the SET, and • and have no SQL statements with INTO clauses referencing arrays. 3. Do not contain SET statements which reference arrays on the left side of the set: <ul style="list-style-type: none"> • and have no SQL statements with INTO clauses referencing arrays while a cursor is open for a query which references an array. |
| ALLOW_EVI_ONLY_ACCESS Specifies whether or not Encoded Vector Indexes can be used to replace table access. | *DEFAULT | The default is set to *NO |
| | *NO | Encoded Vector Indexes cannot replace table access. |
| | *YES | Encoded Vector Indexes can replace table access if an EVI exists for every column being accessed in the table. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|---|----------------|--|
| <p>ALLOW_TEMPORARY_INDEXES</p> <p>Specifies whether temporary indexes can be considered by the optimizer. If temporary indexes are not allowed, then any other viable plan is chosen regardless of cost to implement this query.</p> | *DEFAULT | The default value is set to *YES. |
| | *YES | Allow temporary indexes to be considered. |
| | *ONLY_REQUIRED | Do not allow any temporary indexes to be considered for this access plan. Choose any other implementation regardless of cost to avoid the creation of a temporary index. Only if no viable plan can be found, is a temporary index allowed. |
| <p>APPLY_REMOTE</p> <p>Specifies for database queries involving distributed files, whether the CHGQRYA query attributes are applied to the jobs on the remote systems associated with this job.</p> | *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 use the attributes associated to them on their systems. |
| | *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 system 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. |
| <p>ASYNC_JOB_USAGE</p> <p>Specifies the circumstances in which asynchronous (temp writer) jobs can be used to help process database queries in the job. The option determines which types of database queries can be used in asynchronous jobs (running in parallel) to help complete the query.</p> <p>An asynchronous job is a separate job that handles query requests from jobs running the database queries on the system. The asynchronous job processes each request and puts the results into a temporary file. This intermediate temporary file is then used by the main job to complete the database query.</p> <p>The advantage of an asynchronous job is that it processes its request at the same time (in parallel) that the main job processes another query step. The disadvantage of using an asynchronous job is that it might encounter a situation that it cannot handle in the same way as the main job. For example, the asynchronous job might receive an inquiry message from which it cancels, whereas the main job can choose to ignore the message and continue.</p> <p>There are two different types of database queries that can run asynchronous jobs:</p> <ol style="list-style-type: none"> 1. Distributed queries. These are database queries that involve distributed files. Distributed files are provided through the system feature DB2 Multi-System for IBM i. 2. Local queries. There are database queries that involve only files local to the system where the database queries are being run. | *DEFAULT | The default value is set to *LOCAL. |
| | *LOCAL | Asynchronous jobs might be used for database queries that involve only tables local to the system where the database queries are being run. In addition, this option allows the communications required for queries involving distributed tables to be asynchronous. Each system involved in the query of the distributed tables can run its portion of the query at the same time (in parallel). |
| | *DIST | Asynchronous jobs might be used for database queries that involve distributed tables. |
| | *ANY | Asynchronous jobs might 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 intersystem parallel processing occurs. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|---|---------------|--|
| CACHE_RESULTS Specifies a way for the user to control the cache results processing. For queries involving temporary results, for example, sorts or hashes, the database manager often saves the results across query pseudo-close or pseudo-open. The results are saved as long as they are not large, with the hope that they can be reused for the next run of the query. Beginning in V5R3, the database manager saves these temporary results even when a job is finished with them. The database manager assumes that another job can later reuse the results. The database manager automatically controls the caching of these results, removing cache results as storage usage becomes large. However, the amount of temporary storage used by the database can be noticeably more than in previous releases. | *DEFAULT | The default value is the same as *SYSTEM. |
| | *SYSTEM | The database manager might cache a query result set. A subsequent run of the query by the same job can reuse the cached result set. Or, if the ODP for the query has been deleted, any job can reuse the cached result set. |
| | *JOB | The database manager might cache a query result set from one run to the next for a job. Caching can occur as long as the query uses a reusable ODP. When the reusable ODP is deleted, the cached result set is destroyed. This value mimics V5R2 processing. |
| | *NONE | The database does not cache any query results. |
| COLLATE_ERRORS Specifies how data errors are handled on the GROUP BY and ORDER BY expression during hash or sort processing within queries. | *DEFAULT | The default value is *NO. |
| | *NO | A value of *NO causes the query to be ended with an error when a grouping or ordering expressions results in an error. |
| | *YES | A value of *YES indicates that the grouping or sort continues. |
| COMMITMENT_CONTROL_LOCK_LIMIT Specifies the maximum number of records that can be locked to a commit transaction initiated after setting the new value. The value specified for COMMITMENT_CONTROL_LOCK_LIMIT does not affect transactions running in jobs that have already started commitment control. For the value to be effective, it must be changed before starting commitment control. | *DEFAULT | *DEFAULT is equivalent to 500,000,000. If multiple journals are involved in the transaction, the COMMITMENT_CONTROL_LOCK_LIMIT applies to each journal, not to the transaction as a whole. For example, files F1 to F5 are journaled to journal J1, and files F6 to F10 are journaled to J2. The COMMITMENT_CONTROL_LOCK_LIMIT is set to 100,000. 100,000 record locks can be acquired for files F1 to F5. 100,000 more locks can be acquired for files F6 to F10. |
| | 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. |
| DETERMINISTIC_UDF_SCOPE Specifies the scope or lifetime of the deterministic setting for User Defined Functions (UDFs) and User Defined Table Functions (UDTFs). | *DEFAULT | The default value is *ALWAYS. |
| | *ALWAYS | The UDF is always considered deterministic. Query temporary objects might be shared across query opens and the UDF might not run for a particular query open. |
| | *OPEN | The UDF is considered deterministic only for a single instance of a query open. Query temporary objects are not shared across query open. The UDF is run at least once in the query for a given set of input parameters. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|--|-----------------|--|
| FIELDPROC_ENCODED_COMPARISON Specifies the amount of optimization that the optimizer might use when queried columns have attached field procedures | *DEFAULT | The default value is *ALLOW_EQUAL. |
| | *NONE | No optimization to remove field procedure decode option 4 or transformations to optimize field procedure invocations is allowed. For example, the optimizer cannot transform fieldProc(4, column) = 'literal' to column = fieldProc(0, 'literal'). This option is used when the field procedure is not deterministic. |
| | *ALLOW_EQUAL | Optimization allowed for equal and not equal predicates, GROUP BY, and DISTINCT processing. For example, the optimizer might choose to change the predicate fieldProc(4, column) = 'literal' to column = fieldProc(0, 'literal') in order to facilitate index matching. This option is useful when the field procedure is deterministic but no ordering can be determined based on the result of the field encoding. |
| | *ALLOW_RANGE | Transformation allowed for MIN, MAX grouping functions, ORDER BY, and all predicates except LIKE in addition to the transformations supported by *ALLOW_EQUAL. This option is useful when the field procedure is deterministic and the encoded value implies ordering |
| | *ALL | Transformation allowed for all predicates including LIKE, in addition to the transformations supported by *ALLOW_RANGE. |
| FORCE_JOIN_ORDER Specifies to the query optimizer that the join of files is to occur in the order specified in the query. | *DEFAULT | The default is set to *NO. |
| | *NO | Allow the optimizer to reorder join tables. |
| | *SQL | Only force the join order for those queries that use the SQL JOIN syntax. This option mimics the behavior for the optimizer before V4R4M0. |
| | *PRIMARY nnn | Only force the join position for the file listed by the numeric value nnn into the primary position (or dial) for the join. nnn is optional and defaults to 1. The optimizer then determines the join order for all the remaining files based upon cost. |
| | *YES | Do not allow the query optimizer to specify the order of join tables as part of its optimization process. The join occurs in the order in which the tables were specified in the query. |
| IGNORE_LIKE_REDUNDANT_SHIFTS Specifies whether redundant shift characters are ignored for DBCS-Open operands when processing the SQL LIKE predicate or OPNQRYF command %WLDCRD built-in function. | *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. The optimizer cannot use 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 might be ignored for DBCS-Open operands. These characters are ignored depending on whether an index is used to perform key row positioning for these predicates. This option enables the query optimizer to consider key row positioning for SQL LIKE or OPNQRYF %WLDCRD predicates involving DBCS-Open, DBCS-Either, or DBCS-Only operands. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|---|--|---|
| LIMIT_PREDICATE_OPTIMIZATION Specifies that the query optimizer can only use simple isolatable predicates (OIF) when performing its index optimization. An OIF is a predicate that can eliminate a record without further evaluation. Any predicate that cannot be classified as an OIF is ignored by the optimizer and needs to be evaluated as a non-key selection predicate. A=10 and (A => 10 AND B=9) are OIFs. A=10 OR B=9 are not OIFs. Note: *YES impairs or limits index optimization. | *DEFAULT | Do not eliminate the predicates that are not simple isolatable predicates (OIF) when doing index optimization. Same as *NO. |
| | *NO | Do not eliminate the predicates that are not simple isolatable predicates (OIF) when doing index optimization. |
| | *YES | Eliminate the predicates that are not simple isolatable predicates (OIF) when doing index optimization. |
| LOB_LOCATOR_THRESHOLD Specifies either *DEFAULT or an Integer Value -- the threshold to free eligible LOB locators that exist within the job. | *DEFAULT | The default value is set to 0. This option indicates that the database does not free locators. |
| | Integer Value | If the value is 0, then the database does not free locators. For values 1 through 250,000, on a FETCH request, the database compares the SQL current LOB locator count for the job against the threshold value. If the locator count is greater than or equal to the threshold, the database frees host server created locators that have been retrieved. This option applies to all host server jobs (QZDASOINIT) and has no impact to other jobs. |
| MATERIALIZED_QUERY_TABLE_REFRESH_AGE Specifies the usage of materialized query tables in query optimization and runtime. | *DEFAULT | The default value is set to 0. |
| | 0 | No materialized query tables can be used. |
| | *ANY | Any tables indicated by the MATERIALIZED_QUERY_TABLE_USAGE INI parameter can be used. |
| MATERIALIZED_QUERY_TABLE_USAGE Specifies the ability to examine which materialized query tables are eligible to be used based on the last time a REFRESH TABLE statement was run. | Timestamp duration | Only tables indicated by MATERIALIZED_QUERY_TABLE_USAGE INI option which have a REFRESH TABLE performed within the specified timestamp duration can be used. |
| | *NONE | Materialized query tables cannot be used in query optimization and implementation. |
| | *ALL | User-maintained materialized query tables may be used. |
| | *USER | User-maintained materialized query tables can be used. |
| MEMORY_POOL_PREFERENCE Specifies the preferred memory pool that database operations uses. This option does not guarantee use of the specified pool, but directs database to perform its paging into this pool when supported by the database operation. | *DEFAULT | The default value is set to *JOB. |
| | *JOB | Paging is done in the pool of the job. This option is normal paging behavior. |
| | *BASE | Attempt to page storage into the base pool when paging is needed and a database operation that supports targeted paging occurs. |
| | nn | Attempt to page storage into pool nn when paging is needed and a database operation that supports targeted paging occurs. |
| | *NAME PoolName | Attempt to page storage into a named storage pool when paging is needed and a database operation that supports targeted paging occurs. |
| | *PRIVATE Library/ Subsystem/ PoolNumber | Attempt to page storage into a private storage pool in specified library and subsystem when paging is needed and a database operation that supports targeted paging occurs. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|---|---------------|--|
| MESSAGES_DEBUG Specifies whether Query Optimizer debug messages are displayed to the job log. These messages are regularly issued when the job is in debug mode. | *DEFAULT | The default is set to *NO. |
| | *NO | No debug messages are to be displayed. |
| | *YES | Issue all debug messages that are generated for STRDBG. |
| NORMALIZE_DATA Specifies whether normalization is performed on Unicode constants, host variables, parameter markers, and expressions that combine strings. | *DEFAULT | The default is set to *NO. |
| | *NO | Unicode constants, host variables, parameter markers, and expressions that combine strings is not normalized. |
| | *YES | Unicode constants, host variables, parameter markers, and expressions that combine strings is normalized |
| OPEN_CURSOR_CLOSE_COUNT Specifies either *DEFAULT or an Integer Value: the number of cursors to full close when the threshold is encountered. | *DEFAULT | *DEFAULT is equivalent to 0. See Integer Value for details. |
| | Integer Value | <p>This value determines the number of cursors to be closed. The valid values for this parameter are 1 - 65536. The value for this parameter is less than or equal to the number in the OPEN_CURSOR_THRESHOLD parameter.</p> <p>If the number of open cursors reaches the value specified by the OPEN_CURSOR_THRESHOLD, pseudo-closed cursors are hard (fully) closed. The least recently used cursors are closed first.</p> <p>This value is ignored if OPEN_CURSOR_THRESHOLD is *DEFAULT. If OPEN_CURSOR_THRESHOLD is specified and the value is *DEFAULT, the number of cursors closed is equal to OPEN_CURSOR_THRESHOLD multiplied by 10 percent. The result is rounded up to the next integer value.</p> <p>OPEN_CURSOR_CLOSE_COUNT is used with OPEN_CURSOR_THRESHOLD to manage the number of open cursors within a job. Open cursors include pseudo-closed cursors.</p> |
| OPEN_CURSOR_THRESHOLD Specifies either *DEFAULT or an Integer Value -- the threshold to start full close of pseudo-closed cursors. | *DEFAULT | *DEFAULT is equivalent to 0. See Integer Value for details. |
| | Integer Value | <p>This value determines the threshold to start full close of pseudo-closed cursors. When the number of open 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.</p> <p>The valid user-entered values for this parameter are 1 - 65536. A default value of 0 indicates that there is no threshold. Hard closes are not forced based on the number of open cursors within a job.</p> <p>OPEN_CURSOR_THRESHOLD is used with OPEN_CURSOR_CLOSE_COUNT to manage the number of open cursors within a job. Open cursors include pseudo-closed cursors.</p> |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|---|---|--|
| OPTIMIZATION_GOAL Specifies the goal that the query optimizer uses when making costing decisions. | *DEFAULT | Optimization goal is determined by the interface (ODBC, SQL precompiler options, OPTIMIZE FOR nnn ROWS clause). |
| | *FIRSTIO | All queries are optimized with the goal of returning the first page of output as fast as possible. This option works well when the output is controlled by a user likely to cancel the query after viewing the first page of data. Queries coded with OPTIMIZE FOR nnn ROWS honor the goal specified by the clause. |
| | *ALLIO | All queries are optimized with the goal of running the entire query to completion in the shortest amount of elapsed time. This option is better when the output of a query is written to a file or report, or the interface is queuing the output data. Queries coded with OPTIMIZE FOR nnn ROWS honor the goal specified by the clause. |
| OPTIMIZE_STATISTIC_LIMITATION Specifies limitations on the statistics gathering phase of the query optimizer. One of the most time consuming aspects of query optimization is in gathering statistics from indexes associated with the queried tables. Generally, the larger the size of the tables involved in the query, the longer the gathering phase of statistics takes. This option provides the ability to limit the amount of resources spend during this phase of optimization. The more resources spent on statistics gathering, the more accurate (optimal) the optimization plan is. | *DEFAULT | The amount of time spent in gathering index statistics is determined by the query optimizer. |
| | *NO | No index statistics are gathered by the query optimizer. Default statistics are used for optimization. (Use this option sparingly.) |
| | *PERCENTAGE integer value | Specifies the maximum percentage of the index that is searched while gathering statistics. Valid values for are 1 - 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 does not gather statistics and uses default values. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|---|--|---|
| <p>PARALLEL_DEGREE</p> <p>Specifies the parallel processing option that can be used when running database queries and database file keyed access path builds, rebuilds, and maintenance in the job. The specified parallel processing option determines the types of parallel processing allowed. There are two types of parallel processing:</p> <ol style="list-style-type: none"> 1. Input/Output (I/O) parallel processing. With I/O parallel processing, the database manager uses multiple tasks for each query to do the I/O processing. The central processor unit (CPU) processing is still done serially. 2. Symmetric Multiprocessing (SMP). SMP assigns both CPU and I/O processing to tasks that run the query in parallel. Actual CPU parallelism requires a system with multiple processors. SMP can only be used if the system feature, DB2 Symmetric Multiprocessing, is installed. Use of SMP parallelism can affect the order in which records are returned. | *DEFAULT | The default value is *SYSVAL. |
| | *SYSVAL | Set to the current system value QQRYDEGREE. |
| | *IO | Any number of tasks can be used. SMP parallel processing is not allowed. |
| | *OPTIMIZE | <p>Any number of tasks for:</p> <ul style="list-style-type: none"> • I/O or SMP parallel processing of the query • database file keyed access path build, rebuild, or maintenance. <p>SMP parallel processing is used only if the system feature, DB2 Symmetric Multiprocessing for IBM i, is installed.</p> <p>Use of parallel processing and the number of tasks used is determined by:</p> <ul style="list-style-type: none"> • the number of processors available in the system • the job share of the amount of active memory available in the pool in which the job is run • 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. <p>The query optimizer chooses an implementation that minimizes elapsed time based on the job share of the memory in the pool.</p> |
| | *OPTIMIZE nnn | <p>Like *OPTIMIZE, with the value nnn indicating a percentage from 1 to 200, used to influence the number of tasks. If not specified, 100 is used.</p> <p>The query optimizer determines the parallel degree for the query using the same processing as is done for *OPTIMIZE. Once determined, the optimizer adjusts the actual parallel degree used for the query by the percentage given.</p> <p>Allows the user to override the parallel degree used without having to specify a particular parallel degree under *NUMBER_OF_TASKS.</p> |
| nnn | <p>The query optimizer chooses to use either I/O or SMP parallel processing to process the query. SMP parallel processing is used only if the system feature, DB2 Symmetric Multiprocessing for IBM i, is installed.</p> <p>nnn is a percentage from 1 to 200 and is used to influence the number of tasks. If not specified, 100 is used.</p> <p>The choices made by the query optimizer are like those choices made for parameter value *OPTIMIZE. The exception is the assumption that all pool active memory can be used for query processing, database file keyed access path build, rebuild, or maintenance.</p> | |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|--|-------------------------|---|
| PARALLEL_DEGREE (continued) | *NONE | No parallel processing is allowed for database query processing or database table index build, rebuild, or maintenance. |
| | *NUMBER_OF_TASKS nnn | Indicates the maximum number of tasks that can be used for a single query. The number of tasks is limited to either this value or the number of disk arms associated with the table. Not recommended if running SQE. The SQE optimizer attempts to use this degree and override many of the normal costing mechanisms. For SQE, use *OPTIMIZE with a percentage. |
| | *MAX xxx | Like *MAX, with the value xxx indicating the ability to specify an integer percentage value 1 - 200. The query optimizer determines the parallel degree for the query using the same processing as is done for *MAX. Once determined, the optimizer adjusts the actual parallel degree used for the query by the percentage given. This option provides the user the ability to override the parallel degree used to some extent without having to specify a particular parallel degree under *NUMBER_OF_TASKS. |
| PARAMETER_MARKER_CONVERSION Specifies whether to allow literals to be implemented as parameter markers in dynamic SQL queries. | *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 Specifies a time limit for database queries allowed to be started based on the estimated number of elapsed seconds that the query requires to process. | *DEFAULT | The default value is set to *SYSVAL. |
| | *SYSVAL | The query time limit for this job is 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 are greater than this value, the query is not started. Valid values range from 0 to 2,147,352,578. |
| REOPTIMIZE_ACCESS_PLAN Specifies whether the query optimizer reoptimizes a query with a saved access plan. Queries can have a saved access plan stored in the associated storage of an HLL program, or in the plan cache managed by the optimizer itself. Note: If you specify *NO the query could still be revalidated. Some of the reasons this option might be necessary are: <ul style="list-style-type: none"> • The queried file was deleted and recreated. • The query was restored to a different system than the one on which it was created. • An OVRDBF command was used. | *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 is optimized. |
| | *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 option could mean that you might not get all the performance benefits that a reoptimization plan might 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. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|---|---------------|---|
| SQLSTANDARDS_MIXED_CONSTANT Specifies whether to allow IGC constants to always be treated as IGC-OPEN in SQL queries. Note: When *NO is specified, DB2 for i is not compatible with the other DB2 platforms. | *DEFAULT | The default value is set to *YES. |
| | *YES | SQL IGC constants are treated as IGC-OPEN constants. |
| | *NO | If the data in the IGC constant only contains shift-out DBCS-data shift-in, then the constant are treated as IGC-ONLY, otherwise it is treated as IGC-OPEN. |
| SQL_CONCURRENT_ACCESS_RESOLUTION Specifies the concurrent access resolution to use for an SQL query. | *DEFAULT | The default value is set to *WAIT. |
| | *WAIT | The database manager must wait for the commit or rollback when encountering data in the process of being updated, deleted, or inserted. Rows encountered that are in the process of being inserted are not skipped. This option applies if possible when the isolation level in effect is Cursor Stability or Read Stability and is ignored otherwise. |
| | *CURCMT | The database manager can use the currently committed version of the data for read-only scans when it is in the process of being updated or deleted. Rows in the process of being inserted can be skipped. This option applies if possible when the isolation level in effect is Cursor Stability and is ignored otherwise. |
| SQL_DECFLOAT_WARNINGS Specifies the warnings returned for SQL DECFLOAT computations and conversions involving: <ul style="list-style-type: none"> • division by 0. • overflow. • underflow. • an invalid operand. • an inexact result. • | *DEFAULT | The default value is set to *NO. |
| | *YES | A warning is returned to the caller for DECFLOAT computations and conversions involving division by 0, overflow, underflow, invalid operand, inexact result, or subnormal number. |
| | *NO | An error or a mapping error is returned to the caller for DECFLOAT computations and conversions involving division by 0, overflow, underflow, or an invalid operand. A warning or error is not returned for an inexact result or a subnormal number. |
| SQL_FAST_DELETE_ROW_COUNT Specifies how the delete is implemented by the database manager. This value is used when processing a DELETE FROM table-name SQL statement without a WHERE clause. | *DEFAULT | The default value is set to 0. 0 indicates that the database manager chooses how many rows to consider when determining whether fast delete could be used instead of traditional delete. When using the default value, the database manager will most likely use 1000 as a row count. This means that using the INI option with a value of 1000 results in no operational difference from using 0 for the option. |
| | *NONE | This value forces the database manager to never attempt to fast delete on the rows. |
| | *OPTIMIZE | This value is same as using *DEFAULT. |
| | Integer Value | Specifying a value for this option allows the user to tune the behavior of DELETE. The target table for the DELETE statement must match or exceed the number of rows specified on the option for fast delete to be attempted. A fast delete does not write individual rows into a journal. The valid values are 1 - 999,999,999,999,999. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|--|---------------|--|
| SQL_MODIFIES_SQL_DATA | *DEFAULT | The default value is set to *NO. |
| From the SQL Standard, no MODIFIES SQL DATA operations are allowed in an SQL BEFORE trigger. | *NO | No MODIFIES SQL DATA operations are allowed in an SQL BEFORE trigger. |
| The Informix® database allows MODIFIES SQL DATA operations in SQL BEFORE triggers. Setting the option to *YES allows SQL BEFORE triggers to perform the SQL MODIFIES SQL DATA operations. | *YES | MODIFIES SQL DATA operations are allowed in an SQL BEFORE trigger. |
| SQL_PSEUDO_CLOSE | *DEFAULT | The default behavior depends upon whether the QSQPSCLS1 *DTAARA exists. If the QSQPSCLS1 *DTAARA was found on the first OPEN within the job, then SQL cursors are marked as candidates for reuse. The cursors are pseudo-closed on the first close. If the QSQPSCLS1 *DTAARA was not found on the first OPEN within the job, then SQL cursors are marked as candidates for reuse. The cursors are pseudo-closed on the second close. |
| Before V6R1: SQL cursor open processing checks for the presence of a data area named QSQPSCLS1 in the library list of the job. If the data area is found, all reusable cursors are marked as candidates for reuse. They are pseudo-closed the first time rather than the second time the application closes the cursor. Without this data area, a cursor does not become reusable until the second close. Pseudo-closing the first time results in leaving some cursors open that might not be reused. These open cursors can increase the amount of auxiliary and main storage required for the application. The storage can be monitored using the WRKSYSSTS command. For the amount of auxiliary storage used, look at the "% system ASP used." For the amount of main storage, examine the faulting rates on the WRKSYSSTS display. | | |
| The format and the contents of the data area are not important. The data area can be deleted using the following command: DLTDTAARA DTAARA(QGPL/QSQPSCLS1). | Integer Value | Specifies a value greater than zero that indicates when a cursor is pseudo-closed. The value of this option minus 1 indicates how many times the cursor is hard closed before being marked as candidate for pseudo-close. Valid values are 1 - 65535. |
| The existence of the data area is checked during the first SQL open operation for each job. It is checked only once and the processing mode remains the same for the life of the job. Because the library list is used for the search, the change can be isolated to specific jobs. Create the data area in a library that is included only in the library lists for those jobs. | | |
| SQL_STMT_COMPRESS_MAX | *DEFAULT | The default value is set to 2. The default indicates that the access plan associated with any statement will be removed after a statement has been compressed twice without being executed. |
| Specifies the compression maximum setting, which is used when statements are prepared into a package. | Integer Value | The integer value represents the number of times that a statement is compressed before the access plan is removed to create more space in the package. Executing the SQL statement resets the count for that statement to 0. The valid Integer values are 1 - 255. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|--|---------------|---|
| SQL_STMT_REUSE Specifies the number of times the statement must be reused before the statement is stored in the SQL extended dynamic package. If the number of times the statement has been reused is less than the specified INI option, a temporary copy of the statement is used. Any other job preparing the statement does a complete prepare. | *DEFAULT | The default value is 3. The statement is stored on the first prepare of the statement. |
| | Integer Value | The integer value represents the number of times the statement must be reused before the statement is stored in the SQL package. The valid integer values are 1 - 255. Before V6R1M0, the default behavior was to store the statement in the package on the first prepare. To revert to using this previous behavior, use a value of 0 for the SQL_STMT_REUSE. The default value was changed to 3 to reduce package growth by preventing seldom reused statements from being contained within the package. |
| SQL_SUPPRESS_WARNINGS For SQL statements, this parameter provides the ability to suppress SQL 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 ' '. Warnings: <ul style="list-style-type: none"> • SQL0335 • SQL0030 • SQL7909 (on a DROP PROCEDURE/ROUTINE/FUNCTION) |
| | *NO | Specifies that SQL warnings are returned to the caller. |
| SQL_TRANSLATE_ASCII_TO_JOB Specifies whether to translate SQL statement text on the application server (AS) according to the CCSID of the job. This option applies when using DRDA to connect to an IBM i as the AS where the application requestor (AR) machine is an ASCII-based platform. | *DEFAULT | The default value is set to *NO. |
| | *YES | Translate ASCII SQL statement text to the CCSID of the IBM i job. |
| | *NO | Translate ASCII SQL statement text to the EBCDIC CCSID associated with the ASCII CCSID. |
| SQL_XML_DATA_CCSD Specifies the CCSID to be used for XML columns, host variables, parameter markers, and expressions, if not explicitly specified. See "SQL_XML_DATA_CCSD QAQQINI option" on page 177 | *DEFAULT | The default value is set to 1208. |
| | *JOB | The job CCSID is used for XML columns, host variables, parameter markers, and expressions, if not explicitly specified. If the job CCSID is 65535, the default CCSID of 1208 is used. |
| | Integer Value | The CCSID used for XML columns, host variables, parameter markers, and expressions, if not explicitly specified. This value must be a valid single-byte or mixed EBCDIC CCSID or Unicode CCSID. The value cannot be 65535. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|--|---------------|--|
| STAR_JOIN Note: Only modifies the environment for the Classic Query Engine. Specifies enhanced optimization for hash queries where both a hash join table and a Distinct List of values is constructed from the data. This Distinct List of values is appended to the selection against the primary table of the hash join Any EVI indexes built over these foreign key columns can be used to perform bitmap selection against the table before matching the join values. The use of this option does not guarantee that star join is chosen by the optimizer. It only allows the use of this technique if the optimizer has decided to implement the query by using a hash join. | *DEFAULT | The default value is set to *NO |
| | *NO | The EVI Star Join optimization support is not enabled. |
| | *COST | Allow query optimization to cost the usage of EVI Star Join support. The optimizer determines whether the Distinct List selection is used based on how much benefit can be derived from using that selection. |
| STORAGE_LIMIT Specifies a temporary storage limit for database queries. If a query is expected to use more than the specified amount of storage, the query is not allowed to run. The value specified is in megabytes. | *DEFAULT | The default value is set to *NOMAX. |
| | *NOMAX | Never stop a query from running because of storage concerns. |
| | Integer Value | The maximum amount of temporary storage in megabytes that can be used by a query. This value is checked against the estimated amount of temporary storage required to run the query as calculated by the query optimizer. If the estimated amount of temporary storage is greater than this value, the query is not started. Valid values range from 0 through 2147352578. |
| SYSTEM_SQL_STATEMENT_CACHE Specifies whether to disable the system-wide SQL Statement Cache for SQL queries. | *DEFAULT | The default value is set to *YES. |
| | *YES | Examine the system-wide SQL Statement Cache when an SQL prepare request is processed. If a matching statement exists in the cache, use the results of that prepare. This option allows the application to potentially have better performing prepares. |
| | *NO | Specifies that the system-wide SQL Statement Cache is not examined when processing an SQL prepare request. |
| TEXT_SEARCH_DEFAULT_TIMEZONE Specifies the time zone to apply to any date or dateTime value specified in an XML text search using the CONTAINS or SCORE function. The time zone is the offset from UTC (Greenwich mean time). It is only applicable when a specific time zone is not given for the value. | *DEFAULT | Use the default as defined by database. This option is equivalent to UTC. |
| | sHH:MM | A time zone formatted value where <ul style="list-style-type: none"> • s is the sign, + or - • HH is the hour • MM is the minute The valid range for HH is 00 - 23. The valid range for MM is 00 - 59. The format is specific. All values are required, including sign. If HH or MM is less than 10, it must have a leading zero specified. |
| UDF_TIME_OUT Note: Only modifies the environment for the Classic Query Engine. Specifies the amount of time, in seconds, that the database waits for a User Defined Function (UDF) to finish processing. | *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 waits for the UDF to finish. |
| | integer value | Specify the number of seconds that the database waits for a UDF to finish. If the value given exceeds the database maximum wait time, the maximum wait time is used by the database. Minimum value is 1 and maximum value is system defined. |

Table 46. Query Options Specified on QAQQINI Command (continued)

| Parameter | Value | Description |
|--|----------|---|
| VARIABLE_LENGTH_OPTIMIZATION Specifies whether aggressive optimization techniques are used on variable length columns. | *DEFAULT | The default value is set to *YES. |
| | *YES | Enables aggressive optimization of variable-length columns, including index-only access. It also allows constant value substitution when an equal predicate is present against the columns. As a consequence, the length of the data returned for the variable-length column might not include any trailing blanks that existed in the original data. As a result, the application can receive the substituted value back instead of the original data. Function calls could operate on the substituted value instead of the original string value. |
| | *NO | Do not allow aggressive optimization of variable length columns. |

SQL_XML_DATA_CCSID QAQQINI option:

The SQL_XML_DATA_CCSID QAQQINI option has several settings that affect SQL processing.

The SQL_XML_DATA_CCSID QAQQINI setting is applied within SQL in the following SQL processing:

Table 47. SQL_XML_DATA_CCSID setting application within SQL

| SQL Processing item | Description |
|---|---|
| Valid values for the QAQQINI option are CCSIDs allowed on an XML column. | Valid values are all EBCDIC SBCS and mixed CCSIDs, and Unicode 1208, 1200, and 13488 CCSIDs. |
| Does not affect the promotion of SQL data types. | Other SQL data types cannot be directly promoted to the SQL XML data type. |
| XMLPARSE untyped parameter markers. | The QAQQINI setting applies to untyped parameter markers passed as string-expression. The type is CLOB(2G) for SBCS, mixed, and UTF-8 values. The type is DBCLOB(1G) for Unicode 1200 and 13488. |
| XMLCOMMENT, XMLTEXT, XMLPI untyped parameter markers. | The QAQQINI setting applies to untyped parameter markers passed as string-expression. The type is VARCHAR(32740) for SBCS, mixed, and UTF-8 values. The type is VARGRAPHIC(16370) for Unicode 1200 and 13488. |
| Applies to parameter marker casts to the XML type for XMLCONCAT, and XMLDOCUMENT. | Applies to an untyped parameter marker passed as an XML-expression. Unless an explicit CCSID clause is specified, the CCSID of the parameter marker is obtained from the QAQQINI setting. |
| The QAQQINI setting does not affect storage and retrieval assignment rules. | The CCSID of the host variables and table columns apply. |
| String to column assignment on SQL INSERT and UPDATE. | An implicit or explicit XMLPARSE is required on the column assignment. |
| String to host variable assignment. | An implicit or explicit XMLSERIALIZE is required on the host variable assignment. |
| Column to column assignment. | When the target column is XML, an implicit XMLPARSE is applied if the source column is not XML. The target XML column has a defined XML CCSID. When the source column is XML, an explicit XMLSERIALIZE is required if the target column is not XML. |
| Host variable to column assignment. | The target column has a defined CCSID. |
| UNION ALL (if XML publishing functions in query). | The XML result CCSID is obtained from the QAQQINI setting. |

Table 47. SQL_XML_DATA_CCSID setting application within SQL (continued)

| SQL Processing item | Description |
|---|---|
| Does not apply to SQL constants. | UX constants are defined as UTF-16. FX constants are defined as UTF-8. |
| Result type of XML data built-in functions. | If the first operand of XMLPARSE and XMLVALIDATE is an untyped parameter marker, the CCSID is set from the QAQQINI setting, which then affects the XML result CCSID. The QAQQINI setting is used for XMLSERIALIZE for CHAR, VARCHAR, and LOB AS data-type. UTF-16 is used for GRAPHIC, DBCLOB, and NCHAR. |
| Result type of XML publishing functions - XMLAGG, XMLGROUP, XMLATTRIBUTES, XMLCOMMENT, XMLCONCAT, XMLDOCUMENT, XMELEMENT, XMLFOREST, XMLNAMESPACES, XMLPI, XMLROW, and XMLTEXT. | The XML result CCSID for XML publishing functions is obtained from the QAQQINI setting. |
| Result type of XML publishing functions in a view. | The XML result CCSID is set when the view is created. |
| XML data type on external procedure XML AS parameters. | The XML parameter CCSID is set when the procedure is created. |
| XML data type on external user-defined functions. | The XML parameter and result CCSID are set when the function is created. |
| CREATE TABLE XML column. | The QAQQINI setting is used for dynamic SQL. The QAQQINI setting is set in *PGM, *SRVPGM, and *SQLPKG objects when created. |
| MQTs containing select-statement with XML publishing functions. | The CCSID is set when the MQT is created. The CCSID is maintained for an ALTER TABLE. |
| ALTER TABLE ADD MATERIALIZED QUERY definition. | The QAQQINI setting is used if the select-statement contains XML publishing functions. |
| XML AS CLOB CCSID | The QAQQINI setting is built into *PGM and *SRVPGM objects when the program is created. The CCSID defaults to UTF-8 for CLOB when QAQQINI setting is UTF-16 or UCS2. |
| XML AS DBCLOB CCSID | The default for DBCLOB is always UTF-16 for XML. |
| SQL GET and SET DESCRIPTOR XML data type. | QAQQINI setting applied to XML data type. |
| SQL Global variables. | QAQQINI setting applied to global variables with the XML data type. |

Related information:

- XML values
- SQL statements and SQL/XML functions

Setting resource limits with the Predictive Query Governor

The DB2 for i Predictive Query Governor can stop the initiation of a query if the estimated run time (elapsed execution time) or estimated temporary storage 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 system. It can be used with all DB2 for i 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 system resources.

- Some CQE 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 take longer than the user wants to wait.

The governor in DB2 for i is based on two measurements:

- The estimated runtime for a query.
- The estimated temporary storage consumption for a query.

If the query estimated runtime or temporary storage usage exceed the user-defined limits, the initiation of the query can be stopped.

To define a time limit (in seconds) 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. The command language used is the first place where the optimizer attempts to find the time limit.
- Set the Query Time Limit option in the query options file. The query options file 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 **Change Query Attributes (CHGQRYA)** CL command, and set the query options file to *DEFAULT. The system value is the third place where the query optimizer attempts to find the time limit.

To define a temporary storage limit (in megabytes) for the governor to use, do the following:

- Use the Query Storage Limit (QRYSTGLMT) parameter on the **Change Query Attributes (CHGQRYA)** CL command. The command language used is the first place where the query optimizer attempts to find the limit.
- Set the Query Storage Limit option STORAGE_LIMIT in the query options file. The query options file is the second place where the query optimizer attempts to find the time limit.

The time and temporary storage values generated by the optimizer are *only* estimates. The actual query runtime might be more or less than the estimate. In certain cases when the optimizer does not have full information about the data being queried, the estimate could vary considerably from the actual resource used. In those cases, you might need to artificially adjust your limits to correspond to an inaccurate estimate.

When setting the time limit for the entire system, set it to the maximum allowable time that any query must be allowed to run. By setting the limit too low you 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 are also compared to the user-defined time limit.

You can check the inquiry message CPA4259 for the predicted runtime and storage. If the query is canceled, debug messages are still written to the job log.

You can also add the Query Governor Exit Program that is called when estimated runtime and temporary storage limits have exceeded the specified limits.

Related information:

Query Governor Exit Program

End Request (ENDRQS) command

Change Query Attributes (CHGQRYA) command

Using the Query Governor:

The resource governor works with the query optimizer.

When a user issues a request to the system to run a query, the following occurs:

1. The query access plan is created by the optimizer.
As part of the evaluation, the optimizer predicts or estimates the runtime for the query. This estimate helps determine the best way to access and retrieve the data for the query. In addition, as part of the estimating process, the optimizer also computes the estimated temporary storage usage for the query.
2. The estimated runtime and estimated temporary storage are compared against the user-defined query limit currently in effect for the job or user session.
3. If the estimates for the query are less than or equal to the specified limits, the query governor lets the query run without interruption. No message is sent to the user.
4. If the query limit is exceeded, inquiry message CPA4259 is sent to the user. The message states the estimates as well as the specified limits. Realize that only one limit needs to be exceeded; it is possible that you see that only one limit was exceeded. Also, if no limit was explicitly specified by the user, a large integer value is shown for that limit.

Note: A default reply can be established for this message so that the user does not have the option to reply. 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 run.
 - Continue and run the query even though the estimated value exceeds the associated governor limit.

Setting the resource limits for jobs other than the current job

You can set either or both resource limits for a job other than the current job. You set these limits by using the JOB parameter on the **Change Query Attributes (CHGQRYA)** command. Specify either a query options file library to search (QRYOPLIB) or a specific QRYTMLMT, or QRYSTGLMT, or both for that job.

Using the resource limits to balance system resources

After the source job runs the **Change Query Attributes (CHGQRYA)** command, effects of the governor on the target job are not dependent upon the source job. The query resource limits remain in effect for the duration of the job or user session, or until a resource limit is changed by a **Change Query Attributes (CHGQRYA)** command.

Under program control, a user might be given different limits depending on the application function performed, time of day, or system resources available. These limits provide a significant amount of flexibility when trying to balance system resources with temporary query requirements.

Cancel a query with the Query Governor:

When a query is expected to take more resources than the set 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 exceeded limit and let the query run to completion.

Control 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 **Change Job (CHGJOB)** CL command.

Changes made include the following:

- If a value of *DFT is specified for the INQMSGRPY parameter of the **Change Job (CHGJOB)** CL command, the interactive user does not see the inquiry messages. The query is canceled immediately.
- If a value of *RQD is specified for the INQMSGRPY parameter of the **Change Job (CHGJOB)** CL command, the interactive user sees the inquiry. The user must reply to the inquiry.
- If a value of *SYSRPLYL is specified for the INQMSGRPY parameter of the **Change Job (CHGJOB)** CL command, a system reply list is used to determine whether the interactive user sees the inquiry and whether a reply is necessary. 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 algorithm allows 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 adds a reply list element that causes the default reply of C to cancel requests for jobs whose user profile is 'QPGMR'.

```
ADDRPYLE SEQNBR(56) MSGID(CPA4259) CMPDTA(QPGMR      51) RPY(C)
```

The following example adds a reply list element that causes the default reply of C to cancel requests for jobs whose process name is 'QPADEV0011'.

```
ADDRPYLE SEQNBR(57) MSGID(CPA4259) CMPDTA(QPADEV0011 27) RPY(C)
```

Related information:

Change Job (CHGJOB) command

Testing performance with the query governor:

You can use the query governor to test the performance of your queries.

To test the performance of a query with the query governor, do the following:

1. Set the query time limit to zero (QRYTIMLMT(0)) using the **Change Query Attributes (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 **Print SQL Information (PRTSQLINF)** 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 process 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 method allows you to try several permutations of the query with different attributes, indexes, and syntax, or both. You can then determine what performs better through the optimizer without actually running the query to completion. This process saves on system resources because the actual query of the data is never done. If the tables to be queried contain many rows, this method represents a significant savings in system resources.

Be careful when you use this technique for performance testing, because all query requests are stopped before they are run. This caution is especially important for a CQE 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 for that intermediate step, and not for the entire query.

Related information:

Print SQL Information (PRTSQLINF) command

Change Query Attributes (CHGQRYA) command

Examples of setting query time limits:

You can set the query time limit for the current job or user session using query options file QAQQINI. Specify the QRYOPTLIB parameter on the **Change Query Attributes (CHGQRYA)** command. Use 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.

To set the query time limit for 45 seconds you can use the following **Change Query Attributes (CHGQRYA)** command:

```
CHGQRYA JOB(*) QRYTIMLMT(45)
```

This command 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 **Change Query Attributes (CHGQRYA)** command.

Assume that the query optimizer estimated the runtime for a query as 135 seconds. A message is 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 **Change Query Attributes (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 **Change Query Attributes (CHGQRYA)** command:

```
CHGQRYA JOB(123456/USERNAME/JOBNAME) QRYTIMLMT(45)
```

This command 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. The message states 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 **Change Query Attributes (CHGQRYA)** command.

To set or change the query time limit to the QQRYTIMLMT system value, use the following **Change Query Attributes (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 **Change Query Attributes (CHGQRYA)** command. This use is the default behavior for the **Change Query Attributes (CHGQRYA)** command.

Note: The query time limit can also be set in the INI file, or by using the **Change System Value (CHGSYSVAL)** command.

Related information:

Change Query Attributes (CHGQRYA) command

Change System Value (CHGSYSVAL) command

Test temporary storage usage with the query governor:

The predictive storage governor specifies a temporary storage limit for database queries. You can use the query governor to test if a query uses any temporary object, such as a hash table, sort, or temporary index.

To test for usage of a temporary object, do the following:

- Set the query storage limit to zero (QRYSTGLMT(0)) using the **Change Query Attributes (CHGQRYA)** command or in the INI file. This forces an inquiry message from the governor anytime a temporary object is used for the query. The message is sent regardless of the estimated size of the temporary object.
- Prompt for message help on the inquiry message and find the same information that you can find by running the **Print SQL Information (PRTSQLINF)** command. This command allows you to see what temporary objects were involved.

Related information:

Print SQL Information (PRTSQLINF) command

Change Query Attributes (CHGQRYA) command

Examples of setting query temporary storage limits:

The temporary storage limit can be specified either in the QAQQINI file or on the **Change Query Attributes (CHGQRYA)** command.

You can set the query temporary storage limit for a job using query options file QAQQINI. Specify the QRYOPTLIB parameter on the **Change Query Attributes (CHGQRYA)** command. Use a user library where the QAQQINI file exists with a valid value set for parameter STORAGE_LIMIT.

To set the query temporary storage limit on the **Change Query Attributes (CHGQRYA)** command itself, specify a valid value for the QRYSTGLMT parameter.

If a value is specified both on the **Change Query Attributes (CHGQRYA)** command QRYSTGLMT parameter and in the QAQQINI file specified on the QRYOPTLIB parameter, the QRYSTGLMT value is used.

To set the temporary storage limit for 100 MB in the current job, you can use the following **Change Query Attributes (CHGQRYA)** command:

```
CHGQRYA JOB(*) QRYSTGLMT(100)
```

If the user runs any query with an estimated temporary storage consumption equal to or less than 100 MB, the query runs without interruption. If the estimate is more than 100 MB, the CPA4259 inquiry message is sent by the database. 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 same limit for job 123456/USERNAME/JOBNAME use the following CHGQRYA command:

```
CHGQRYA JOB(123456/USERNAME/JOBNAME) QRYSTGLMT(100)
```

This sets the query temporary storage limit to 100 MB for job 123456/USERNAME/JOBNAME.

Note: Unlike the query time limit, there is no system value for temporary storage limit. The default behavior is to let any queries run regardless of their temporary storage usage. The query temporary storage limit can be specified either in the INI file or on the **Change Query Attributes (CHGQRYA)** command.

Related information:

Change Query Attributes (CHGQRYA) command

Controlling 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 Symmetric Multiprocessing, a feature that you can purchase. You can turn parallel processing on and off.

Even if parallelism is enabled for a system or job, the individual queries that run in a job might not actually use a parallel method. This decision might be because of functional restrictions, or the optimizer might choose a non-parallel method because it runs faster.

Queries processed with parallel access methods aggressively use main storage, CPU, and disk resources. The number of queries that use parallel processing must be limited and controlled.

Controlling system-wide parallel processing for queries:

You can use the QQRYDEGREE system value to control parallel processing for a system.

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 system. 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 Symmetric Multiprocessing feature is installed. The query optimizer chooses to use parallel processing to minimize elapsed time based on the job 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 Symmetric Multiprocessing feature is installed. The choices made by the query optimizer are like the choices made for parameter value *OPTIMIZE. The exception is that the optimizer assumes that all active memory in the pool can be used to process the query.

The default QQRYDEGREE system value is *NONE. You must change the value if you want parallel query processing as the default for jobs run on the system.

Changing this system value affects all jobs that is run or are currently running on the system 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. You can also use the SET_CURRENT_DEGREE SQL statement.

Using the Change Query Attributes (CHGQRYA) command

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 **Change Query Attributes (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 Symmetric Multiprocessing feature is installed. Use of parallel processing and the number of tasks used is determined by:

- the number of system processors available
- the job share of active memory available in the pool
- whether the expected elapsed time is limited by CPU processing or I/O resources

The query optimizer chooses an implementation that minimizes elapsed time based on the job 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 Symmetric Multiprocessing feature is installed. The choices made by the query optimizer are like the choices made for parameter value *OPTIMIZE. The exception is that 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 Symmetric Multiprocessing feature is installed.

Using a number of tasks less than the number of system processors available restricts the number of processors used simultaneously for running a query. A larger number of tasks ensures that the query is allowed to use all the processors available on the system 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 the tasks.

***SYSVAL**

Specifies that the processing option used is set to the current value of the QQRYDEGREE system value.

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

Using the SET CURRENT DEGREE SQL statement

You can use the SET CURRENT DEGREE SQL statement to change the value of the CURRENT_DEGREE special register. The possible values for the CURRENT_DEGREE special register are:

1 No parallel processing is allowed.

2 through 32767

Specifies the degree of parallelism that is used.

ANY

Specifies that the database manager can choose to use any number of tasks for either I/O or SMP parallel processing. Use of parallel processing and the number of tasks used is determined by:

- the number of system processors available
- the job share of active memory available in the pool
- whether the expected elapsed time is limited by CPU processing or I/O resources

The database manager chooses an implementation that minimizes elapsed time based on the job share of the memory in the pool.

NONE

No parallel processing is allowed.

MAX

The database manager can choose to use any number of tasks for either I/O or SMP parallel processing. MAX is like ANY except the database manager assumes that all active memory in the pool can be used.

IO Any number of tasks can be used when the database manager chooses to use I/O parallel processing for queries. SMP is not allowed.

The value can be changed by invoking the SET CURRENT DEGREE statement.

The initial value of CURRENT DEGREE comes from the CHGQRYA CL command, PARALLEL_DEGREE parameter in the current query options file (QAQQINI), or the QQRYDEGREE system value.

Related information:

Set Current Degree statement

Change Query Attributes (CHGQRYA) command

DB2 Symmetric Multiprocessing

Collecting statistics with the statistics manager

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 is current.

On many platforms, statistics collection is a manual process that is the responsibility of the database administrator. With IBM i products, 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 must always provide an answer to the questions from the Optimizer. It uses the best method available to provide the answers. For example, it could 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 can use to provide greater latitude for sizing algorithms. If the statistics manager provides a low confidence in the number of groups estimated for a grouping request, the optimizer can increase the size of the temporary hash table allocated.

Related concepts:

“Statistics manager” on page 5

In CQE, the retrieval of statistics is a function of the Optimizer. When the Optimizer needs to know information about a table, it looks at the table description to retrieve the row count and table size. If an index is available, the Optimizer might extract information about the data in the table. In SQE, the collection and management of statistics is handled by a separate component called the statistics manager. The statistics manager leverages all the same statistical sources as CQE, but adds more sources and capabilities.

Automatic statistics collection

When the statistics manager prepares its responses to the optimizer, it tracks the responses that were generated using default filter factors. Default filter factors are used when column statistics or indexes are not available. The statistics manager uses this information to automatically generate a statistic collection request for the columns. This request occurs while the access plan is written to the plan cache. If system resources allow, statistics collections occur in real time for direct use by the current query, avoiding a default answer to the optimizer.

Otherwise, as system resources become available, the requested column statistics are collected in the background. The next time the query is executed, the missing column statistics are available to the statistics manager. This process allows the statistics manager to provide more accurate information to the optimizer at that time. More statistics make it easier for the optimizer to generate a better performing access plan.

If a query is canceled before or during execution, the requests for column statistics are still processed. These requests occur if 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. 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 into one request. This grouping allows both column statistics to be created with only one pass through table T1.

One thing to note is that column statistics are usually 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. In this scenario, there might be cases where optimization time benefits from column statistics. Using column statistics to answer questions from the optimizer is more efficient than using the index data. So if query performance seems extended, you might want to verify that there are indexes over the relevant columns in your query. If so, try manually generating column statistics for these columns.

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

Automatic statistics refresh

Column statistics are not maintained when the underlying table data changes. The statistics manager determines 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 new query was optimized or because an existing plan was reoptimized.

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 are stale, the statistics manager still uses them to answer the questions from the optimizer. However, the statistics manager marks the statistics as stale in the plan cache and generates a request to refresh them.

Viewing statistics requests

You can view the current statistics requests by using System i Navigator or by using Statistics APIs.

To view requests in System i Navigator, right-click **Database** and select **Statistic Requests**. This window shows all user requested statistics collections that are pending or active. The view also shows all system requested statistics collections that are being considered, are active, or have failed. You can change the status of the request, order the request to process immediately, or cancel the request.

Related reference:

“Statistics manager APIs” on page 191

You can use APIs to implement the statistics function of System i Navigator.

Indexes and column statistics

While performing similar functions, indexes and column statistics are different.

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

One major difference between indexes and column statistics is that indexes are permanent objects that are updated when changes to the underlying table occur. Column statistics are not updated. If your data is constantly changing, the statistics manager might need to rely on stale column statistics. However, maintaining an index after each table change might use more system resources than refreshing stale column statistics after a group of changes 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 considers them for implementation. If they are candidates, the optimizer reoptimizes the query and tries to find a better implementation. However, this reoptimization is not true for column statistics. When new or refreshed column statistics are available, the statistics manager interrogates immediately. Reoptimization occurs only if the answers are different from the ones that were given before these refreshed statistics. 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 the columns in the predicates, it tries to find a combination of indexes that can be used.
3. For single column questions, it uses 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. *ALL is the default setting.

*NONE

Restricts everyone from creating statistics in the background. *NONE 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 existing column statistics requests. This background task also identifies column statistics that have been used by a plan in the plan cache. The task 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 by a user are performed by the system job QDBFSTCCOL. User-initiated immediate requests are run within the user 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 system. However, parallel processing is allowed only for immediate statistics collection if SMP is installed on the system. The job that requests the column statistics also must allow parallelism.

Related information:

Performance system values: Allow background database statistics collection

Replication of column statistics with CRTDUPOBJ versus CPYF

You can replicate column statistics with the **Create Duplicate Object (CRTDUPOBJ)** or the **Copy File (CPYF)** commands.

Statistics are not copied to new tables when using the **Copy File (CPYF)** command. If statistics are needed immediately after using this command, then you must manually generate the statistics using System i Navigator or the statistics APIs. If statistics are not needed immediately, then they could be created automatically by the system after the first touch of a column by a query.

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

Related information:

Create Duplicate Object (CRTDUPOBJ) command

Copy File (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 System i 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 System i Navigator or by using statistics APIs.

To collect statistics using System i 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 System i 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.

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

High Availability (HA) solutions

High availability solutions replicate data to a secondary system by using journal entries.

However, column statistics are 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, you might want to propagate the column statistics that were gathered on your production system. Recreate them on your backup system manually.

ISV (Independent Solution Provider) preparation

An ISV might want to deliver a customer solution that already includes column statistics frequently used in the application, rather than waiting for the automatic statistics collection to create them. 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 execute on the customer system after the initial data load takes place. The script file can be shipped as part of the application

Business Intelligence environments

In a large Business Intelligence environment, it is common for large data load and update operations to occur overnight. Column statistics are marked stale only when they are touched by the statistics manager, and then refreshed after first touch. You might want to consider refreshing the column statistics manually after loading the data.

You can do this refresh easily by toggling the system value QDBFSTCCOL to *NONE and then back to *ALL. This process causes all stale column statistics to be refreshed. It also 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. An IPL clears the plan cache.

This procedure works only if you do not delete (drop) the tables and recreate 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 can significantly change the cardinality, add new ranges of values, or change the distribution of data values. These updates can affect query performance on the first query run against the new data. On the first run of such a query, the optimizer uses stale column statistics to determine the access plan. At that point, it starts a request to refresh the column statistics.

Prior to this data update, you might want to toggle the system value `QDBFSTCCOL` to `*NONE` and back to `*ALL` or `*SYSTEM`. This toggle causes an analysis of the plan cache. The analysis includes searching for column statistics used in access plan generation, 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, the automatic column statistics collection tries to refresh every time 15% of the data is changed. This processing can be redundant since you are still updating or loading the data. In this case, you might want to block automatic statistics collection for the tables and deblock 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. Switch 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 recreated by using techniques such as applying journal entries. If you have a long interval between save operations and rely on journaling to restore your environment, consider tracking column statistics that are generated after the latest save operation.

Related information:

Performance system values: Allow background database statistics collection

Statistics manager APIs

You can use APIs to implement the statistics function of System i 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 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 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

Related reference:

“Viewing statistics requests” on page 188

You can view the current statistics requests by using System i Navigator or by using Statistics APIs.

Displaying materialized query table columns

You can display materialized query tables associated with another table using System i Navigator.

To display materialized query tables, follow these steps:

1. In the System i Navigator window, expand the system that you want to use.
2. Expand **Databases** and the database that you want to work with.
3. Expand **Schemas** and the schema that you want to work with.
4. Right-click a table and select **Show Materialized Query Tables**.

Table 48. Columns used in Show materialized query table window

| Column name | Description |
|----------------------------|--|
| Name | The SQL name for the materialized query table |
| Schema | Schema or library containing the materialized query table |
| Partition | Partition detail for the index. Possible values: <ul style="list-style-type: none"> • <blank>, which means For all partitions • For Each Partition • specific name of the partition |
| Owner | The user ID of the owner of the materialized query table. |
| System Name | System table name for the materialized query table |
| Enabled | Whether the materialized query table is enabled. Possible values are: <ul style="list-style-type: none"> • Yes • No <p>If the materialized query table is not enabled, it cannot be used for query optimization. It can, however, be queried directly.</p> |
| Creation Date | The timestamp of when the materialized table was created. |
| Last Refresh Date | The timestamp of the last time the materialized query table was refreshed. |
| Last Query Use | The timestamp when the materialized query table was last used by the optimizer to replace user specified tables in a query. |
| Last Query Statistics Use | The timestamp when the materialized query table was last used by the statistics manager to determine an access method. |
| Query Use Count | The number of instances the materialized query table was used by the optimizer to replace user specified tables in a query. |
| Query Statistics Use Count | The number of instances the materialized query table was used by the statistics manager to determine an access method. |
| Last Used Date | The timestamp when the materialized query table was last used. |
| Days Used Count | The number of days the materialized query table has been used. |
| Date Reset Days Used Count | The year and date when the days-used count was last set to 0. |
| Current Number of Rows | The total number of rows included in this materialized query table at this time. |
| Current Size | The current size of the materialized query table. |
| Last Changed | The timestamp when the materialized query table was last changed. |
| Maintenance | The maintenance for the materialized query table. Possible values are: <ul style="list-style-type: none"> • User • System |

Table 48. Columns used in Show materialized query table window (continued)

| Column name | Description |
|---------------------|--|
| Initial Data | Whether the initial data was inserted immediately or deferred. Possible values are <ul style="list-style-type: none"> Deferred Immediate |
| Refresh Mode | The refresh mode for the materialized query table. A materialized query table can be refreshed whenever a change is made to the table or deferred to a later time. |
| Isolation Level | The isolation level for the materialized query table. |
| Sort Sequence | The alternate character sorting sequence for National Language Support (NLS). |
| Language Identifier | The language code for the object. |
| SQL Statement | The SQL statement that is used to populate the table. |
| Text | The text description of the materialized query table. |
| Table | Schema and table name. |
| Table Partition | Table partition. |
| Table System Name | System name of the table. |

Managing check pending constraints columns

You can view and change constraints that have been placed in a check pending state by the system. Check pending constraints refers to a state in which a mismatch exists between a parent and foreign key in a referential constraint. A mismatch can also occur between the column value and the check constraint definition in a check constraint.

To view constraints that have been placed in a check pending state, follow these steps:

1. Expand the system name and **Databases**.
2. Expand the database that you want to work with.
3. Expand the **Database Maintenance** folder.
4. Select **Check Pending Constraints**.
5. From this interface, you can view the definition of the constraint and the rows that are in violation of the constraint rules. Select the constraint that you want to work with and then select **Edit Check Pending Constraint** from the **File** menu.
6. You can either alter or delete the rows that are in violation.

Table 49. Columns used in Check pending constraints window

| Column name | Description |
|-------------------------------------|---|
| Name of Constraint in Check Pending | Displays the name of the constraint that is in a check pending state. |
| Schema | Schema containing the constraint that is in a check pending state. |
| Type | Displays the type of constraint that is in check pending. Possible values are: <ul style="list-style-type: none"> Check constraint Foreign key constraint |
| Table name | The name of the table associated with the constraint in check pending state. |
| Enabled | Displays whether the constraint is enabled. The constraint must be disabled or the relationship taken out of the check pending state before any input/output (I/O) operations can be performed. |

Creating an index strategy

DB2 for i 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.

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

A radix index is a multilevel, hybrid tree structure that allows many 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 base table row addresses associated with the key value. The key value is used to quickly navigate to the leaf node with a few simple binary search tests.

The binary radix tree structure is good for finding a few rows because it finds a given row with a minimal amount of processing. For example, create a binary radix index over a customer number column. Then create a typical OLTP request like "find the outstanding orders for a single customer". The binary index results in fast performance. An index created over the customer number column is considered the perfect index for this type of query. The index allows the database to find 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. Many users want on demand access to the detail data. For example, they might run a report every week to look at sales data. Then they want to "drill down" for more information related to a particular problem area they found in the report. In this scenario, you cannot write all the queries in advance on behalf of the end users. Without knowing what queries might run, it is impossible to build the perfect index.

Related information:

SQL Create Index statement

Derived key index

You can use the SQL CREATE INDEX statement to create a derived key index using an SQL expression.

- | Traditionally an index could only specify column names in the key of the index over the table it was based on. With this support, an index can have an expression in place of a column name that can use built-in functions, or some other valid expression. Additionally, you can use the SQL CREATE INDEX statement to create a sparse index using a WHERE condition.

For restrictions and other information about derived indexes, see the Create Index statement and Using derived indexes.

Related reference:

"Using derived indexes" on page 219

SQL indexes can be created where the key is specified as an expression. This type of key is also referred to as a derived key.

Related information:

SQL Create Index statement

| Sparse indexes

- | You can use the SQL CREATE INDEX statement to create a sparse index using SQL selection predicates.

| Last release users were given the ability to use the SQL CREATE INDEX statement to create a sparse index using a WHERE condition. With this support, the query optimizer recognizes and considers sparse indexes during its optimization. If the query WHERE selection is a subset of the sparse index WHERE selection, then the sparse index is used to implement the query. Use of the sparse index usually results in improved performance.

| **Examples**

| In this example, the query selection is a subset of the sparse index selection and an index scan over the sparse index is used. The remaining query selection (COL3=30) is executed following the index scan.

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20
|
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| In this example, the query selection is not a subset of the sparse index selection and the sparse index cannot be used.

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
|
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20
```

| **Related reference:**

| "Using sparse indexes" on page 220

| SQL indexes can be created using WHERE selection predicates. These indexes can also be referred to as sparse indexes. The advantage of a sparse index is that fewer entries are maintained in the index. Only those entries matching the WHERE selection criteria are maintained in the index.

| **Related information:**

| SQL Create Index statement

| **Sparse index optimization:**

| An SQL sparse index is like a select/omit access path. Both the sparse index and the select/omit logical file contain only keys that meet the selection specified. For a sparse index, the selection is specified with a WHERE clause. For a select/omit logical file, the selection is specified in the DDS using the COMP operation.

| The reason for creating a sparse index is to provide performance enhancements for your queries. The performance enhancement is done by precomputing and storing results of the WHERE selection in the sparse index. The database engine can use these results instead of recomputing them for a user specified query. The query optimizer looks for any applicable sparse index and can choose to implement the query using a sparse index. The decision is based on whether using a sparse index is a faster implementation choice.

| For a sparse index to be used, the WHERE selection in the query must be a subset of the WHERE selection in the sparse index. That is, the set of records in the sparse index must contain all the records to be selected by the query. It might contain extra records, but it must contain all the records to be selected by the query. This comparison of WHERE selection is performed by the query optimizer during optimization. It is like the comparison that is performed for Materialized Query Tables (MQT).

| Besides the comparison of the WHERE selection, the optimization of a sparse index is identical to the optimization that is performed for any Binary Radix index.

| Refer to section 'Indexes and the Optimizer' for more details on how Binary Radix indexes are optimized.

| **Related concepts:**

| “Indexes & the optimizer” on page 208

| Since the optimizer uses cost based optimization, more information about the database rows and columns makes for a more efficient access plan created 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).

| **Related reference:**

| “Using sparse indexes” on page 220

| SQL indexes can be created using WHERE selection predicates. These indexes can also be referred to as sparse indexes. The advantage of a sparse index is that fewer entries are maintained in the index. Only those entries matching the WHERE selection criteria are maintained in the index.

| **Sparse index matching algorithm:**

| This topic is a generalized discussion of how the sparse index matching algorithm works.

| The selection in the query must be a subset of the selection in the sparse index in order for the sparse index to be used. This statement is true whether the selection is ANDed together, ORed together, or a combination of the two. For selection where all predicates are ANDed together, all WHERE selection predicates specified in the sparse index must also be specified in the query. The query can contain additional ANDed predicates. The selection for the additional predicates will be performed after the entries are retrieved from the sparse index. See examples A1, A2, and A3 following.

| Example A1

| In this example, the query selection exactly matches the sparse index selection and an index scan over the sparse index can be used.

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| Example A2

| In this example, the query selection is a subset of the sparse index selection and an index scan over the sparse index can be used. The remaining query selection (COL3=30) is executed following the index scan.

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| Example A3

| In this example, the query selection is not a subset of the sparse index selection and the sparse index cannot be used.

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20
```

| For selection where all predicates are ORed together, all WHERE selection predicates specified in the query, must also be specified in the sparse index. The sparse index can contain additional ORed

| predicates. All the ORed selection in the query will be executed after the entries are retrieved from the
| sparse index. See examples O1, O2, and O3 following.

| Example O1

| In this example, the query selection exactly matches the sparse index selection and an index scan over the
| sparse index can be used. The query selection is executed following the index scan.

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20 or COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20 or COL3=30
```

| Example O2

| In this example, the query selection is a subset of the sparse index selection and an index scan over the
| sparse index can be used. The query selection is executed following the index scan.

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20 or COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20
```

| Example O3

| In this example, the query selection is not a subset of the sparse index selection and the sparse index
| cannot be used.

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20 or COL3=30
```

| The previous examples used simple selection, all ANDed, or all ORed together. These examples are not
| typical, but they demonstrate how the selection of the sparse index is compared to the selection of the
| query. Obviously, the more complex the selection the more difficult it becomes to determine compatibility.

| In the next example T1, the constant 'MN' was replaced by a parameter marker for the query selection.
| The sparse index had the local selection of COL1='MN' applied to it when it was created. The sparse
| index matching algorithm matches the parameter marker to the constant 'MN' in the query predicate
| COL1=?. It verifies that the value of the parameter marker is the same as the constant in the sparse
| index; therefore the sparse index can be used.

| The sparse index matching algorithm attempts to match where the predicates between the sparse index
| and the query are not the same. An example is a sparse index with a predicate SALARY > 50000, and a
| query with the predicate SALARY > 70000. The sparse index contains the rows necessary to run the
| query. The sparse index is used in the query, but the predicate SALARY > 70000 remains as selection in
| the query (it is not removed).

| Example T1

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1='MN' or COL2='TWINS'
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=? or COL2='TWINS' or COL3='WIN'
```

| In the next example T2, the keys of the sparse index match the ORDER BY fields in the query. For the
| sparse index to satisfy the specified ordering, the optimizer must verify that the query selection is a
| subset of the sparse index selection. In this example, the sparse index can be used.

| Example T2

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL1, COL3)
| WHERE COL1='MN' or COL2='TWINS'
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL2='TWINS'
| ORDER BY COL1, COL3
```

| **Related reference:**

| “Using sparse indexes” on page 220

| SQL indexes can be created using WHERE selection predicates. These indexes can also be referred to as
| sparse indexes. The advantage of a sparse index is that fewer entries are maintained in the index. Only
| those entries matching the WHERE selection criteria are maintained in the index.

| “Details on the MQT matching algorithm” on page 80

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

| **Sparse index examples:**

| This topic shows examples of how the sparse index matching algorithm works.

| In example S1, the query selection is a subset of the sparse index selection and consequently an index
| scan over the sparse index is used. The remaining query selection (COL3=30) is executed following the
| index scan.

| Example S1

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| In example S2, the query selection is not a subset of the sparse index selection and the sparse index
| cannot be used.

| Example S2

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20
```

| In example S3, the query selection exactly matches the sparse index selection and an index scan over the
| sparse index can be used.

| Example S3

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| In example S4, the query selection is a subset of the sparse index selection and an index scan over the
| sparse index can be used. The remaining query selection (COL3=30) is executed following the index scan.

| Example S4

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| In example S5, the query selection is not a subset of the sparse index selection and the sparse index cannot be used.

| Example S5

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20
```

| In example S6, the query selection exactly matches the sparse index selection and an index scan over the sparse index can be used. The query selection is executed following the index scan to eliminate excess records from the sparse index.

| Example S6

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20 or COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20 or COL3=30
```

| In example S7, the query selection is a subset of the sparse index selection and an index scan over the sparse index can be used. The query selection is executed following the index scan to eliminate excess records from the sparse index.

| Example S7

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20 or COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20
```

| In example S8, the query selection is not a subset of the sparse index selection and the sparse index cannot be used.

| Example S8

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20 or COL3=30
```

| In the next example S9, the constant 'MN' was replaced by a parameter marker for the query selection. The sparse index had the local selection of COL1='MN' applied to it when it was created. The sparse index matching algorithm matches the parameter marker to the constant 'MN' in the query predicate COL1=?. It verifies that the value of the parameter marker is the same as the constant in the sparse index; therefore the sparse index can be used.

| Example S9

```

| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1='MN' or COL2='TWINS'
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| Where Col3='WIN' and (Col1=? or Col2='TWINS')

```

In the next example S10, the keys of the sparse index match the order by fields in the query. For the sparse index to satisfy the specified ordering, the optimizer must verify that the query selection is a subset of the sparse index selection. In this example, the sparse index can be used.

Example S10

```

| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL1, COL3)
| WHERE COL1='MN' or COL2='TWINS'
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| Where Col3='WIN' and (Col1='MN' or Col2='TWINS')
| ORDER BY COL1, COL3

```

In the next example S11, the keys of the sparse index do not match the order by fields in the query. But the selection in sparse index T2 is a superset of the query selection. Depending on size, the optimizer might choose an index scan over sparse index T2 and then use a sort to satisfy the specified ordering.

Example S11

```

| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL2, COL4)
| WHERE COL1='MN' or COL2='TWINS'
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| Where Col3='WIN' and (Col1='MN' or Col2='TWINS')
| ORDER BY COL1, COL3

```

The next example S12 represents the classic optimizer decision: is it better to do an index probe using index IX1 or is it better to do an index scan using sparse index SPR1? Both indexes retrieve the same number of index entries and have the same cost from that point forward. For example, both indexes have the same cost to retrieve the selected records from the dataspace, based on the retrieved entries/keys.

The cost to retrieve the index entries is the deciding criteria. In general, if index IX1 is large then an index scan over sparse index SPR1 has a lower cost to retrieve the index entries. If index IX1 is rather small then an index probe over index IX1 has a lower cost to retrieve the index entries. Another cost decision is reusability. The plan using sparse index SPR1 is not as reusable as the plan using index IX1 because of the static selection built into the sparse selection.

Example S12

```

| CREATE INDEX MYLIB/IX1 on MYLIB/T1 (COL1, COL2, COL3)
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| CSELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30

```

Specify PAGESIZE on index creates

You can use the PAGESIZE parameter to specify the access path logical page size used by the system when the access path is created. Use the PAGESIZE parameter when creating keyed files or indexes using the **Create Physical File (CRTPF)** or **Create Logical File (CRTLF)** commands, or the SQL CREATE INDEX statement.

The logical page size is the access path number of bytes that can be moved from auxiliary storage to the job storage pool for a page fault.

Consider using the default of *KEYLEN for this parameter, except in rare circumstances. Then the page size can be determined by the system based on the total length of the keys. When the access path is used by selective queries (for example, individual key lookup), a smaller page size is typically more efficient. When the query-selected keys are grouped in the access path with many records selected, or the access path is scanned, a larger page size is more efficient.

Related information:

Create Logical File (CRTLF) command

Create Physical File (CRTPF) command

SQL Create Index statement

General index maintenance

Whenever indexes are created and used, there is a potential for a decrease in I/O velocity due to maintenance. Therefore, 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 table indexes by creating composite (multiple column) key indexes. Composite indexes can be used for multiple different situations.
- Dropping indexes during batch inserts, updates, and deletes
- Creating in parallel. Either create indexes, one at a time, in parallel using SMP or create multiple indexes simultaneously with multiple batch jobs
- Maintaining indexes in parallel using SMP

The goal of creating indexes is to improve query performance by providing statistics and implementation choices. Maintain a reasonable balance on the number of indexes to limit maintenance overhead.

Encoded vector indexes

An encoded vector index (EVI) is used to provide fast data access in decision support and query reporting environments.

EVI's 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, EVI's 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 byte, 2 bytes or 4 bytes in size.
| By specifying INCLUDE on the create, additional aggregate values can be maintained in real time as an
| extension of the key portion of the symbol table entry. These aggregated values are over non-key data
| in the table grouped by the specified EVI key.
- 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 EVI's:

- Require less storage
- May have better build times than radix, especially if the number of unique values in the columns defined for the key is relatively small.
- Provide more accurate statistics to the query optimizer
- Considerably better performance for certain grouping types of queries
- Good performance characteristics for decision support environments.

- Can be further extended for certain types of grouping queries with the addition of INCLUDE values. Provides ready-made numeric aggregate values maintained in real time as part of index maintenance. INCLUDE values become an extension of the EVI symbol table. Multiple include values can be specified over different aggregating columns and maintained in the same EVI provided the group by values are the same. This technique can reduce overall maintenance.

Disadvantages of EVIs:

- Cannot be used in ordering.
- Use for grouping is specialized. Supports:
 - COUNT, DISTINCT requests over key columns
 - aggregate requests over key columns where all other selection can be applied to the EVI symbol table keys
 - INCLUDE aggregates
 - MIN or MAX, if aggregating value is part of the symbol table key.
- Use with joins always done in cooperation with hash table processing.
- Some additional maintenance idiosyncrasies.

Related reference:

“Encoded vector index” on page 15

An encoded vector index is a permanent object that provides access to a table. This access is done by assigning codes to distinct key values and then representing those values in a vector.

Related information:

SQL Create Index statement

SQL INCLUDE statement

How the EVI works

EVI's work in different ways for costing and implementation.

For costing, the optimizer uses the symbol table to collect metadata information about the query.

For implementation, the optimizer can use the EVI in one of the following ways:

• **Selection (WHERE clause)**

The database engine uses the vector to build a dynamic bitmap or list of selected row ids. The bitmap or list contains 1 bit for each row in the table. The bit is turned on for each selected row. Like a bitmap index, these intermediate dynamic bitmaps (or lists) can be ANDed and ORed together to satisfy a query.

For example, a user wants to see sales data for a specific region and time period. You can define an EVI over the region and quarter columns of the table. When the query runs, the database engine builds dynamic bitmaps using the two EVIs. The bitmaps are ANDed together to produce a single bitmap containing only the relevant rows for both selection criteria.

This ANDing capability drastically reduces the number of rows that the system must read and test. The dynamic bitmaps exists only as long as the query is executing. Once the query is completed, the dynamic bitmaps are eliminated.

• **Grouping or Distinct**

- l The symbol table within the EVI contains distinct values for the specified columns in the key definition. The symbol table also contains a count of the number of records in the base table that have each distinct value. Queries involving grouping or distinct, based solely on columns in the key, are candidates for a technique that uses the symbol table directly to determine the query result.

The symbol table contains only the key values and their associated counts, unless INCLUDE is specified. Therefore, queries involving column function COUNT are eligible for this technique. But queries with column functions MIN or MAX on other non-key columns are not eligible. MIN and MAX values are not stored in the symbol table.

• **EVI INCLUDE aggregates**

Including additional aggregate values further extends the ability of the symbol table to provide ready-made results. Aggregate data is grouped by the specified columns in the key definition. Therefore, aggregate data must be over columns in the table other than those columns specified as EVI key values.

For performance, these included aggregates are limited to numeric results (SUM, COUNT, AVG, VARIANCE) as they can be maintained directly from the inserted or removed row.

MIN or MAX values would occasionally require other row comparisons during maintenance and therefore are not supported with the INCLUDE keyword.

EVI symbol table only access is used to satisfy distinct or grouping requests when the query is run with commitment control *NONE or *CHG.

INCLUDE for additional aggregate values can be used in join queries. When possible, the existence of EVIs with INCLUDE aggregates causes the group by process to be pushed down to each table as necessary. See the following EVI INCLUDE grouping push down example: "EVI INCLUDE aggregate example" on page 69

Related reference:

"Encoded vector index index-only access" on page 17

The encoded vector index can also be used for index-only access.

"Encoded vector index symbol table scan" on page 18

An encoded vector index symbol table scan operation is used to retrieve the entries from the symbol table portion of the index.

"Encoded vector index symbol table probe" on page 21

An encoded vector index symbol table probe operation is used to retrieve entries from the symbol table portion of the index. Scanning the entire symbol table is not necessary.

"Index grouping implementation" on page 67

There are two primary ways to implement grouping using an index: Ordered grouping and pre-summarized processing.

Related information:

SQL INCLUDE statement

When to create EVIs

There are several instances to consider creating EVIs.

Consider creating encoded vector indexes when any one of the following is true:

- You want to gather 'live' statistics
- Full table scan is currently being selected for the query
- Selectivity of the query is 20%-70% and using skip sequential access with dynamic bitmaps speed up the scan
- When a star schema join is expected to be used for star schema join queries.
- When grouping or distinct queries are specified against a column, the columns have few distinct values and only the COUNT column function, if any, is used.
- When ready-made aggregate results grouped by the specified key columns would benefit query performance.

Create encoded vector indexes with:

- Single key columns with a low number of distinct values expected
- Keys columns with a low volatility (do not 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 with INCLUDE vs Materialized Query Tables**

| Although EVIs with INCLUDE are not a substitute for Materialized Query Tables (MQTs), INCLUDE
| EVIs have an advantage over single table aggregate MQTs (materialized query tables). The advantage is
| that the ready-made aggregate results are maintained in real time, not requiring explicit REFRESH TABLE
| requests. For performance and read access to aggregate results, consider turning your single table,
| aggregate MQTs into INCLUDE EVIs. Keep in mind that the other characteristics of a good EVI are
| applicable, such as a relatively low number of distinct key values.

| As indexes, these EVIs are found during optimization just as any other indexes are found. Unlike MQTs,
| there is no INI setting to enable and no second pass through the optimizer to cost the application of this
| form of ready-made aggregate. In addition, EVIs with INCLUDE can be used to populate MQT summary
| tables if the EVI is a match for a portion of the MQT definition.

Related reference:

“Encoded vector index symbol table scan” on page 18

An encoded vector index symbol table scan operation is used to retrieve the entries from the symbol table portion of the index.

“Index grouping implementation” on page 67

There are two primary ways to implement grouping using an index: Ordered grouping and pre-summarized processing.

Related information:

SQL INCLUDE statement

EVI maintenance

There are unique challenges to maintaining EVIs. The following table shows a progression of how EVIs are maintained, the conditions under which EVIs are most effective, and where EVIs are least effective, based on the EVI maintenance characteristics.

Related reference:

"Encoded vector index" on page 15

An encoded vector index is a permanent object that provides access to a table. This access is done by assigning codes to distinct key values and then representing those values in a vector.

Related information:

SQL INCLUDE statement

Recommendations for EVI use

Encoded vector indexes are a powerful tool for providing fast data access in decision support and query reporting environments. To ensure the effective use of EVIs, use 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. This method avoids the EVI maintenance involved in switching byte code sizes.
- | • EVIs with INCLUDE always create with a 4 byte code.

When loading data

- Drop EVIs, load data, create EVIs.
- EVI byte code size is assigned automatically based on the number of actual distinct key values found in the table.
- Symbol table contains all key values, in order, no keys in overflow area.
- | • EVIs with INCLUDE always use 4 byte code

| Consider adding INCLUDE values to existing EVIs

| An EVI index with INCLUDE values can be used to supply ready-made aggregate results. The existing symbol table and vector are still used for table selection, when appropriate, for skip sequential plans over large tables, or for index ANDing and ORing plans. If you already have EVIs, consider creating new ones with additional INCLUDE values, and then drop the pre-existing index.

| Consider specifying multiple INCLUDE values on the same EVI create

| If you need different aggregates over different table values for the same GROUP BY columns specified as EVI keys, define those aggregates in the same EVI. This definition cuts down on maintenance costs and allows for a single symbol table and vector.

| For example:

| Select SUM(revenue) from sales group by Country
| Select SUM(costOfGoods) from sales group by Country, Region

| Both queries could benefit from the following EVI:

```

| CREATE ENCODED VECTOR INDEX eviCountryRegion on Sales(country,region)
|   INCLUDE(SUM(revenue), SUM(costOfGoods))

```

| The optimizer does additional grouping (regrouping) if the EVI key values are wider than the corresponding GROUP BY request of the query. This additional grouping would be the case in the first example query.

| If an aggregate request is specified over null capable results, an implicit COUNT over that same result is included as part of the symbol table entry. The COUNT is used to facilitate index maintenance when a requested aggregate needs to reflect. It can also assist with pushing aggregation through a join if the optimizer determines this push is possible. The COUNT is then used to help compensate for fewer join activity due to the pushed down grouping.

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 IBM i 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 system 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 system can be four times as fast as a 1-processor system.

Checking values in the overflow area

You can also use the **Display File Description (DSPFD)** command (or System i 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 the access path of an index

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.

Related information:

SQL Create Index statement

SQL INCLUDE statement

Change Logical File (CHGLF) command

Display File Description (DSPFD) command

Comparing binary radix indexes and encoded vector indexes

DB2 for IBM i makes indexes a powerful tool.

The following table summarizes some of the differences between binary radix indexes and encoded vector indexes:

| *Table 51. Comparison of radix and EVI indexes*

| Comparison value | Binary Radix Indexes | Encoded Vector Indexes |
|-------------------------------|----------------------------------|-----------------------------|
| Basic data structure | A wide, flat tree | A Symbol Table and a vector |
| Interface for creating | Command, SQL, System i Navigator | SQL, System i Navigator |
| Can be created in parallel | Yes | Yes |
| Can be maintained in parallel | Yes | Yes |

Table 51. Comparison of radix and EVI indexes (continued)

| Comparison value | Binary Radix Indexes | Encoded Vector Indexes |
|--|----------------------|--|
| Used for statistics | Yes | Yes |
| Used for selection | Yes | Yes, with dynamic bitmaps or RRN list |
| Used for joining | Yes | Yes (with a hash table) |
| Used for grouping | Yes | Yes |
| Used for ordering | Yes | No |
| Used to enforce unique Referential Integrity constraints | Yes | No |
| Source for predetermined or ready-made numeric aggregate results | No | Yes, with INCLUDE keyword option on create |

Indexes & the optimizer

Since the optimizer uses cost based optimization, more information about the database rows and columns makes for a more efficient access plan created 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 a table 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 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 assists the optimizer and database engine with this task.

Instances where an index is not used

DB2 for i does not use indexes in the certain 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 department, the system cannot use an index with a key of WORKDEPT.

An index can be used if all the index updatable columns 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 updatable cursor because of dynamic SQL, or FOR UPDATE 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 for i does not use either index. The index has no added benefit because every row of the table needs to be looked at.

Display indexes for a table

You can display indexes that are created on a table using System i Navigator.

To display indexes for a table, follow these steps:

1. In the System i Navigator window, expand the system that you want to use.
2. Expand **Databases** and the database that you want to work with.
3. Expand **Schemas** and the schema that you want to work with.
4. Right-click a table and select **Show Indexes**.

The Show index window includes the following columns:

Table 52. Columns used in Show index window

| Column name | Description |
|-----------------|---|
| Name | The SQL name for the index |
| Type | The type of index displayed. Possible values are: <ul style="list-style-type: none"> • Keyed Physical File • Keyed Logical File • Primary Key Constraint • Unique Key Constraint • Foreign Key Constraint • Index |
| Schema | Schema or library containing the index or access path |
| Owner | User ID of the owner of this index or access path |
| System Name | System table name for the index or access path. |
| Text | The text description of the index or access path |
| Index partition | Partition detail for the index. Possible values: <ul style="list-style-type: none"> • <blank>, For all partitions • For Each Partition • specific name of the partition |
| Valid | Whether the access path or index is valid. The possible values are Yes or No. |

Table 52. Columns used in Show index window (continued)

| Column name | Description |
|----------------------------|--|
| Creation Date | The timestamp of when the index was created. |
| Last Build | The last time that the access path or index was rebuilt. |
| Last Query Use | Timestamp when the access path was last used by the optimizer. |
| Last Query Statistics Use | Timestamp when the access path was last used for statistics |
| Query Use Count | Number of times the access path has been used for a query |
| Query Statistics Use Count | Number of times the access path has been used for statistics |
| Last Used Date | Timestamp when the access path or index was last used. |
| Days Used Count | The number of days the index has been used. |
| Date Reset Days Used Count | The year and date when the days-used count was last set to 0. |
| Number of Key Columns | The number of key columns defined for the access path or index. |
| Key Columns | The key columns defined for the access path or index. |
| Current Key Values | The number of current key values. |
| Current Size | The size of the access path or index. |
| Current Allocated Pages | The current number of pages allocated for the access path or index. |
| Logical Page Size | The number of bytes used for the access path or the logical page size of the index. Indexes with larger logical page sizes are typically more efficient when scanned during query processing. Indexes with smaller logical page sizes are typically more efficient for simple index probes and individual key look ups. If the access path or index is an encoded vector, the value 0 is returned. |
| Duplicate Key Order | How the access path or index handles duplicate key values. Possible values are: <ul style="list-style-type: none"> • Unique - all values are unique. • Unique where not null - all values are unique unless null is specified. |
| Maximum Key Length | The maximum key length for the access path or index. |
| Unique Partial Key Values | The number of unique partial keys for the key fields 1 through 4. If the access path is an encoded vector, this number represents the number of full key distinct values. |
| Overflow Values | The number of overflow values for this encoded vector index. |
| Key Code Size | The length of the code assigned to each distinct key value of the encoded vector index. |
| Sparse | Is the index considered sparse. Sparse indexes only contain keys for rows that satisfy the query. Possible values are: <ul style="list-style-type: none"> • Yes • No |
| Derived Key | Is the index considered derived. A derived key is a key that is the result of an operation on the base column. Possible values are: <ul style="list-style-type: none"> • Yes • No |
| Partitioned | Is the index partition created for each data partition defined for the table using the specified columns. Possible values are: <ul style="list-style-type: none"> • Yes • No |
| Maximum Size | The maximum size of the access path or index. |

Table 52. Columns used in Show index window (continued)

| Column name | Description |
|------------------------------|---|
| Sort Sequence | The alternate character sorting sequence for National Language Support (NLS). |
| Language Identifier | The language code for the object. |
| Estimated Rebuild Time | The estimated time in seconds required to rebuild the access path or index. |
| Held | Is a rebuild of an access path or index held. Possible values are: <ul style="list-style-type: none"> • Yes • No |
| Maintenance | For objects with key fields or join logical files, the type of access path maintenance used. The possible values are: <ul style="list-style-type: none"> • Do not wait • Delayed • Rebuild |
| Delayed Maintenance Keys | The number of delayed maintenance keys for the access path or index. |
| Recovery | When the access path is rebuilt after damage to the access path is recognized. The possible values are: <ul style="list-style-type: none"> • After IPL • During IPL • Next Open |
| Index Logical Reads | The number of access path or index logical read operations since the last IPL. |
| WHERE Clause | Specifies the condition to apply for a row to be included in the index. |
| WHERE Clause Has UDF | Does the WHERE clause have a UDF. Possible values are: <ul style="list-style-type: none"> • Yes • No |
| Table | Table name of the table that the index is based on. |
| Table Partition | Partition name of the table that the index is based on. |
| Table System Name | System name of the table that the index is based on. |
| Last Rebuild Number Keys | Number of keys in the index when the index was last rebuilt. |
| Last Rebuild Parallel Degree | Parallel degree used when the index was last rebuilt. |
| Last Rebuild Time | Amount of time in seconds it took to rebuild the index the last time the index was rebuilt. |
| Keep in Memory | Is the index kept in memory. Possible values are: <ul style="list-style-type: none"> • Yes • No |
| Sort Sequence Schema | Schema of the sort sequence table if one is used. |
| Sort Sequence Name | Name of the sort sequence table if one is used. |
| Random Reads | The number of reads that have occurred in a random fashion. Random means that the location of the row or key could not be predicted ahead of time. |
| Media Preference | Indicates preference whether the storage for the table, partition, or index is allocated on Solid State Disk (SSD), if available. |

Determine unnecessary indexes

You can easily determine which indexes are being used for query optimization.

Before 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, 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, keyed logical file, join logical file, SQL index, primary key or unique constraint, or referential constraint. However, you can now easily find all indexes and retrieve statistics on index usage as a result of System i Navigator and IBM i 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 access index information through the System i Navigator, navigate to: **Database > Schemas > Tables**. Right-click your table and select **Show Indexes**.

You can show all indexes for a schema by right-clicking on **Tables** or **Indexes** and selecting Show indexes.

Note: You can also view the statistics through the Retrieve Member Description (QUSRMBRD) API.

Certain fields available in the **Show Indexes** window can help you to determine any unnecessary indexes. Those fields are:

Last Query Use

States the timestamp when the index was last used to retrieve data for a query.

Last Query Statistic Use

States the timestamp when the index was last used to provide 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.

Last Used Date

The century and date this index was last used.

Days Used Count

The number of days the index was used. If the index does not have a last used date, the count is 0.

Date Reset Days Used Count

The date that the days used count was last reset. You can reset the days used by **Change Object Description (CHGOBJD)** command.

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 is updated regardless of which 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 system, the statistics are reset.

Related information:

Retrieve Member Description (QUSRMBRD) API

Change Object Description (CHGOBJD) command

Reset usage counts

Resetting the usage counts for a table allows you to determine how the changes you made to your indexing strategy affected the indexes and constraints on that table. For example, if your new strategy causes an index to never be used, you could then delete that index. Resetting usage counts on a table affect all indexes and constraints that are created on that object.

Note: Resetting usage counts for a keyed physical file or a constraint in the Show Indexes window resets the counts of all constraints and keyed access for that file or table.

You can reset index usage counts by right-clicking a specific index in the Indexes folder or in the Show Indexes dialog and selecting **Reset Usage Counts**.

View index build status

| You can view a list of indexes that are being built by the database. This view might be helpful in
| determining when the index becomes usable to your applications.

| To display indexes that are being built, follow these steps:

- | 1. In the System i Navigator window, expand the system that you want to use.
- | 2. Expand **Databases**.
- | 3. Expand the database that you want to work with and then expand the Database Maintenance folder.
| Select **Index Builds**.

Manage index rebuilds

You can manage the rebuild of your indexes using System i Navigator. You can view a list of access paths that are rebuilding and either hold the access path rebuild or change the priority of a rebuild.

To display access paths to rebuild, follow these steps:

- | 1. In the System i Navigator window, expand the system that you want to use.
- | 2. Expand **Databases**.
- | 3. Expand the database that you want to work with and then expand the **Database Maintenance** folder.
| Select **Index Rebuilds**.

The access paths to rebuild dialog includes the following columns:

| *Table 53. Columns used in Index rebuilds window*

| Column name | Description |
|---------------|---|
| Name | Name of access path being rebuilt. |
| Schema | Schema name where the index is located. |
| System Name | The system name of the file that owns the index to be rebuilt. |
| System Schema | System schema name of access path being rebuilt. |
| Type | The type of index displayed. Possible values are: Keyed Physical File Keyed Logical File Primary Key Unique Key Foreign Key Index |
| Status | Displays the status of the rebuild. Possible values are: 1-99 – <i>Rebuild Priority</i> Running – <i>Rebuilding</i> Held – <i>Held from be rebuilt</i> |

Table 53. Columns used in Index rebuilds window (continued)

| Column name | Description |
|------------------------|---|
| Rebuild Priority | <p>Displays the priority in which the rebuild for this access path is run. Also referred to as sequence number. Possible values are:</p> <ul style="list-style-type: none"> 1-99: Order to rebuild Held Open |
| Rebuild Reason | <p>Displays the reason why this access path needs to be rebuilt. Possible values are:</p> <ul style="list-style-type: none"> Create or build index IPL Runtime error Change file or index sharing Other Not needed Change End of Data Restore Alter table Change table Change file Reorganize Enable a constraint Alter table recovery Change file recovery Index shared Runtime error Verify constraint Convert member Restore recovery |
| Rebuild Reason Subtype | <p>Displays the subtype reason why this access path needs to be rebuilt. Possible values are:</p> <ul style="list-style-type: none"> Unexpected error Index in use during failure Unexpected error during update, delete, or insert Delayed maintenance overflow or catch-up error Other No event Change End of Data Delayed maintenance mismatch Logical page size mismatch Partial index restore Index conversion Index not saved and restored Partitioning mismatch Partitioning change Index or key attributes change Original index invalid Index attributes change Force rebuild of index Index not restored Asynchronous rebuilds requested Job ended abnormally Alter table Change constraint Index invalid or attributes change Invalid unique index found Invalid constraint index found Index conversion required <p>If there is no subtype, this field displays 0.</p> |

Table 53. Columns used in Index rebuilds window (continued)

| Column name | Description |
|---------------------------|---|
| Invalidation Reason | Displays the reason why this access path was invalidated. Possible values are: User requested (See Invalidation Reason type for more information) Create or build Index Load (See Invalidation Reason type for more information) Initial Program Load (IPL) Runtime error Modify Journal failed to build the index Marked index as fixable during runtime Marked index as fixable during IPL Change end of data |
| Invalidation Reason Type | Displays the reason type for why this access path was invalidation. Possible reason types for User requested: Invalid because of REORG It is a copy Alter file Converting new member Change to *FRCRBDAP Change to *UNIQUE Change to *REBLD Possible reason type for LOAD The index was marked for invalidation but the system crashed before the invalidation could actually occur The index was associated with the overlaid data space header during a load, therefore it was invalidated Index was in IMPI format. The header was converted and now it is invalidated to be rebuilt in RISC format The RISC index was converted to V5R1 format Index invalidated due to partial load Index invalidated due to a delayed maintenance mismatch Index invalidated due to a pad key mismatch Index invalidated due to a significant fields bitmap fix Index invalidated due to a logical page size mismatch Index was not restored. File might have been saved with ACCPTH(*NO) or index did not exist when file was saved. Index was not restored. File might have been saved with ACCPTH(*NO) or index did not exist when file was saved. Index was rebuilt because file was saved in an inconsistent state with SAVACT(*SYSDFN). For other invalidation codes, this field displays 0. |
| Estimated Rebuild Time | Estimated amount of time in seconds that it takes to rebuild the index access path. |
| Rebuild Start Time | Time when the rebuild was started. |
| Elapsed Rebuild Time | Amount of time that has elapsed in seconds since the start of the rebuild of the access path |
| Unique | Indicates whether the rows in the access path are unique. Possible values are: Yes No |
| Last Query Use | Timestamp when the access path was last used |
| Last Query Statistics Use | Timestamp when the access path was last used for statistics |
| Query Use Count | Number of times the access path has been used for a query |

Table 53. Columns used in Index rebuilds window (continued)

| Column name | Description |
|----------------------------|--|
| Query Statistics Use Count | Number of times the access path has been used for statistics |
| Partition | Partition detail for the index. Possible values: <ul style="list-style-type: none"> • <blank>, which means For all partitions • For Each Partition • specific name of the partition |
| Owner | User ID of the owner of this access path. |
| Parallel Degree | Number of processors to be used to rebuild the index. |
| Text | Text description of the file owning the index. |

You can also use the **Edit Rebuild of Access Paths (EDTRBDAP)** command to manage rebuilding of access paths.

Related information:

Rebuild access paths

Edit Rebuild of Access Paths (EDTRBDAP) command

Indexing strategy

There are two approaches to index creation: proactive and reactive. Proactive index creation involves anticipating which columns are most often used for selection, joining, grouping, and ordering. 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 applications 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

Related information:

 [Indexing and statistics strategies for DB2 for i5/OS](#)

Reactive approach to tuning

To perform reactive tuning, build a prototype of the proposed application without any indexes and start running some queries. Or you could build an initial set of indexes and start running the application to see which ones get used and which do not. Even with a smaller database, the slow running queries become obvious quickly.

The reactive tuning method is also used when trying to understand and tune an existing application that is not performing up to expectations. Use the appropriate debugging and monitoring tools, described in the next section, to view the database feedback messages:

- the indexes the optimizer recommends for local selection
- the temporary indexes used for a query
- the query implementation methods the optimizer chose

If the database engine is building temporary indexes to process joins or perform grouping and selection over permanent tables, build permanent indexes over the same columns. This technique is used to eliminate the temporary index creation. In some cases, a temporary index is built over a temporary table,

so a permanent index is not able to be built for those tables. You can use the optimization tools listed in the previous section to note the temporary index creation, the reason it was created, and the key columns.

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 use 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' AND
      b.col2=some_number AND
      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 to help you design code which allows DB2 for i to take advantage of available indexes:

Avoid numeric conversions

When a column value and a host variable (or constant value) are being compared, try to specify the same data types and attributes. DB2 for i might 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 for i needs 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 might not use the index in the second example. The constant precision is greater than the column precision. It attempts to convert the constant to 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 compare to a column in a row selection predicate. The optimizer does not use an index on a column compared to an arithmetic expression. While this technique might not cause the column index to become unusable, it prevents any estimates and possibly the use of index scan-key positioning. 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 for i might 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 (%), and underline (_), used in the pattern of a LIKE (OPNQRYF %WLDCRD) predicate, specify a character string like the row column values to select. They can take advantage of indexes when used to denote characters in the middle or at the end of a character string.

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 for i 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*'')')
```

Avoid patterns with a % so that you can get the best performance with key processing on the predicate. If possible, 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 returns all names starting with "S." Adjust the query to return all names with "Smi%". By forcing the use of partial strings, you might get better performance in the long term.

Using derived indexes

SQL indexes can be created where the key is specified as an expression. This type of key is also referred to as a derived key.

For example, look at the following:

```
CREATE INDEX TOTALIX ON EMPLOYEE(SALARY+BONUS+COMM AS TOTAL)
```

In this example, return all the employees whose total compensation is greater than 50000.

```
SELECT * FROM EMPLOYEE
WHERE SALARY+BONUS+COMM > 50000
ORDER BY SALARY+BONUS+COMM
```

Since the optimizer uses the index TOTALIX with index probe to satisfy the WHERE selection and the ordering criteria.

Some special considerations to with derived key index usage and matching include:

- There is no matching for index key constants to query host variables. This non-match includes implicit parameter marker conversion performed by the database manager.

```
CREATE INDEX D_IDX1 ON EMPLOYEE (SALARY/12 AS MONTHLY)
```

In this example, return all employees whose monthly salary is greater than 3000.

```
long months = 12;
```

```
EXEC SQL SELECT * FROM EMPLOYEE WHERE SALARY/:months > 3000
```

However, in this case the optimizer does not use the index since there is no support for matching the host variable value months in the query to the constant 12 in the index.

Usage of the QAQQINI option `PARAMETER_MARKER_CONVERSION` with value `*NO` can be used to prevent conversion of constants to parameter markers. This technique allows for improved derived index key matching. However, because of the performance implications of using this QAQQINI setting, take care with its usage.

- In general, expressions in the index must match the expression in the query:

```
.... WHERE SALARY+COMM+BONUS > 50000
```

In this case, the `WHERE SALARY+COMM+BONUS` is different from the index key `SALARY+BONUS+COMM` and would not match.

- It is recommended that the derived index keys be kept as simple as possible. The more complex the query expression to match and the index key expression is, the less likely it is that the index is used.
- The CQE optimizer has limited support for matching derived key indexes.

Related reference:

“Derived key index” on page 194

You can use the SQL `CREATE INDEX` statement to create a derived key index using an SQL expression.

Related information:

SQL Create Index statement

| **Using sparse indexes**

| SQL indexes can be created using `WHERE` selection predicates. These indexes can also be referred to as sparse indexes. The advantage of a sparse index is that fewer entries are maintained in the index. Only those entries matching the `WHERE` selection criteria are maintained in the index.

| In general, the query `WHERE` selection must be a subset of the sparse index `WHERE` selection in order for the sparse index to be used.

| Here is a simple example of when a sparse index can be used:

```
| CREATE INDEX MYLIB/SPR1 ON MYLIB/T1 (COL3)
| WHERE COL1=10 AND COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 AND COL2=20 AND COL3=30
```

| It is recommended that the `WHERE` selection in the sparse index is kept as simple as possible. The more complex the `WHERE` selection, the more difficult it becomes to match the sparse index `WHERE` selection to the query `WHERE` selection. Then it is less likely that the sparse index is used. The CQE optimizer does not support sparse indexes. It does support `select/omit` logical files however. The SQE optimizer matches the CQE optimizer in its support for `select/omit` logical files and has nearly full support for sparse indexes.

| **Related reference:**

| “Sparse indexes” on page 194

| You can use the SQL `CREATE INDEX` statement to create a sparse index using SQL selection predicates.

| **Related information:**

| SQL Create Index statement

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

Before using an existing index, DB2 for i 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 query sort sequence table (specified by the SRTSEQ and LANGID) must match the index sort sequence table. DB2 for i compares the sort sequence tables. If they do not match, the existing index cannot be used.

There is an exception to this rule, however. If the sort sequence table associated with the query is a unique-weight sequence table (including *HEX), DB2 for i acts as though no sort sequence table is specified for selection, join, or grouping columns that use the following operators and predicates:

- equal (=) operator
- not equal (^= or <>) operator
- LIKE predicate (OPNQRYP %WLDCRD and *CT)
- IN predicate (OPNQRYP %VALUES)

When these conditions are true, DB2 for i is free to use any existing index where the key columns match the columns and either:

- The index does not contain a sort sequence table or
- The index contains a unique-weight sort sequence table

Note:

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 referenced in the query selection, then those indexes must either use the same sort sequence table or no sort sequence table.

Using indexes and sort sequence with ordering

Unless the optimizer chooses a sort to satisfy the ordering request, the index sort sequence table must match the query sort sequence table.

When a sort is used, the translation is done during the sort. Since the sort is handling the sort sequence requirement, this technique allows DB2 for i to use any existing index that meets the selection criteria.

Index examples

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)
```

Index example: Equal selection with no sort sequence table

Equal selection with no sort sequence table (SRTSEQ(*HEX)).

```
SELECT * FROM STAFF
WHERE JOB = 'MGR'
```

When using the OPNQRYP command, specify:

```
OPNQRYF FILE((STAFF))
  QRYSLT('JOB *EQ ''MGR''')
  SRTSEQ(*HEX)
```

The system can use either index HEXIX or index UNQIX.

Index example: Equal selection with a unique-weight sort sequence table

Equal 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 is 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 three 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 creates 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 does one of the following actions:

- 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 creates 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 creates 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.

| Sparse index examples

| This topic shows examples of how the sparse index matching algorithm works.

| In example S1, the query selection is a subset of the sparse index selection and consequently an index scan over the sparse index is used. The remaining query selection (COL3=30) is executed following the index scan.

| Example S1

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| In example S2, the query selection is not a subset of the sparse index selection and the sparse index cannot be used.

| Example S2

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20
```

| In example S3, the query selection exactly matches the sparse index selection and an index scan over the sparse index can be used.

| Example S3

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| In example S4, the query selection is a subset of the sparse index selection and an index scan over the sparse index can be used. The remaining query selection (COL3=30) is executed following the index scan.

| Example S4

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30
```

| In example S5, the query selection is not a subset of the sparse index selection and the sparse index cannot be used.

| Example S5

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20
```

| In example S6, the query selection exactly matches the sparse index selection and an index scan over the sparse index can be used. The query selection is executed following the index scan to eliminate excess records from the sparse index.

| Example S6

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20 or COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20 or COL3=30
```

| In example S7, the query selection is a subset of the sparse index selection and an index scan over the sparse index can be used. The query selection is executed following the index scan to eliminate excess records from the sparse index.

| Example S7

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20 or COL3=30
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20
```

| In example S8, the query selection is not a subset of the sparse index selection and the sparse index cannot be used.

| Example S8

```
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 or COL2=20
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 or COL2=20 or COL3=30
```

| In the next example S9, the constant 'MN' was replaced by a parameter marker for the query selection. The sparse index had the local selection of COL1='MN' applied to it when it was created. The sparse index matching algorithm matches the parameter marker to the constant 'MN' in the query predicate COL1=?. It verifies that the value of the parameter marker is the same as the constant in the sparse index; therefore the sparse index can be used.

| Example S9

```

| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1='MN' or COL2='TWINS'
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| Where Col3='WIN' and (Col1=? or Col2='TWINS')

```

| In the next example S10, the keys of the sparse index match the order by fields in the query. For the sparse index to satisfy the specified ordering, the optimizer must verify that the query selection is a subset of the sparse index selection. In this example, the sparse index can be used.

| Example S10

```

| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL1, COL3)
| WHERE COL1='MN' or COL2='TWINS'
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| Where Col3='WIN' and (Col1='MN' or Col2='TWINS')
| ORDER BY COL1, COL3

```

| In the next example S11, the keys of the sparse index do not match the order by fields in the query. But the selection in sparse index T2 is a superset of the query selection. Depending on size, the optimizer might choose an index scan over sparse index T2 and then use a sort to satisfy the specified ordering.

| Example S11

```

| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL2, COL4)
| WHERE COL1='MN' or COL2='TWINS'
| SELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| Where Col3='WIN' and (Col1='MN' or Col2='TWINS')
| ORDER BY COL1, COL3

```

| The next example S12 represents the classic optimizer decision: is it better to do an index probe using index IX1 or is it better to do an index scan using sparse index SPR1? Both indexes retrieve the same number of index entries and have the same cost from that point forward. For example, both indexes have the same cost to retrieve the selected records from the dataspace, based on the retrieved entries/keys.

| The cost to retrieve the index entries is the deciding criteria. In general, if index IX1 is large then an index scan over sparse index SPR1 has a lower cost to retrieve the index entries. If index IX1 is rather small then an index probe over index IX1 has a lower cost to retrieve the index entries. Another cost decision is reusability. The plan using sparse index SPR1 is not as reusable as the plan using index IX1 because of the static selection built into the sparse selection.

| Example S12

```

| CREATE INDEX MYLIB/IX1 on MYLIB/T1 (COL1, COL2, COL3)
| CREATE INDEX MYLIB/SPR1 on MYLIB/T1 (COL3)
| WHERE COL1=10 and COL2=20 and COL3=30
| CSELECT COL1, COL2, COL3, COL4
| FROM MYLIB/T1
| WHERE COL1=10 and COL2=20 and COL3=30

```

Application design tips for database performance

There are some design tips that you can apply when designing SQL applications to maximize your database performance.

Using 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 control is done by specifying the allow copy data (ALWCPYDTA) parameter on the precompiler commands or the **Start SQL (STRSQL)** command.

Specifying ALWCPYDTA(*NO) instructs the database manager to always use live data. In most cases, forcing live data access is a detriment to performance. It severely limits the possible plan choices that the optimizer could use to implement the query. Avoid it in most cases. However, in specialized cases involving a simple query, live data access can be used as a performance advantage. The cursor does not need to be closed and opened again to refresh the data being retrieved.

An example application demonstrating this advantage is one that produces a list on a display. If the display can show only 20 list elements at a time, then, after the initial 20 elements are displayed, the programmer can request that the next 20 rows be displayed. A typical SQL application designed for an operating system other than the IBM i 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 preceding example, notice that an additional cursor had to be opened to continue the list and to get current data. This technique can result in creating an additional ODP that increases the processing

time on the system. In place of the preceding 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 preceding 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.

Related information:

Start SQL Interactive Session (STRSQL) command

Reducing 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 could also require one open per subselect.

To minimize the number of opens, DB2 for i 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 optimizer could 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 does not have the entries required for a different host variable value.
- 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 causes the ODP to be closed within a given program invocation.

- The join is a complex join that requires temporary objects to contain the intermediate steps of the join.
- Some cases involve a complex sort, where a temporary file is required, might 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 by the CQE optimizer using hash join.

For embedded static SQL, DB2 for i 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 for i 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 **Reclaim Resources (RCLRSC)** is issued when the first COBOL program on the call stack ends or when a COBOL program issues the STOP RUN COBOL statement. **Reclaim Resources (RCLRSC)** does not close the ODPs created for programs precompiled using CLOSQLCSR(*ENDJOB). For interaction of **Reclaim Resources (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
- the last program containing SQL statements on the call stack exits. Exception is for ODPs created for programs precompiled using CLOSQLCSR(*ENDJOB) or modules precompiled using CLOSQLCSR(*ENDACTGRP).
- a CONNECT (Type 1) statement changes the application server for an activation group, all ODPs created for the activation group are closed.
- a DISCONNECT statement ends a connection to the application server, all ODPs for that application server are closed.
- a released connection is ended by a successful COMMIT, all ODPs for that application server are closed.
- the threshold for open cursors specified by the query options file (QAQQINI) parameter OPEN_CURSOR_THRESHOLD is reached.
- the SQL LOCK TABLE or CL ALCOBJ OBJ((filename *FILE *EXCL)) CONFLICT(*RQSRLS) command closes any pseudo-closed cursors associated with the specified table.
- an application has requested a close, but the data path was left open. The ODP can be forced closed for a specific file by using the ALCOBJ CL command. This close does not force the ODP to close if the application has not requested that the cursor be closed. The syntax for the command is: ALCOBJ OBJ((library/file *FILE *EXCL)) CONFLICT(*RQSRLS).
- | • an MQT plan expired based on the timestamp.
- | • an incompatible commitment control change occurred.

- | • the table size changed beyond tolerance. The optimizer needs to reoptimize based on the new table size.
- | • a new index or indexes were created. The optimizer can cost a plan created with the new indexes and compare its cost to the previous plan.
- | • new statistics were created. The optimizer can take advantage of these new statistics to create a more efficient plan.
- | • host variables are incompatible with a non-reusable MTI, an MQT, or a sparse index used to implement the query.
- | • data is warm (in memory).
- | • the OPTIMIZATION_GOAL *All IO or *First IO specified in query options file QAQQINI was changed.
- | • a hard close was forced.

| The optimizer does not recognize that query selectivity has changed due to host variable changes. It continues to use the existing open and access plan. Change of selectivity due to host variables is only evaluated at full open time.

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)

An open operation occurs for the first execution of each UPDATE WHERE CURRENT OF, when any SET clause expression 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)**.

Related information:

Reclaim Resources (RCLRSC) command

Trace Job (TRCJOB) command

Display Journal (DSPJRN) command
RPG
COBOL
C and C++

Retaining cursor positions

You can improve performance by retaining cursor positions.

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**

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**

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.

Related reference:

“Effects of precompile options on database performance” on page 242

Several precompile options are available for creating SQL programs with improved performance. They are only options because using them could impact the function of the application. For this reason, the default value for these parameters is the value that ensures successful migration of applications from prior releases. However, you can improve performance by specifying other options.

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

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 are also 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

Programs compiled with either CLOSQLCSR(*ENDPGM) or CLOSQLCSR(*ENDMOD) must open a cursor every time the program or module is called, in order to access the data. If the SQL program or module is 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 might 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 is able to use a reusable ODP. This technique means that, as long as SQL remains active between the calls to this program, the OPEN statement does 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 or *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. Use a FETCH statement first, and then run 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 is only needed on the first call to the program, assuming that the correct CLOSQLCSR option was chosen. If the statement can change between calls to the program, perform the PREPARE in all cases.

The main program might also control cursors by sending a special parameter on the first call only. This special parameter value indicates that because it is the first call, the subprogram performs 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.

Programming techniques for database performance

By changing the coding of your queries, you can improve their performance.

Use the OPTIMIZE clause

If an application is not going to retrieve the entire result table for a cursor, using the OPTIMIZE clause can improve performance. The query optimizer modifies the cost estimates to retrieve the subset of rows using the value specified on the OPTIMIZE clause.

Assume that the following query returns 1000 rows:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'A00'
  ORDER BY LASTNAME
  OPTIMIZE FOR 100 ROWS
END EXEC.
```

Note: The values that can be used for the preceding OPTIMIZE clause 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 method allows 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, or output device.

Table 54. OPTIMIZE FOR n ROWS default value

| 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 run a sort or hash by specifying many possible rows in the answer set.

Related information:

select-statement

Use FETCH FOR n ROWS

Applications that perform many FETCH statements in succession could be improved by using FETCH FOR n ROWS. With this clause, you can retrieve multiple rows of table data with a single FETCH, putting them into a host structure array or row storage area.

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 is filled by the FETCH, the application loops through the data, processing 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.

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 runtime and the database manager when blocking can be performed.

Table 55. 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 one database call |
| Multiple-Row FETCH Statement | one SQL runtime call 100 database calls | one SQL runtime call one database call |

Related information:

FETCH statement

Improve SQL blocking performance when using FETCH FOR n ROWS

Use these performance techniques to improve SQL blocking performance when using FETCH FOR n ROWS.

You can improve SQL blocking performance with the following:

- Match the attribute information in the host structure array or the descriptor associated with the row storage area with the attributes of the columns retrieved.
- 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.
- Do not mix single- and multiple-row FETCH requests against the same cursor 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.
- Do not use the PRIOR, CURRENT, and RELATIVE scroll options 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 could 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 inserts the entire array into the table. The statement runs faster because the SQL runtime 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 runtime and to the database manager when blocking can be performed.

Table 56. Number of Calls Using an INSERT Statement

| | Database Manager Not Using Blocking | Database Manager Using Blocking |
|-------------------------------|--|---|
| Single-Row INSERT Statement | 100 SQL runtime calls 100 database calls | 100 SQL runtime calls one database call |
| Multiple-Row INSERT Statement | 1 SQL runtime call 100 database calls | 1 SQL runtime call 1 database call |

Related information:

INSERT statement

Control database manager blocking

To improve performance, the SQL runtime 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 index key columns used by the cursor, and 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 runtime 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 inserted into the target table until the block is full. The SQL runtime automatically does blocking for blocked inserts.

Note: If an INSERT with VALUES is specified, the SQL runtime might not close the internal cursor 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 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.

Related reference:

“Effects of precompile options on database performance” on page 242

Several precompile options are available for creating SQL programs with improved performance. They are only options because using them could impact the function of the application. For this reason, the default value for these parameters is the value that ensures successful migration of applications from prior releases. However, you can improve performance by specifying other options.

Related information:

Override Database File (OVRDBF) command

Optimize the number of columns that are selected with SELECT statements

For each column in the SELECT statement, the database manager retrieves the data from the underlying table and maps it to a host variable in the application program. 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 required for the application. This technique is especially important for index-only access, or if all the columns participate in a sort operation (as in SELECT DISTINCT and SELECT UNION).

This technique is also important when considering index only access. You minimize the number of columns in a query and increase the odds that an index can be used to completely satisfy the data request.

Related information:

select-statement

Eliminate redundant validation with SQL PREPARE statements

The processing which occurs when an SQL PREPARE statement is run is like 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 is valid.
- An access plan is built.

Again when the statement is executed or opened, the database manager revalidates 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 completely validates the dynamic statement. The validation is 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 must check the SQLCODE and SQLSTATE after running the OPEN or EXECUTE statement to ensure that the statement is valid. DLYPRP(*YES) does 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.

Related reference:

“Effects of precompile options on database performance” on page 242

Several precompile options are available for creating SQL programs with improved performance. They are only options because using them could impact the function of the application. For this reason, the default value for these parameters is the value that ensures successful migration of applications from prior releases. However, you can improve performance by specifying other options.

Related information:

Prepare statement

Page interactively displayed data with REFRESH(*FORWARD)

In large tables, paging performance is typically degraded because of the REFRESH(*ALWAYS) parameter on the **Start SQL (STRSQL)** command. STRSQL 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 change 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 in the temporary table instead of the updated table.

The refresh option can be changed on the Session Services display.

Related information:

Start SQL (STRSQL) command

Improve concurrency by avoiding lock waits

The concurrent access resolution option directs the database manager on how to handle cases of record lock conflicts under certain isolation levels.

The concurrent access resolution, when applicable, can have one of the following values:

- **Wait for outcome** (default). This value directs the database manager to wait for the commit or rollback when encountering locked data in the process of being updated or deleted. Locked rows that are in the process of being inserted are not skipped. This option does not apply for read-only queries running under isolation level None or Uncommitted Read.
- **Use currently committed**. This value allows the database manager to use the currently committed version of the data for read-only queries when encountering locked data in the process of being updated or deleted. Locked rows in the process of being inserted can be skipped. This option applies if possible when the isolation level in effect is Cursor Stability and is ignored otherwise.
- **Skip locked data**. This value directs the database manager to skip rows in the case of record lock conflicts. This option is applicable only when the query is running under an isolation level of Cursor Stability or Read Stability and additionally for UPDATE and DELETE queries when the isolation level is None or Uncommitted Read.

The concurrent access resolution values of USE CURRENTLY COMMITTED and SKIP LOCKED DATA can be used to improve concurrency by avoiding lock waits. However, care must be used when using these options because they might affect application functionality.

WAIT FOR OUTCOME, USE CURRENTLY COMMITTED, and SKIP LOCKED DATA can be specified as the concurrent-access-resolution-clause in the attribute-string of a PREPARE statement.

Additionally, they can be specified as the concurrent-access-resolution-clause at the statement level on a select-statement, SELECT INTO, searched UPDATE, or searched DELETE statement.

Concurrent access resolution is also specifiable as a precompiler option by using the CONACC parameter on the CRTSQLxxx and RUNSQLSTM commands. The CONACC parameter accepts one of the following values:

- ***DFT** - specifies that the concurrent access option is not explicitly set for this program. The value that is in effect when the program is invoked is used. The value can be set using the SQL_CONCURRENT_ACCESS_RESOLUTION option in the query options file QAQQINI.
- ***CURCMT** - use currently committed.
- ***WAIT** - wait for outcome.

| When the concurrent access resolution option is not directly set by the application, it is set to the value of the SQL_CONCURRENT_ACCESS_RESOLUTION option in the query options file QAQQINI. This option accepts one of the following values:

- | • *DEFAULT - the default value is set to *WAIT.
- | • *CURCMT - use currently committed.
- | • *WAIT - wait for outcome.

| **Related reference:**

| “QAQQINI query options” on page 162
 | There are different options available for parameters in the QAQQINI file.

| **Related information:**

| concurrent-access-resolution-clause
 | Concurrency

General DB2 for i performance considerations

As you code your applications, there are some general tips that can help you optimize performance.

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 could impact the function of the application. For this reason, the default value for these parameters is the value that ensures 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 might be suitable for most of your applications. Use the command **CRTDUPOBJ** to create a copy of the SQL **CRSQLxxx** 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.

Table 57. Precompile options and their performance impacts

| Precompile Option | Optimal Value | Improvements | Considerations |
|-------------------|-------------------------|--|--|
| ALWCPYDTA | *OPTIMIZE (the default) | Queries where the ordering or grouping criteria conflicts with the selection criteria. | A copy of the data could be made when the query is opened. |
| ALWBLK | *ALLREAD (the default) | Additional read-only cursors use blocking. | ROLLBACK HOLD might not change the position of a read-only cursor. Dynamic processing of positioned updates or deletes might fail. |

Table 57. Precompile options and their performance impacts (continued)

| Precompile Option | Optimal Value | Improvements | Considerations |
|-------------------|---------------------------------|--|--|
| 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. |
| DLYPRP | *YES | Programs using SQL PREPARE statements could run faster. | Complete validation of the prepared statement is delayed until the statement is run or opened. |
| TGTRLS | *CURRENT (the default) | The precompiler can generate code that takes advantage of performance enhancements available in the current release. | The program object cannot be used on a system from a previous release. |

Related reference:

“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.

“Control database manager blocking” on page 239

To improve performance, the SQL runtime attempts to retrieve and insert rows from the database manager a block at a time whenever possible.

“Retaining cursor positions for non-ILE program calls” on page 234

For non-ILE program calls, the close SQL cursor (CLOSQLCSR) parameter allows you to specify the scope of the following:

“Eliminate redundant validation with SQL PREPARE statements” on page 240

The processing which occurs when an SQL PREPARE statement is run is like the processing which occurs during precompile processing.

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 preceding example, when ALWCPYDTA(*NO) or ALWCPYDTA(*YES) is specified, the database manager could 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 ALWCOPYDTA(*OPTIMIZE) is specified, the database manager uses an index with the first index column of WORKDEPT. It then makes a copy of all the rows that match the WHERE condition. Finally, it could sort the copied rows by the values in LASTNAME. This row selection processing is more efficient, because the index used immediately locates the rows to be selected.

ALWCOPYDTA(*OPTIMIZE) optimizes the total time that is required to process the query. However, the time required to receive the first row could 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 could be important for applications that are presenting interactive displays or that retrieve only the first few rows of the query. The DB2 for i query optimizer can be influenced to avoid sorting by using the OPTIMIZE clause.

Queries that involve a join operation might also benefit from ALWCOPYDTA(*OPTIMIZE) because the join order can be optimized regardless of the ORDER BY specification.

Related concepts:

“Plan cache” on page 6

The plan cache is a repository that contains the access plans for queries that were optimized by SQE.

Related reference:

“Effects of precompile options on database performance” on page 242

Several precompile options are available for creating SQL programs with improved performance. They are only options because using them could impact the function of the application. For this reason, the default value for these parameters is the value that ensures successful migration of applications from prior releases. However, you can improve performance by specifying other options.

“Radix index scan” on page 12

A radix index scan operation is used to retrieve the rows from a table in a keyed sequence. Like a table scan, all the rows in the index are sequentially processed, but the resulting row numbers are sequenced based upon the key columns.

“Radix index probe” on page 13

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 scan is that the rows returned are first identified by a probe operation to subset them.

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 typically 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 must 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 two 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 the data stored in the overflow area) uses the least disk space.

Table 58. Disk space used with variable-length columns

| 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 doubles 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, use variable length host variables. 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. 30 of 35 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.

Running the **Reorganize Physical File Member (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 **Reorganize Physical File Member (RGZPFM)** command. This technique reduces the read time for rows that overflow, increases the locality of reference, and produces optimal order for serial batch processing.

Choose the appropriate maximum length for variable-length columns. Selecting lengths that are too long increases the process access group (PAG). A large PAG slows performance. A large maximum length makes SEQONLY(*YES) less effective. Variable-length columns longer than 2000 bytes are not eligible as key columns.

Using LOBs and VARCHAR in the same table

Storage for LOB columns allocated in the same manner as VARCHAR columns. When a column stored in the overflow storage area is referenced, currently all the columns in that area are paged into memory. A reference to a "smaller" VARCHAR column that is in the overflow area can potentially force extra paging of LOB columns. For example, A VARCHAR(256) column retrieved by application has side-effect of paging in two 5 MB BLOB columns that are in the same row. In order to prevent this side-effect, you might want to use ALLOCATE keyword to ensure that only LOB columns are stored in the overflow area.

Related information:

Reorganize Physical File Member (RGZPFM) command

Reorganizing a physical file

Embedded SQL programming

Using field procedures to provide column level encryption

Field procedures can provide column level encryption in DB2 for i.

A field procedure is a user-written exit routine to transform values in a single column. When values in the column are changed, or new values inserted, the field procedure is invoked for each value. The field procedure can transform that value (encode it) in any way. The encoded value is then stored. When values are retrieved from the column, the field procedure is invoked for each encoded value. The field procedure decodes each value back to the original value. Any indexes defined on a column that uses a field procedure are built with encoded values.

Field procedures are assigned to a table by the FIELDPROC clause of CREATE TABLE and ALTER TABLE.

A field procedure that is specified for a column is invoked in three general situations:

- For field-definition, when the CREATE TABLE or ALTER TABLE statement that names the procedure is executed. During this invocation, the procedure is expected to:
 - Determine whether the data type and attributes of the column are valid.
 - Verify the literal list, and change it if wanted.
 - Provide the field description of the column.
- For field-encoding, when a column value is field-encoded. That occurs for any value that:
 - is inserted in the column by an SQL INSERT statement, SQL MERGE statement, or native write.
 - is changed by an SQL UPDATE statement, SQL MERGE statement, or native update.

- is the target column for a copy from a column with an associated field procedure. The field procedure might be invoked to encode the copied data. Examples include SQL Statements ALTER TABLE or CREATE TABLE LIKE/AS and CL commands CPYF and RGZPFM.
- is compared to a column with a field procedure. The QAQQINI option FIELDPROC_ENCODED_COMPARISON is used to determine if the column value is decoded, or the host variable, constant, or join column is encoded.
- is the DEFAULT value for a column with an associated field procedure in a CREATE or ALTER TABLE statement.

If there are any **after** or **read** triggers, the field procedure is invoked *before* any of these triggers. If there are any **before** triggers, the field procedure is invoked *after* the before trigger.

- For field-decoding, when a stored value is field-decoded back into its original value. Field-decoding occurs for any value that:

- is retrieved by an SQL SELECT or FETCH statement, or by a native read.
- is a column with an associated field procedure that is copied. The field procedure might be invoked to decode the data before making the copy. Examples include SQL Statements ALTER TABLE, CREATE TABLE LIKE/AS, and CL commands CPYF and RGZPFM.
- is compared to a column with a field procedure. The QAQQINI option FIELDPROC_ENCODED_COMPARISON is used by the optimizer to decide if the column value is decoded, or if the host variable or constant is encoded.

A field procedure is never invoked to process a null value. It is also not invoked for a DELETE operation without a WHERE clause when the table has no DELETE triggers. The field procedure is invoked for empty strings.

Improving performance

For queries that use field procedures, the path length is longer due to the additional processing of calling the field procedure. In order to improve performance of queries, the SQE optimizer:

- attempts to remove decoding operations, based on the QAQQINI FIELDPROC_ENCODED COMPARISON setting.
- matches existing indexes over columns that have an associated field procedure.
- creates and uses MTIs over columns with field procedures.
- creates statistics over the encoded values through statistics collection.

The SQE optimizer attempts to do the following optimizations:

- optimization of predicates that compare a field procedure column to a constant or host variable. For example, predicate FP1(4, C1) = 'abc' is optimized as C1 = FP1(0,'abc'). With this specific example, the optimization is done as long as the QAQQINI option is not *NONE.
- remove field procedure decoding operations from join predicates when the same field procedure is applied to both sides of the join predicate, and no compatibility conversion is required. For example, join predicate FP1(4,T1.C1) > FP1(4,T2.C1) is rewritten as T1.C1 > T2.C1. With this specific example, the optimization is done as long as the QAQQINI option is either *ALLOW_RANGE or *ALL. This technique is also applied to = predicates when the QAQQINI option is *ALLOW_EQUAL.
- remove field procedures from GROUP BY and ORDER BY clauses. For example, ORDER BY FP1(4,C1) is rewritten as ORDER BY C1 if the QAQQINI setting is either *ALLOW_RANGE or *ALL

The CQE optimizer does not look at the QAQQINI option, which means it always runs in *NONE mode. *NONE mode requires that all references to the column are decoded before any other operation is performed. A CQE query does not match any existing indexes when the column has an associated field procedure. If an index is required, a temporary index is built with the index keys decoded.

Related reference:

“QAQQINI query options” on page 162

There are different options available for parameters in the QAQQINI file.

| **Related information:**

| Defining field procedures

| CREATE TABLE

| **Field procedure examples**

| The following examples show various field procedure-related optimizations done by the SQE optimizer.

| The examples show the FieldProc name along with the encoding (field procedure function code 0) or decoding (field procedure function code 4) in the pseudo-SQL. These codes indicate how the optimizer is optimizing the field procedure calls.

| Given the following table and index:

| CREATE TABLE T1 (col1 CHAR(10), col2 CHAR(10) FIELDPROC 'FP1')

| CREATE INDEX IX1 on T1(col2)

| **Example 1**

| A user query written as:

| SELECT col1, col2 FROM T1 WHERE col2 = 'abc'

| Is represented by the optimizer as:

| SELECT col1, FP1(4, col2) FROM T1 WHERE FP1(4,col2) = 'abc'

| Note the FP1 with the decode operation around the COL2 references in the SELECT list and the WHERE clause.

| Assuming the QAQQINI FIELDPROC_ENCODED COMPARISON is set to *ALLOW_EQUAL, *ALLOW_RANGE or *ALL:

| The query optimizer rewrites the query as:

| SELECT col1, 'abc' FROM T1 WHERE col2 = FP1(0, 'abc')

| This rewrite allows the query optimizer to use the encoded index IX1 to implement the WHERE clause and only cause one invocation of the field procedure for the query.

| **Example 2**

| SELECT col2 FROM T1 ORDER BY col2

| Is represented by the query optimizer as:

| SELECT FP1(4, col2) FROM T1 ORDER BY FP1(4, col2)

| The optimized version removes the FieldProc from the ORDER BY clause assuming that the field procedure QAQQINI option is set to *ALLOW_RANGE or *ALL:

| SELECT FP1(4, col2) FROM T1 ORDER BY col2

| **Example 3**

| Select col2, COUNT(*) FROM T1 GROUP BY col2

| Is represented by the query optimizer as:

| Select FP1(4, col2), COUNT(*) FROM T1 GROUP BY FP1(4, col2)

| The optimized version removes the field procedure invocation from the GROUP BY clause column col2, allowing it to group the encoded data and only run the field procedure once per group. The decoded grouped data is returned to the user. This optimization is done if the field procedure QAQQINI option is set to *ALLOW_RANGE or *ALL:

```
| SELECT FP1(4, col2), COUNT(*) FROM T1 GROUP BY col2
```

| IS NULL/IS NOT NULL predicate does not require calling the field procedure field-decode option 4. The field procedure cannot change the nullability of the field.

| SYSTOOLS

| SYSTOOLS is a set of DB2 for IBM i supplied examples and tools.

| SYSTOOLS is the name of a Database supplied schema (library). SYSTOOLS differs from other DB2 for i supplied schemas (QSYS, QSYS2, SYSIBM, and SYSIBMADM) in that it is not part of the default system path. As general purpose useful tools or examples are built by IBM, they are considered for inclusion within SYSTOOLS. SYSTOOLS provides a wider audience with the opportunity to extract value from the tools.

| It is the intention of IBM to add content dynamically to SYSTOOLS, either on base releases or through PTFs for field releases. A best practice for customers who are interested in such tools would be to periodically review the contents of SYSTOOLS.

| Using SYSTOOLS

| You can generate the sample SQL procedures, learn how to call the procedures, and understand the outcome that is expected. You can also modify the procedure source to customize an example into your business operations.

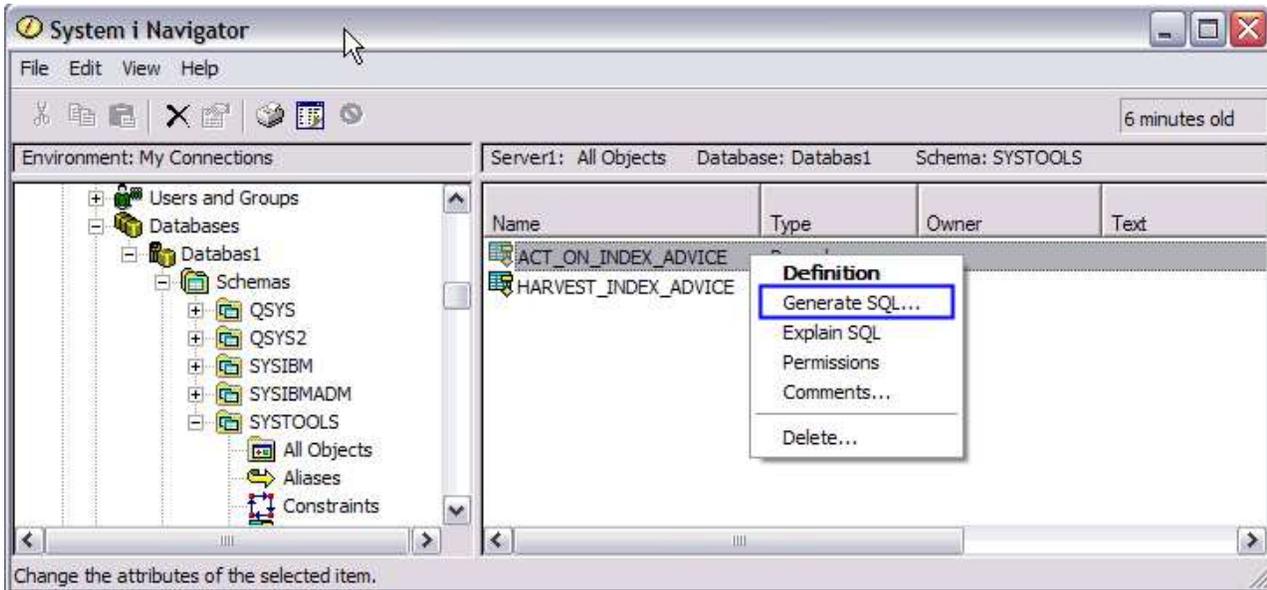
| Use System i Navigator, as shown in Figure 1.

| Figure 1. System i Navigator schema view of SYSTOOLS:



Start with the Generate SQL action, as shown in Figure 2, to discover and learn within SYSTOOLS. This action utilizes the Generate Data Definition Language (QSQGNDDL) API to produce the CREATE PROCEDURE (SQL) statement. This statement is needed to create a replica of the IBM supplied procedure.

Figure 2. Launching Generate SQL from System i Navigator:



After the Generate SQL action completes, as shown in figure 3, you will have a Run SQL Scripts window active, allowing you to do the following:

1. Scroll down and read the procedure prolog.
2. Understand how to call the procedure and the outcome that is expected.
3. Modify the procedure source, including the procedure name and schema. This capability could be the most useful aspect of SYSTOOLS, allowing you to quickly claim and customize an IBM supplied example into your business operations.

Figure 3. Run SQL Scripts view of the generated SQL:

```

-- Example: PROCEDURE SYSTOOLS.ACT_ON_INDEX_ADVICE
--
-- Created on July 16, 2009 by Bob Smith (IBMer@us.ibm.com)
--
-- Disclaimer:
-- This example is being provided by IBM to allow IBM i users to understand how index advice
-- could be consumed to improve an index strategy. The creation of indexes can be time consuming
-- and having seldom used indexes may result in a performance degradation. As with any index
-- strategy, it is recommended that you carefully consider the performance characteristics of
-- your application prior to creating new indexes and that you evaluate the index usage
-- statistics.
--
-- While efforts were made to verify the completeness and accuracy of this sample procedure,
-- this sample is provided 'as is' without any warranty whatsoever and to the maximum extent permitted,
-- IBM disclaims all implied warranties.
--
-- Parameters:
-- 1) P_LIBRARY - The system name of the library which contains the file (table)
-- 2) P_FILE - The system name of the file (table) which may have advised indexes
-- 3) P_TIMES_ADVISED - The number of times an index should be advised before creation of a permanent index.
-- Pass in 1 if you don't want to limit the index creation by times advised.
-- 4) P_MTI_USED - The number of times an Maintained Temporary Index has been used since a matching
-- permanent index did not exist.
-- Pass in 0 if you don't want to limit the index creation by MTI used.
-- 5) P_AVERAGE_QUERY_ESTIMATE - The average estimated number of seconds needed to execute the query
-- which drove the index advice.
-- Pass in 1 if you don't want to limit the index creation by average query estimate.
--
-- Other columns which might be useful parameters to gauge index advisor consumption are:
-- LAST_ADVISED
-- ESTIMATED_CREATION_TIME
-- MOST_EXPENSIVE_QUERY
-- TABLE_SIZE
-- MTI_CREATED
-- LAST_MTI_USED
--
-- QSYS2.SYSIXADV documentation can be found here:
-- http://publib.boulder.ibm.com/infocenter/iserics/v6r1m0/topic/rzajq/rzajqindexcols.htm
--
-- Note: This procedure is hard-coded to work against index advice with Sort Sequence = '*HEX';
-- If you want it to work against index advice with sort sequence tables, search for *HEX and replace
-- with the NLSS schema and library. This restriction is necessary because the procedure should be
-- running with a sort sequence which matches the sort sequence of the index advice.
--
-- DECLARE V_DYNSTMT VARCHAR ( 30000 ) ;
-- DECLARE V_INDEXNAME VARCHAR ( 128 ) ;
-- DECLARE V_COUNT INTEGER DEFAULT 0 ;
-- DECLARE V1 INT DEFAULT 0 ;

```

The IBM maintenance of SYSTOOLS includes periodically dropping and recreating the IBM supplied objects. Customers are allowed to create their own objects within SYSTOOLS. However, if your user created objects conflict with the IBM supplied objects, your objects might be deleted. The tools and examples within SYSTOOLS are considered ready for use. However, they are not subject to IBM Service and Support as they are not considered part of any IBM product.

Database monitor formats

This section contains the formats used to create the database monitor SQL tables and views.

Database monitor SQL table format

Displays the format used to create the QSYS/QAQQDBMN performance statistics table, that is shipped with the system.

```

CREATE TABLE QSYS.QAQQDBMN (
  QQRID DECIMAL(15, 0) NOT NULL DEFAULT 0 ,
  QQTIME TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
  QQJFLD CHAR(46) CCSID 65535 NOT NULL DEFAULT '' ,
  QQRDBN CHAR(18) CCSID 37 NOT NULL DEFAULT '' ,
  QQSYS CHAR(8) CCSID 37 NOT NULL DEFAULT '' ,
  QQJOB CHAR(10) CCSID 37 NOT NULL DEFAULT '' ,
  QQUSER CHAR(10) CCSID 37 NOT NULL DEFAULT '' ,
  QQJNUM CHAR(6) CCSID 37 NOT NULL DEFAULT '' ,
  QQUCNT DECIMAL(15, 0) DEFAULT NULL ,
  QQUDEF VARCHAR(100) CCSID 37 DEFAULT NULL ,
  QQSTN DECIMAL(15, 0) DEFAULT NULL ,
  QQQDTN DECIMAL(15, 0) DEFAULT NULL ,
  QQQDTL DECIMAL(15, 0) DEFAULT NULL ,
  QQMATN DECIMAL(15, 0) DEFAULT NULL ,
  QQMATL DECIMAL(15, 0) DEFAULT NULL ,
  QQTLN CHAR(10) CCSID 37 DEFAULT NULL ,
  QQTFN CHAR(10) CCSID 37 DEFAULT NULL ,
  QQTMN CHAR(10) CCSID 37 DEFAULT NULL ,
  QQPTLN CHAR(10) CCSID 37 DEFAULT NULL ,
  QQPTFN CHAR(10) CCSID 37 DEFAULT NULL ,
  QQPTMN CHAR(10) CCSID 37 DEFAULT NULL ,
  QQILNM CHAR(10) CCSID 37 DEFAULT NULL ,
  QQIFNM CHAR(10) CCSID 37 DEFAULT NULL ,
  QQIMNM CHAR(10) CCSID 37 DEFAULT NULL ,
  QQNTNM CHAR(10) CCSID 37 DEFAULT NULL ,
  QQNLNM CHAR(10) CCSID 37 DEFAULT NULL ,
  QQSTIM TIMESTAMP DEFAULT NULL ,
  QQETIM TIMESTAMP DEFAULT NULL ,
  QQKP CHAR(1) CCSID 37 DEFAULT NULL ,
  QQKS CHAR(1) CCSID 37 DEFAULT NULL ,
  QQTOTR DECIMAL(15, 0) DEFAULT NULL ,
  QQTMPR DECIMAL(15, 0) DEFAULT NULL ,
  QQJNP DECIMAL(15, 0) DEFAULT NULL ,
  QQEPT DECIMAL(15, 0) DEFAULT NULL ,
  QQDSS CHAR(1) CCSID 37 DEFAULT NULL ,
  QQIDXA CHAR(1) CCSID 37 DEFAULT NULL ,
  QQORDG CHAR(1) CCSID 37 DEFAULT NULL ,
  QQGRPG CHAR(1) CCSID 37 DEFAULT NULL ,
  QQJNG CHAR(1) CCSID 37 DEFAULT NULL ,
  QQUNIN CHAR(1) CCSID 37 DEFAULT NULL ,
  QQSUBQ CHAR(1) CCSID 37 DEFAULT NULL ,
  QQHSTV CHAR(1) CCSID 37 DEFAULT NULL ,
  QQRCDS CHAR(1) CCSID 37 DEFAULT NULL ,
  QQRCOD CHAR(2) CCSID 37 DEFAULT NULL ,
  QQRSS DECIMAL(15, 0) DEFAULT NULL ,
  QQREST DECIMAL(15, 0) DEFAULT NULL ,
  QQRIDX DECIMAL(15, 0) DEFAULT NULL ,
  QQFKEY DECIMAL(15, 0) DEFAULT NULL ,
  QQKSEL DECIMAL(15, 0) DEFAULT NULL ,
  QQAJN DECIMAL(15, 0) DEFAULT NULL ,
  QQIDXV VARCHAR(1000) ALLOCATE(48) CCSID 37 DEFAULT NULL ,
  QQC11 CHAR(1) CCSID 37 DEFAULT NULL ,
  QQC12 CHAR(1) CCSID 37 DEFAULT NULL ,
  QQC13 CHAR(1) CCSID 37 DEFAULT NULL ,
  QQC14 CHAR(1) CCSID 37 DEFAULT NULL ,
  QQC15 CHAR(1) CCSID 37 DEFAULT NULL ,
  QQC16 CHAR(1) CCSID 37 DEFAULT NULL ,
  QQC18 CHAR(1) CCSID 37 DEFAULT NULL ,
  QQC21 CHAR(2) CCSID 37 DEFAULT NULL ,
  QQC22 CHAR(2) CCSID 37 DEFAULT NULL ,
  QQC23 CHAR(2) CCSID 37 DEFAULT NULL ,
  QQI1 DECIMAL(15, 0) DEFAULT NULL ,
  QQI2 DECIMAL(15, 0) DEFAULT NULL ,
  QQI3 DECIMAL(15, 0) DEFAULT NULL ,
  QQI4 DECIMAL(15, 0) DEFAULT NULL ,
  QQI5 DECIMAL(15, 0) DEFAULT NULL ,

```

QQI6 DECIMAL(15, 0) DEFAULT NULL ,
 QQI7 DECIMAL(15, 0) DEFAULT NULL ,
 QQI8 DECIMAL(15, 0) DEFAULT NULL ,
 QQI9 DECIMAL(15, 0) DEFAULT NULL ,
 QQIA DECIMAL(15, 0) DEFAULT NULL ,
 QQF1 DECIMAL(15, 0) DEFAULT NULL ,
 QQF2 DECIMAL(15, 0) DEFAULT NULL ,
 QQF3 DECIMAL(15, 0) DEFAULT NULL ,
 QQC61 CHAR(6) CCSID 37 DEFAULT NULL ,
 QQC81 CHAR(8) CCSID 37 DEFAULT NULL ,
 QQC82 CHAR(8) CCSID 37 DEFAULT NULL ,
 QQC83 CHAR(8) CCSID 37 DEFAULT NULL ,
 QQC84 CHAR(8) CCSID 37 DEFAULT NULL ,
 QQC101 CHAR(10) CCSID 37 DEFAULT NULL ,
 QQC102 CHAR(10) CCSID 37 DEFAULT NULL ,
 QQC103 CHAR(10) CCSID 37 DEFAULT NULL ,
 QQC104 CHAR(10) CCSID 37 DEFAULT NULL ,
 QQC105 CHAR(10) CCSID 37 DEFAULT NULL ,
 QQC106 CHAR(10) CCSID 37 DEFAULT NULL ,
 QQC181 VARCHAR(128) ALLOCATE(18) CCSID 37 DEFAULT NULL ,
 QQC182 VARCHAR(128) ALLOCATE(18) CCSID 37 DEFAULT NULL ,
 QQC183 VARCHAR(128) ALLOCATE(15) CCSID 37 DEFAULT NULL ,
 QQC301 VARCHAR(30) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QQC302 VARCHAR(30) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QQC303 VARCHAR(30) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QQ1000 VARCHAR(1000) ALLOCATE(48) CCSID 37 DEFAULT NULL ,
 QQTIM1 TIMESTAMP DEFAULT NULL ,
 QQTIM2 TIMESTAMP DEFAULT NULL ,
 QVQTBL VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVQLIB VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVPTBL VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVPLIB VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVINAM VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVILIB VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVQTBLI CHAR(1) CCSID 37 DEFAULT NULL ,
 QVPTBLI CHAR(1) CCSID 37 DEFAULT NULL ,
 QVINAMI CHAR(1) CCSID 37 DEFAULT NULL ,
 QVBNDY CHAR(1) CCSID 37 DEFAULT NULL ,
 QVJFANO CHAR(1) CCSID 37 DEFAULT NULL ,
 QVPARPF CHAR(1) CCSID 37 DEFAULT NULL ,
 QVPARPL CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC11 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC12 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC13 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC14 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC15 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC16 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC17 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC18 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC19 CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC1A CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC1B CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC1C CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC1D CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC1E CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC1F CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC11 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC12 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC13 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC14 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC15 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC16 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC17 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC18 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC19 CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC1A CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC1B CHAR(1) CCSID 37 DEFAULT NULL ,

QWC1C CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC1D CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC1E CHAR(1) CCSID 37 DEFAULT NULL ,
 QWC1F CHAR(1) CCSID 37 DEFAULT NULL ,
 QVC21 CHAR(2) CCSID 37 DEFAULT NULL ,
 QVC22 CHAR(2) CCSID 37 DEFAULT NULL ,
 QVC23 CHAR(2) CCSID 37 DEFAULT NULL ,
 QVC24 CHAR(2) CCSID 37 DEFAULT NULL ,
 QVCTIM DECIMAL(15, 0) DEFAULT NULL ,
 QVPARD DECIMAL(15, 0) DEFAULT NULL ,
 QVPARU DECIMAL(15, 0) DEFAULT NULL ,
 QVPARRC DECIMAL(15, 0) DEFAULT NULL ,
 QVRCNT DECIMAL(15, 0) DEFAULT NULL ,
 QVFILES DECIMAL(15, 0) DEFAULT NULL ,
 QVP151 DECIMAL(15, 0) DEFAULT NULL ,
 QVP152 DECIMAL(15, 0) DEFAULT NULL ,
 QVP153 DECIMAL(15, 0) DEFAULT NULL ,
 QVP154 DECIMAL(15, 0) DEFAULT NULL ,
 QVP155 DECIMAL(15, 0) DEFAULT NULL ,
 QVP156 DECIMAL(15, 0) DEFAULT NULL ,
 QVP157 DECIMAL(15, 0) DEFAULT NULL ,
 QVP158 DECIMAL(15, 0) DEFAULT NULL ,
 QVP159 DECIMAL(15, 0) DEFAULT NULL ,
 QVP15A DECIMAL(15, 0) DEFAULT NULL ,
 QVP15B DECIMAL(15, 0) DEFAULT NULL ,
 QVP15C DECIMAL(15, 0) DEFAULT NULL ,
 QVP15D DECIMAL(15, 0) DEFAULT NULL ,
 QVP15E DECIMAL(15, 0) DEFAULT NULL ,
 QVP15F DECIMAL(15, 0) DEFAULT NULL ,
 QVC41 CHAR(4) CCSID 37 DEFAULT NULL ,
 QVC42 CHAR(4) CCSID 37 DEFAULT NULL ,
 QVC43 CHAR(4) CCSID 37 DEFAULT NULL ,
 QVC44 CHAR(4) CCSID 37 DEFAULT NULL ,
 QVC81 CHAR(8) CCSID 37 DEFAULT NULL ,
 QVC82 CHAR(8) CCSID 37 DEFAULT NULL ,
 QVC83 CHAR(8) CCSID 37 DEFAULT NULL ,
 QVC84 CHAR(8) CCSID 37 DEFAULT NULL ,
 QVC85 CHAR(8) CCSID 37 DEFAULT NULL ,
 QVC86 CHAR(8) CCSID 37 DEFAULT NULL ,
 QVC87 CHAR(8) CCSID 37 DEFAULT NULL ,
 QVC88 CHAR(8) CCSID 37 DEFAULT NULL ,
 QVC101 CHAR(10) CCSID 37 DEFAULT NULL ,
 QVC102 CHAR(10) CCSID 37 DEFAULT NULL ,
 QVC103 CHAR(10) CCSID 37 DEFAULT NULL ,
 QVC104 CHAR(10) CCSID 37 DEFAULT NULL ,
 QVC105 CHAR(10) CCSID 37 DEFAULT NULL ,
 QVC106 CHAR(10) CCSID 37 DEFAULT NULL ,
 QVC107 CHAR(10) CCSID 37 DEFAULT NULL ,
 QVC108 CHAR(10) CCSID 37 DEFAULT NULL ,
 QVC1281 VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVC1282 VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVC1283 VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVC1284 VARCHAR(128) ALLOCATE(10) CCSID 37 DEFAULT NULL ,
 QVC3001 VARCHAR(300) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC3002 VARCHAR(300) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC3003 VARCHAR(300) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC3004 VARCHAR(300) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC3005 VARCHAR(300) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC3006 VARCHAR(300) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC3007 VARCHAR(300) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC3008 VARCHAR(300) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC5001 VARCHAR(500) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC5002 VARCHAR(500) ALLOCATE(32) CCSID 37 DEFAULT NULL ,
 QVC1000 VARCHAR(1000) ALLOCATE(48) CCSID 37 DEFAULT NULL ,
 QWC1000 VARCHAR(1000) ALLOCATE(48) CCSID 37 DEFAULT NULL ,
 QQINT01 INTEGER DEFAULT NULL ,
 QQINT02 INTEGER DEFAULT NULL ,

```

QQINT03 INTEGER DEFAULT NULL ,
QQINT04 INTEGER DEFAULT NULL ,
QQSMINT1 SMALLINT DEFAULT NULL ,
QQSMINT2 SMALLINT DEFAULT NULL ,
QQSMINT3 SMALLINT DEFAULT NULL ,
QQSMINT4 SMALLINT DEFAULT NULL ,
QQSMINT5 SMALLINT DEFAULT NULL ,
QQSMINT6 SMALLINT DEFAULT NULL ,
QQ1000L CLOB(2147483647) ALLOCATE(48) CCSID 37 DEFAULT NULL ,
QFC11 CHAR(1) CCSID 37 DEFAULT NULL ,
QFC12 CHAR(1) CCSID 37 DEFAULT NULL ,
QFC13 CHAR(1) CCSID 37 DEFAULT NULL ,
QQCLOB2 CLOB(2147483647) ALLOCATE(48) CCSID 37 DEFAULT NULL ,
QFC14 CHAR(1) CCSID 37 DEFAULT NULL ,
QFC15 CHAR(1) CCSID 37 DEFAULT NULL ,
QFC16 CHAR(1) CCSID 37 DEFAULT NULL ,
QQCLOB3 CLOB(2147483647) CCSID 37 DEFAULT NULL ,
QFC17 CHAR(1) CCSID 37 DEFAULT NULL ,
QFC18 CHAR(1) CCSID 37 DEFAULT NULL ,
QFC19 CHAR(1) CCSID 37 DEFAULT NULL ,
QQDBCLOB1 DBCLOB(1073741823) ALLOCATE(24) CCSID 1200 DEFAULT NULL ,
QFC1A CHAR(1) CCSID 37 DEFAULT NULL ,
QFC1B CHAR(1) CCSID 37 DEFAULT NULL ,
QFC1C CHAR(1) CCSID 37 DEFAULT NULL ,
QQDBCLOB2 DBCLOB(1073741823) CCSID 1200 DEFAULT NULL ,
QFC1D CHAR(1) CCSID 37 DEFAULT NULL ,
QFC1E CHAR(1) CCSID 37 DEFAULT NULL ,
QFC1F CHAR(1) CCSID 37 DEFAULT NULL ,
QQBLOB1 BLOB(2147483647) DEFAULT NULL ,
QXC11 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC12 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC13 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC14 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC15 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC16 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC17 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC18 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC19 CHAR(1) CCSID 37 DEFAULT NULL ,
QXC1A CHAR(1) CCSID 37 DEFAULT NULL ,
QXC1B CHAR(1) CCSID 37 DEFAULT NULL ,
QXC1C CHAR(1) CCSID 37 DEFAULT NULL ,
QXC1D CHAR(1) CCSID 37 DEFAULT NULL ,
QXC1E CHAR(1) CCSID 37 DEFAULT NULL ,
QXC21 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC22 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC23 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC24 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC25 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC26 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC27 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC28 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC29 CHAR(2) CCSID 37 DEFAULT NULL ,
QXC41 CHAR(4) CCSID 37 DEFAULT NULL ,
QXC42 CHAR(4) CCSID 37 DEFAULT NULL ,
QXC43 CHAR(4) CCSID 65535 DEFAULT NULL ,
QXC44 CHAR(4) CCSID 37 DEFAULT NULL ,
QQINT05 INTEGER DEFAULT NULL ,
QQINT06 INTEGER DEFAULT NULL ,
QQINT07 INTEGER DEFAULT NULL ,
QQINT08 INTEGER DEFAULT NULL ,
QQINT09 INTEGER DEFAULT NULL ,
QQINT0A INTEGER DEFAULT NULL ,
QQINT0B INTEGER DEFAULT NULL ,
QQINT0C INTEGER DEFAULT NULL ,
QQINT0D INTEGER DEFAULT NULL ,
QQINT0E INTEGER DEFAULT NULL ,
QQINT0F INTEGER DEFAULT NULL ,

```

```

QQSMINT7 SMALLINT DEFAULT NULL ,
QQSMINT8 SMALLINT DEFAULT NULL ,
QQSMINT9 SMALLINT DEFAULT NULL ,
QQSMINTA SMALLINT DEFAULT NULL ,
QQSMINTB SMALLINT DEFAULT NULL ,
QQSMINTC SMALLINT DEFAULT NULL ,
QQSMINTD SMALLINT DEFAULT NULL ,
QQSMINTE SMALLINT DEFAULT NULL ,
QQSMINTF SMALLINT DEFAULT NULL )

```

```

RCDFMT QQQDBMN ;
RENAME QSYS/QQQDBMN TO SYSTEM NAME QAQQDBMN;

```

```

LABEL ON TABLE QSYS/QAQQDBMN
IS 'Database Monitor Physical File' ;

```

```

LABEL ON COLUMN QSYS.QAQQDBMN
( QQRID IS 'Record ID' ,
  QQTIME IS 'Created Time' ,
  QQJFLD IS 'Join Column' ,
  QQRDBN IS 'Relational Database Name' ,
  QQSYS IS 'System Name' ,
  QQJOB IS 'Job Name' ,
  QQUSER IS 'Job User' ,
  QQJNUM IS 'Job Number' ,
  QQUCNT IS 'Unique Counter' ,
  QQUDEF IS 'User Defined Column' ,
  QQSTN IS 'Statement Number' ,
  QQQDTN IS 'Subselect Number' ,
  QQQDTL IS 'Subselect Nested Level' ,
  QQMATN IS 'Subselect Number of Materialized View' ,
  QQMATL IS 'Subselect Level of Materialized View' ,
  QQTLN IS 'Library of Table Queried' ,
  QQTFN IS 'Name of Table Queried' ,
  QQTMN IS 'Member of Table Queried' ,
  QQPTLN IS 'Library of Base Table' ,
  QQPTFN IS 'Name of Base Table' ,
  QQPTMN IS 'Member of Base Table' ,
  QQILNM IS 'Library of Index Used' ,
  QQIFNM IS 'Name of Index Used' ,
  QQIMNM IS 'Member of Index Used' ,
  QQNTNM IS 'NLSS Table' ,
  QQNLNM IS 'NLSS Library' ,
  QQSTIM IS 'Start Time' ,
  QQETIM IS 'End Time' ,
  QQKP IS 'Key Positioning' ,
  QQKS IS 'Key Selection' ,
  QQTOTR IS 'Total Rows' ,
  QQTMPR IS 'Number of Rows in Temporary' ,
  QQJNP IS 'Join Position' ,
  QQEPT IS 'Estimated Processing Time' ,
  QQDSS IS 'Data Space Selection' ,
  QQIDXA IS 'Index Advised' ,
  QQORDG IS 'Ordering' ,
  QQGRPG IS 'Grouping' ,
  QQJNG IS 'Join' ,
  QQUNIN IS 'Union' ,
  QQSUBQ IS 'Subquery' ,
  QQHSTV IS 'Host Variables' ,
  QQRCDS IS 'Row Selection' ,
  QQRCOD IS 'Reason Code' ,
  QQRSS IS 'Number of Rows Selected' ,
  QQREST IS 'Estimated Number of Rows Selected' ,
  QQRIDX IS 'Number of Entries in Index Created' ,
  QQFKEY IS 'Estimated Entries for Key Positioning' ,
  QQKSEL IS 'Estimated Entries for Key Selection' ,
  QQAJN IS 'Estimated Number of Joined Rows' ,

```

```

QQIDXD IS 'Advised          Key          Columns' ,
QQI9 IS 'Thread            Identifier' ,
QVQTBL IS 'Queried         Table          Long Name' ,
QVQLIB IS 'Queried         Library        Long Name' ,
QVPTBL IS 'Base           Table          Long Name' ,
QVPLIB IS 'Base           Library        Long Name' ,
QVINAM IS 'Index Used     Long Name' ,
QVILIB IS 'Index Used     Library        Name' ,
QVQTBLI IS 'Table         Long          Required' ,
QVPTBLI IS 'Base         Long          Required' ,
QVINAMI IS 'Index         Long          Required' ,
QVBNDY IS 'I/O or CPU     Bound' ,
QVJFANO IS 'Join          Fan           Out' ,
QVPARPF IS 'Parallel      Pre-Fetch' ,
QVPARPL IS 'Parallel      Pre-Load' ,
QVCTIM IS 'Estimated     Cumulative    Time' ,
QVPARD IS 'Parallel      Degree        Requested' ,
QVPARU IS 'Parallel      Degree        Used' ,
QVPARRC IS 'Parallel     Limited       Reason Code' ,
QVRCNT IS 'Refresh       Count' ,
QVFILES IS 'Number of    Tables        Joined' ) ;

```

LABEL ON COLUMN QSYS.QAQQDBMN

```

( QQRID TEXT IS 'Record ID' ,
  QQTIME TEXT IS 'Time record was created' ,
  QQJFLD TEXT IS 'Join Column' ,
  QQRDBN TEXT IS 'Relational Database Name' ,
  QSYS TEXT IS 'System Name' ,
  QQJOB TEXT IS 'Job Name' ,
  QQUSER TEXT IS 'Job User' ,
  QQJNUM TEXT IS 'Job Number' ,
  QQUCNT TEXT IS 'Unique Counter' ,
  QQUDEF TEXT IS 'User Defined Column' ,
  QQSTN TEXT IS 'Statement Number' ,
  QQQDTN TEXT IS 'Subselect Number' ,
  QQQDTL TEXT IS 'Subselect Nested Level' ,
  QQMATN TEXT IS 'Subselect Number of Materialized View' ,
  QQMATL TEXT IS 'Subselect Level of Materialized View' ,
  QQTLN TEXT IS 'Library of Table Queried' ,
  QQTFN TEXT IS 'Name of Table Queried' ,
  QQTMN TEXT IS 'Member of Table Queried' ,
  QQPTLN TEXT IS 'Base Table Library' ,
  QQPTFN TEXT IS 'Base Table' ,
  QQPTMN TEXT IS 'Base Table Member' ,
  QQILNM TEXT IS 'Library of Index Used' ,
  QQIFNM TEXT IS 'Name of Index Used' ,
  QQIMNM TEXT IS 'Member of Index Used' ,
  QQNTNM TEXT IS 'NLSS Table' ,
  QQNLNM TEXT IS 'NLSS Library' ,
  QQSTIM TEXT IS 'Start timestamp' ,
  QQETIM TEXT IS 'End timestamp' ,
  QQKP TEXT IS 'Key positioning' ,
  QQKS TEXT IS 'Key selection' ,
  QQTOTR TEXT IS 'Total row in table' ,
  QQTMPR TEXT IS 'Number of rows in temporary' ,
  QQJNP TEXT IS 'Join Position' ,
  QQEPT TEXT IS 'Estimated processing time' ,
  QQDSS TEXT IS 'Data Space Selection' ,
  QQIDXA TEXT IS 'Index advised' ,
  QQORDG TEXT IS 'Ordering' ,
  QQGRPG TEXT IS 'Grouping' ,
  QQJNG TEXT IS 'Join' ,
  QQUNIN TEXT IS 'Union' ,
  QQSUBQ TEXT IS 'Subquery' ,
  QQHSTV TEXT IS 'Host Variables' ,
  QQRCD S TEXT IS 'Row Selection' ,
  QQRCD TEXT IS 'Reason Code' ,

```

```

QQRSS TEXT IS 'Number of rows selected or sorted' ,
QQRST TEXT IS 'Estimated number of rows selected' ,
QQRIDX TEXT IS 'Number of entries in index created' ,
QQFKEY TEXT IS 'Estimated keys for key positioning' ,
QQKSEL TEXT IS 'Estimated keys for key selection' ,
QQAJN TEXT IS 'Estimated number of joined rows' ,
QQIDXO TEXT IS 'Key columns for the index advised' ,
QQI9 TEXT IS 'Thread Identifier' ,
QVQTBL TEXT IS 'Queried Table, Long Name' ,
QVQLIB TEXT IS 'Queried Library, Long Name' ,
QVPTBL TEXT IS 'Base Table, Long Name' ,
QVPLIB TEXT IS 'Base Library, Long Name' ,
QVINAM TEXT IS 'Index Used, Long Name' ,
QVILIB TEXT IS 'Index Used, Library Name' ,
QVQTBLI TEXT IS 'Table Long Required' ,
QVPTBLI TEXT IS 'Base Long Required' ,
QVINAMI TEXT IS 'Index Long Required' ,
QVBNDY TEXT IS 'I/O or CPU Bound' ,
QVJFANO TEXT IS 'Join Fan out' ,
QVPPRF TEXT IS 'Parallel Pre-Fetch' ,
QVPPRL TEXT IS 'Parallel Pre-Load' ,
QVCTIM TEXT IS 'Cumulative Time' ,
QVPARD TEXT IS 'Parallel Degree, Requested' ,
QVPARU TEXT IS 'Parallel Degree, Used' ,
QVPARRC TEXT IS 'Parallel Limited, Reason Code' ,
QVRCNT TEXT IS 'Refresh Count' ,
QVFILES TEXT IS 'Number of, Tables Joined' ) ;

```

Optional database monitor SQL view format

These examples show the different optional SQL view format that you can create with the SQL shown. The column descriptions are explained in the tables following each example. These views are not shipped with the system, and you must create them, if you choose to do so. These views are optional and are not required for analyzing monitor data.

Any rows that have a row identification number (QQRID) of 5000 or greater are for internal database use.

Database monitor view 1000 - SQL Information

Displays the SQL logical view format for database monitor QQQ1000.

```

| Create View QQQ1000 as
|   (SELECT QQRID as Row_ID,
|         QQTIME as Time_Created,
|         QQJFLD as Join_Column,
|         QQRDBN as Relational_Database_Name,
|         QQSYS as System_Name,
|         QQJOB as Job_Name,
|         QQUSER as Job_User,
|         QQJNUM as Job_Number,
|         QQI9 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQI5 as Unique_Refresh_Counter,
|         QQUDEF as User_Defined,
|         QQSTN as Statement_Number,
|         QQC11 as Statement_Function,
|         QQC21 as Statement_Operation,
|         QQC12 as Statement_Type,
|         QQC13 as Parse_Required,
|         QQC103 as Package_Name,
|         QQC104 as Package_Library,
|         QQC181 as Cursor_Name,
|         QQC182 as Statement_Name,
|         QQSTIM as Start_Timestamp,
|         QQ1000 as Statement_Text,
|         QQC14 as Statement_Outcome,
|         QQI2 as Result_Rows,

```

QQC22 as Dynamic_Replan_Reason_Code,
 QQC16 as Data_Conversion_Reason_Code,
 QQI4 as Total_Time_Milliseconds,
 QQI3 as Rows_Fetched,
 QQETIM as End_Timestamp,
 QQI6 as Total_Time_Microseconds,
 QQI7 as SQL_Statement_Length,
 QQI1 as Insert_Unique_Count,
 QQI8 as SQLCode,
 QQC81 as SQLState,
 QVC101 as Close_Cursor_Mode,
 QVC11 as Allow_Copy_Data_Value,
 QVC12 as PseudoOpen,
 QVC13 as PseudoClose,
 QVC14 as ODP_Implementation,
 QVC21 as Dynamic_Replan_SubCode,
 QVC41 as Commitment_Control_Level,
 QWC1B as Concurrent_Access_Resolution,
 QVC15 as Blocking_Type,
 QVC16 as Delay_Prepare,
 QVC1C as Explainable,
 QVC17 as Naming_Convention,
 QVC18 as Dynamic_Processing_Type,
 QVC19 as LOB_Data_Optimized,
 QVC1A as Program_User_Profile_Used,
 QVC1B as Dynamic_User_Profile_Used,
 QVC1281 as Default_Collection,
 QVC1282 as Procedure_Name,
 QVC1283 as Procedure_Library,
 QQCLOB2 as SQL_Path,
 QVC1284 as Current_Schema,
 QQC18 as Binding_Type,
 QQC61 as Cursor_Type,
 QVC1D as Statement_Originator,
 QQC15 as Hard_Close_Reason_Code,
 QQC23 as Hard_Close_Subcode,
 QVC42 as Date_Format,
 QWC11 as Date_Separator,
 QVC43 as Time_Format,
 QWC12 as Time_Separator,
 QWC13 as Decimal_Point,
 QVC104 as Sort_Sequence_Table ,
 QVC105 as Sort_Sequence_Library,
 QVC44 as Language_ID,
 QVC23 as Country_ID,
 QQIA as First_N_Rows_Value,
 QQF1 as Optimize_For_N_Rows_Value,
 QVC22 as SQL_Access_Plan_Reason_Code,
 QVC24 as Access_Plan_Not_Saved_Reason_Code,
 QVC81 as Transaction_Context_ID,
 QVP152 as Activation_Group_Mark,
 QVP153 as Open_Cursor_Threshold,
 QVP154 as Open_Cursor_Close_Count,
 QVP155 as Commitment_Control_Lock_Limit,
 QWC15 as Allow_SQL_Mixed_Constants,
 QWC16 as Suppress_SQL_Warnings,
 QWC17 as Translate_ASCII,
 QWC18 as System_Wide_Statement_Cache,
 QVP159 as LOB_Locator_Threshold,
 QVP156 as Max_Decimal_Precision,
 QVP157 as Max_Decimal_Scale,
 QVP158 as Min_Decimal_Divide_Scale ,
 QWC19 as Unicode_Normalization,
 QQ1000L as Statement_Text_Long,
 QVP15B as Old_Access_Plan_Length,
 QVP15C as New_Access_Plan_Length,
 QVP151 as Fast_Delete_Count,

```

|         QQF2 as Statement_Max_Compression,
|         QVC102 as Current_User_Profile,
|         QVC1E as Expression_Evaluator_Used,
|         QVP15A as Host_Server_Delta,
|         QQC301 as NTS_Lock_Space_Id,
|         QQC183 as IP_Address,
|         QFC11 as IP_Type,
|         QQSMINT2 as IP_Port_Number,
|         QVC3004 as NTS_Transaction_Id,
|         QQSMINT3 as NTS_Format_Id_Length,
|         QQSMINT4 as NTS_Transaction_ID_SubLength,
|         QVRCNT as Unique_Refresh_Counter2,
|         QVP15F as Times_Run,
|         QVP15E as FullOpens,
|         QVC1F as Proc_In_Cache,
|         QWC1A as Combined_Operation,
|         QVC3001 as Client_Applname,
|         QVC3002 as Client_Userid,
|         QVC3003 as Client_Wrkstnname,
|         QVC3005 as Client_Acctng,
|         QVC3006 as Client_Progamid,
|         QVC5001 as Interface_Information,
|         QVC82 as Open_Options,
|         QWC1D as Extended_Indicators,
|         QWC1C as DECFLOAT_Rounding_Mode,
|         QWC1E as SQL_DECFLOAT_Warnings,
|         QVP15D as Worst_Time_Micro,
|         QQINT05 as SQ_Unique_Count,
|         QFC13 as Concurrent_Access_Res_Used,
|         QQSMINT8 as SQL_Scalar_UDFs_Not_Inlined,
|         QVC3007 as Result_Set_Cursor,
|         QFC12 as Implicit_XMLPARSE_Option,
|         QQSMINT7 as SQL_XML_Data_CCsid,
|         QQSMINT5 as OPTIMIZER_USE,
|         QFC14 as XML_Schema_In_Cache
|
| FROM DbMonLib/DbMonTable
| WHERE QQRID=1000)

```

Table 59. QQQ1000 - SQL Information

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|---|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| Unique_Refresh_Counter | QQI5 | Unique refresh counter |
| User_Defined | QQUDEF | User-defined column |
| Statement_Number | QQSTN | Statement number (unique per statement) |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|--------------------|-------------------|---|
| Statement_Function | QQC11 | Statement function: <ul style="list-style-type: none"> • S - Select • U - Update • I - Insert • D - Delete • L - Data definition language • O - Other |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|---------------------|-------------------|--|
| Statement_Operation | QQC21 | <p>Statement operation:</p> <ul style="list-style-type: none"> • AC - Allocate cursor • AD - Allocate descriptor • AF - Alter function • AL - Alter table • AP - Alter procedure • AQ - Alter sequence • AS - Associate locators • CA - Call • CB - Create variable • 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 • DA - Deallocate descriptor • DE - Describe • DI - Disconnect • DL - Delete • DM - Describe parameter marker • DO - Describe procedure • DP - Declare procedure • DR - Drop • DS - Describe cursor • DT - Describe table • EI - Execute immediate • EX - Execute • FE - Fetch • FL - Free locator • GR - Grant • GS - Get descriptor • HC - Hard close • HL - Hold locator • IN - Insert |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|---------------------------------|-------------------|--|
| Statement_Operation (continued) | QQC21 | <ul style="list-style-type: none"> • JR - Server job reused • LK - Lock • LO - Label on • MG - Merge • MT - More text (Deprecated in V5R4) • OP - Open • PD - Prepare and describe • PR - Prepare • QF - OPNQRYF command • QM - Query/400 STRQMQRV command • QQ - QQQQRY() API • QR - RUNQRY command • RB - Rollback to savepoint • RE - Release • RF - Refresh Table • RG - Resignal • RO - Rollback • RS - Release Savepoint • RT - Rename table • RV - Revoke • SA - Savepoint • SC - Set connection • SD - Set descriptor • SE - Set encryption password • SN - Set session user • SI - Select into • SO - Set current degree • SP - Set path • SR - Set result set • SS - Set current schema • ST - Set transaction • SV - Set variable • SX - Set current implicit XMLPARSE option • UP - Update • VI - Values into • X0 - Unknown statement • X1 - Unknown statement • X2 - DRDA (AS) Unknown statement • X3 - Unknown statement • X9 - Internal error • XA - X/Open API • ZD - Host server only activity |
| Statement_Type | QQC12 | Statement type: <ul style="list-style-type: none"> • D - Dynamic statement • S - Static statement |
| Parse_Required | QQC13 | Parse required (Y/N) |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|----------------------------|-------------------|---|
| Package_Name | QQC103 | Name of the package or name of the program that contains the current SQL statement |
| Package_Library | QQC104 | Name of the library containing the package |
| Cursor_Name | QQC181 | Name of the cursor corresponding to this SQL statement, if applicable |
| Statement_Name | QQC182 | Name of statement for SQL statement, if applicable |
| Start_Timestamp | QQSTIM | Time this statement entered |
| Statement_Text | QQ1000 | First 1000 bytes of statement text |
| Statement_Outcome | QQC14 | Statement outcome <ul style="list-style-type: none"> • S - Successful • U - Unsuccessful |
| Result_Rows | QQI2 | Number of result rows returned. Will only be set for the following SQL operations and is 0 for all others: <ul style="list-style-type: none"> • IN - Insert • UP - Update • DL - Delete |
| Dynamic_Replan_Reason_Code | QQC22 | Dynamic replan (access plan rebuilt) <ul style="list-style-type: none"> • NA - No replan. • NR - SQL QDT rebuilt for new release. • A1 - A table or member is not the same object as the one referenced when the access plan was last built. Some reasons why they might be different are: <ul style="list-style-type: none"> – Object was deleted and recreated. – Object was saved and restored. – Library list was changed. – Object was renamed. – Object was moved. – Object was overridden to a different object. – This run is the first run of this query after the object containing the query has been restored. • A2 - Access plan was built to use a reusable Open Data Path (ODP) and the optimizer chose to use a nonreusable ODP for this call. • A3 - Access plan was built to use a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP for this call. • A4 - The number of rows in the table member has changed by more than 10% since the access plan was last built. • A5 - A new index exists over one of the tables in the query. • A6 - An index that was used for this access plan no longer exists or is no longer valid. • A7 - IBM i Query requires the access plan to be rebuilt because of system programming changes. • A8 - The CCSID of the current job is different from the CCSID of the job that last created the access plan. • A9 - The value of one or more of the following values is different for the current job than it was for the job that last created this access plan: <ul style="list-style-type: none"> – date format – date separator – time format – time separator |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|--|-------------------|---|
| Dynamic_Replan_Reason_Code (continued) | QQC22 | <ul style="list-style-type: none"> • AA - The sort sequence table specified is different from 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 Symmetric Multiprocessing has been installed or removed. • AD - The value of the degree query attribute has changed. • AE - A view is either being opened by a high-level language or a view is being materialized. • AF - A user-defined type or user-defined function is not the same object as the one referred to in the access plan; or the SQL Path is not the same as when the access plan was built. • B0 - The options specified have changed as a result of the query options file. • B1 - The access plan was generated with a commitment control level that is different in the current job. • B2 - The access plan was generated with a static cursor answer set size that is different from the previous access plan. • B3 - The query was reoptimized because this run is the first run of the query after it was prepared. This run is the first run with actual parameter marker values. • B4 - The query was reoptimized because referential or check constraints have changed. • B5 - The query was reoptimized because Materialized query tables have changed. • B6 - The query was reoptimized because the value of a host variable changed and the access plan is no longer valid. • B7 - The query was reoptimized because AQP determined that it was beneficial. |
| Data_Conversion_Reason_Code | QQC16 | <p>Data conversion</p> <ul style="list-style-type: none"> • N - No. • 0 - Not applicable. • 1 - Lengths do not match. • 2 - Numeric types do not match. • 3 - C host variable is NUL-terminated. • 4 - Host variable or column is variable length and the other is not variable length. • 5 - Host variable or column is not variable length and the other is variable length. • 6 - Host variable or column is variable length and the other is not variable length. • 7 - CCSID conversion. • 8 - DRDA and NULL capable, variable length, contained in a partial row, derived expression, or blocked fetch with not enough host variables. • 9 - Target table of an insert is not an SQL table. |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|---|-------------------|--|
| Data_Conversion_Reason_Code (continued) | | <ul style="list-style-type: none"> • 10 - Host variable is too short to hold a TIME or TIMESTAMP value being retrieved. • 11 - Host variable is DATE, TIME, or TIMESTAMP and value being retrieved is a character string. • 12 - Too many host variables specified and records are blocked. • 13 - DRDA used for a blocked FETCH. Also, the number of host variables specified in the INTO clause is less than the number of result values in the select list. • 14 - LOB locator used and the commitment control level was not *ALL. |
| Total_Time_Milliseconds | QQI4 | <p>Total time for this statement, in milliseconds. For fetches, the time includes all fetches for this OPEN of the cursor.</p> <p>Note: When monitor files are created when using an SQL Plan Cache snapshot, this time represents the aggregate time for all runs of this query. This time can be divided by the total number of runs, COALESCE(QVP15F,1), to determine an average time for a given run of the query.</p> |
| Rows_Fetched | QQI3 | <p>Total rows fetched for cursor</p> <p>Note: When monitor files are created when using an SQL Plan Cache snapshot, this count represents the aggregate count for all runs of this query. This count can be divided by the total number of runs, COALESCE(QVP15F,1), to determine the average rows fetched for a given query run</p> |
| End_Timestamp | QQETIM | Time SQL request completed |
| Total_Time_Microseconds | QQI6 | <p>Total time for this statement, in microseconds. For fetches, this time includes all fetches for this OPEN of the cursor.</p> <p>Note: When monitor files are created when using an SQL Plan Cache snapshot, this time represents the aggregate time for all runs of this query. This time can be divided by the total number of runs, COALESCE(QVP15F,1), to determine an average time for a given run of the query.</p> |
| SQL_Statement_Length | QQI7 | Length of SQL Statement |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|-----------------------|-------------------|---|
| Insert_Unique_Count | QQI1 | <p>If the operation (QQC21) indicate INSERT (IN), this field contains the 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.</p> <p>If the operation (QQC21) indicates DELETE (DL), this field contains the fast delete reason code.</p> <p>Possible values if the operation is a DELETE are:</p> <ul style="list-style-type: none"> • 0 - Fast delete results unknown or fast delete is not relevant because the delete failed. • 1 - Fast delete was achieved. <p>All other values if the operation is a DELETE indicate the reason that the database was unable to implement the DELETE request using fast delete. Fast delete attempt denied values:</p> <ul style="list-style-type: none"> • 2 - File is a DDM file. • 3 - File is a multi member file. • 4 - File is a distributed file. • 5 - File is a logical file or SQL view. • 6 - File is a parent file. • 7 - File has one or more triggers created over it. • 8 - Number of rows in table is less than 1000 or less that the QAQQINI SQL_FAST_DELETE_ROW_COUNT value in effect for this statement. • 9 - DBMAINT failed. This reason code could appear for many reasons, including the existence of a logical open within this job, pending record changes, ragged save in progress or other possible reasons. • 10 - Failed to acquire an exclusive no read (LENR) lock on the file. • 11 - Failed to acquire an exclusive allow read (LEAR) lock on the file's data space. • 12 - The user does not have *EXECUTE authority to the library. • 51 - A WHERE clause was used on the DELETE. • 52 - QAQQINI SQL_FAST_DELETE_ROW_COUNT indicated to disallow fast delete. |
| SQLCode | QQI8 | SQL return code |
| SQLState | QQC81 | SQLSTATE |
| Close_Cursor_Mode | QVC101 | <p>Close Cursor. Possible values are:</p> <ul style="list-style-type: none"> • *ENDJOB - SQL cursors are closed when the job ends. • *ENDMOD - SQL cursors are closed when the module ends • *ENDPGM - SQL cursors are closed when the program ends. • *ENDSQL - SQL cursors are closed when the first SQL program on the call stack ends. • *ENDACTGRP - SQL cursors are closed when the activation group ends. |
| Allow_Copy_Data_Value | QVC11 | <p>ALWCPYDTA setting (Y/N/O)</p> <ul style="list-style-type: none"> • Y - A copy of the data might be used. • N - Cannot use a copy of the data. • O - The optimizer can choose to use a copy of the data for performance. |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|------------------------------|-------------------|--|
| PseudoOpen | QVC12 | <p>Pseudo Open (Y/N) for SQL operations that can trigger opens.</p> <ul style="list-style-type: none"> • OP - Open • IN - Insert • UP - Update • DL - Delete • SI - Select Into • SV - Set • VI - Values into <p>For all operations, it can be blank.</p> |
| PseudoClose | QVC13 | <p>Pseudo Close (Y/N) for SQL operations that can trigger a close.</p> <ul style="list-style-type: none"> • CL - Close • IN - Insert • UP - Update • DL - Delete • SI - Select Into • SV - Set • VI - Values into <p>For all operations, it can be blank.</p> |
| ODP_Implementation | QVC14 | <p>ODP implementation</p> <ul style="list-style-type: none"> • R - Reusable ODP • N - Nonreusable ODP • ' ' - Column not used |
| Dynamic_Replan_SubCode | QVC21 | Dynamic replan, subtype reason code |
| Commitment_Control_Level | QVC41 | <p>Commitment control level. Possible values are:</p> <ul style="list-style-type: none"> • CS - Cursor stability • CSKL - Cursor stability. Keep exclusive locks. • NC - No commit • RR - Repeatable read • RREL - Repeatable read. Keep exclusive locks. • RS - Read stability • RSEL - Read stability. Keep exclusive locks. • UR - Uncommitted read |
| Concurrent_Access_Resolution | QWC1B | <p>Indicates what method of concurrent access resolution was specified.</p> <ul style="list-style-type: none"> • N - Concurrent access resolution was not specified. • S - SKIP LOCKED DATA clause was specified. • U - USE CURRENTLY COMMITTED clause was specified. • W- WAIT FOR OUTCOME clause was specified. |
| Blocking_Type | QVC15 | <p>Type of blocking. Possible values are:</p> <ul style="list-style-type: none"> • S - Single row, ALWBLK(*READ) • F - Force one row, ALWBLK(*NONE) • L - Limited block, ALWBLK(*ALLREAD) |
| Delay_Prepare | QVC16 | Delay prepare of statement (Y/N). |
| Explainable | QVC1C | The SQL statement is explainable (Y/N). |
| Naming_Convention | QVC17 | <p>Naming convention. Possible values:</p> <ul style="list-style-type: none"> • N - System naming convention • S - SQL naming convention |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|---------------------------|-------------------|---|
| Dynamic_Processing_Type | QVC18 | Type of dynamic processing. <ul style="list-style-type: none"> • E - Extended dynamic • S - System wide cache • L - Local prepared statement |
| LOB_Data_Optimized | QVC19 | Optimize LOB data types (Y/N) |
| Program_User_Profile_Used | QVC1A | User profile used when compiled programs are executed. Possible values are: <ul style="list-style-type: none"> • N = User Profile is determined by naming conventions. For *SQL, USRPRF(*OWNER) is used. For *SYS, USRPRF(*USER) is used. • U = USRPRF(*USER) is used. • O = USRPRF(*OWNER) is used. |
| Dynamic_User_Profile_Used | QVC1B | User profile used for dynamic SQL statements. <ul style="list-style-type: none"> • U = USRPRF(*USER) is used. • O = USRPRF(*OWNER) is used. |
| Default_Collection | QVC1281 | Name of the default collection. |
| Procedure_Name | QVC1282 | Procedure name on CALL to SQL. |
| Procedure_Library | QVC1283 | Procedure library on CALL to SQL. |
| SQL_Path | QQCLOB2 | Path used to find procedures, functions, and user-defined types for static SQL statements. |
| Current_Schema | QVC1284 | SQL current schema. |
| Binding_Type | QQC18 | Binding type: <ul style="list-style-type: none"> • C - Column-wise binding • R - Row-wise binding |
| Cursor_Type | QQC61 | Cursor Type: <ul style="list-style-type: none"> • NSA - Non-scrollable, asensitive, forward only • NSI - Non-scrollable, insensitive, forward only • NSS - Non-scrollable, sensitive, forward only • SCA - scrollable, asensitive • SCI - scrollable, insensitive • SCS - scrollable, sensitive |
| Statement_Originator | QVC1D | SQL statement originator: <ul style="list-style-type: none"> • U - User • S - System |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|------------------------|-------------------|---|
| Hard_Close_Reason_Code | QQC15 | SQL cursor hard close reason. Possible reasons are: <ul style="list-style-type: none"> • 1 - Internal Error • 2 - Exclusive Lock • 3 - Interactive SQL Reuse Restriction • 4 - Host variable Reuse Restriction • 5 - Temporary Result Restriction • 6 - Cursor Restriction • 7 - Cursor Hard Close Requested • 8 - Internal Error • 9 - Cursor Threshold • A - Refresh Error • B - Reuse Cursor Error • C - DRDA AS Cursor Closed • D - DRDA AR Not WITH HOLD • E - Repeatable Read • F - Lock Conflict Or QSQPRCED Threshold - Library • G - Lock Conflict Or QSQPRCED Threshold - File • H - Execute Immediate Access Plan Space • I - QSQCSRTH Dummy Cursor Threshold • J - File Override Change • K - Program Invocation Change • L - File Open Options Change • M - Statement Reuse Restriction • N - Internal Error • O - Library List Changed • P - Exit Processing • Q - SET SESSION USER statement |
| Hard_Close_Subcode | QQC23 | SQL cursor hard close reason subcode. |
| Date_Format | QVC42 | Date Format. Possible values are: <ul style="list-style-type: none"> • ISO • USA • EUR • JIS • JUL • MDY • DMY • YMD |
| Date_Separator | QWC11 | Date Separator. Possible values are: <ul style="list-style-type: none"> • "/" • "." • "," • "-" • " " |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------|-------------------|---|
| Time_Format | QVC43 | Time Format. Possible values are: <ul style="list-style-type: none"> • ISO • USA • EUR • JIS • HMS |
| Time_Separator | QWC12 | Time Separator. Possible values are: <ul style="list-style-type: none"> • ":" • ". • "/" • " " |
| Decimal_Point | QWC13 | Decimal Point. Possible values are: <ul style="list-style-type: none"> • "." • "," |
| Sort_Sequence_Table | QVC104 | Sort Sequence Table |
| Sort_Sequence_Library | QVC105 | Sort Sequence Library |
| Language_ID | QVC44 | Language ID |
| Country_ID | QVC23 | Country ID |
| First_N_Rows_Value | QQIA | Value specified on the FIRST n ROWS clause. |
| Optimize_For_N_Rows_Value | QQF1 | Value specified on the OPTIMIZE FOR n ROWS clause. |
| SQL_Access_Plan_Reason_Code | QVC22 | SQL access plan rebuild reason code. Possible reasons are: <ul style="list-style-type: none"> • A1 - A table or member is not the same object as the one referenced when the access plan was last built. Some reasons they might be different are: <ul style="list-style-type: none"> – Object was deleted and recreated. – Object was saved and restored. – Library list was changed. – Object was renamed. – Object was moved. – Object was overridden to a different object. – This rebuild is the first run of this query after the object containing the query has been restored. • A2 - Access plan was built to use a reusable Open Data Path (ODP) and the optimizer chose to use a non-reusable ODP for this call. • A3 - Access plan was built to use a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP for this call. • A4 - The number of rows in the table has changed by more than 10% since the access plan was last built. • A5 - A new index exists over one of the tables in the query • A6 - An index that was used for this access plan no longer exists or is no longer valid. |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|--|-------------------|--|
| SQL_Access_Plan_Reason_Code (continued) | | <ul style="list-style-type: none"> • A7 - IBM i Query requires the access plan to be rebuilt because of system programming changes. • A8 - The CCSID of the current job is different from the CCSID of the job that last created the access plan. • A9 - One or more of the following values is different for the current job than it was for the job that last created this access plan: <ul style="list-style-type: none"> – date format – date separator – time format – time separator. • AA - The sort sequence table specified is different from 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 Symmetric Multiprocessing has been installed or removed. • AD - The value of the degree query attribute has changed. • AE- A view is either being opened by a high-level language or a view is being materialized. • AF - A user-defined type or user-defined function is not the same object as the one referred to in the access plan, or, the SQL Path is not the same as when the access plan was built. • B0 - The options specified have changed as a result of the query options file. • B1 - The access plan was generated with a commitment control level that is different in the current job. • B2 - The access plan was generated with a static cursor answer set size that is different from the previous access plan. • B3 - The query was reoptimized because this run is the first run after the query was prepared. It is the first run with actual parameter marker values. • B4 - The query was reoptimized because referential or check constraints have changed. • B5 - The query was reoptimized because Materialized query tables have changed. • B6 - The query was reoptimized because the value of a host variable changed and the access plan is no longer valid. • B7 - The query was reoptimized because AQP determined that the query must be reoptimized. |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------------|-------------------|--|
| Access_Plan_Not_Saved_Reason_Code | QVC24 | <p>Access plan not saved reason code. Possible reasons are:</p> <ul style="list-style-type: none"> • A1 - Failed to get an LSUP lock on associated space of program or package. • A2 - Failed to get an immediate LEAR space location lock on first byte of associated space of program. • A3 - Failed to get an immediate LENR space location lock on first byte of associated space of program. • A5 - Failed to get an immediate LEAR space location lock on first byte of ILE associated space of a program. • A6 - Error trying to extend space of an ILE program. • A7 - No room in program. • A8 - No room in program associated space. • A9 - No room in program associated space. • AA - No need to save. Save already done in another job. • AB - Query optimizer cannot lock the QDT. • B1 - Saved at the end of the program associated space. • B2 - Saved at the end of the program associated space. • B3 - Saved in place. • B4 - Saved in place. • B5 - Saved at the end of the program associated space. • B6 - Saved in place. • B7 - Saved at the end of the program associated space. • B8 - Saved at the end of the program associated space. |
| Transaction_Context_ID | QVC81 | Transaction context ID. |
| Activation_Group_Mark | QVP152 | Activation Group Mark |
| Open_Cursor_Threshold | QVP153 | Open cursor threshold |
| Open_Cursor_Close_Count | QVP154 | Open cursor close count |
| Commitment_Control_Lock_Limit | QVP155 | Commitment control lock limit |
| Allow_SQL_Mixed_Constants | QWC15 | Using SQL mixed constants (Y/N) |
| Suppress_SQL_Warnings | QWC16 | Suppress SQL warning messages (Y/N) |
| Translate_ASCII | QWC17 | Translate ASCII to job (Y/N) |
| System_Wide_Statement_Cache | QWC18 | Using system-wide SQL statement cache (Y/N) |
| LOB_Locator_Threshold | QVP159 | LOB locator threshold |
| Max_Decimal_Precision | QVP156 | Maximum decimal precision (63/31) |
| Max_Decimal_Scale | QVP157 | Maximum decimal scale |
| Min_Decimal_Divide_Scale | QVP158 | Minimum decimal divide scale |
| Unicode_Normalization | QWC19 | Unicode data normalization requested (Y/N) |
| Statement_Text_Long | QQ1000L | Complete statement text |
| Old_Access_Plan_Length | QVP15B | Length of old access plan |
| New_Access_Plan_Length | QVP15C | Length of new access plan |
| Fast_Delete_Count | QVP151 | <p>SQL fast count delete count. Possible values are:</p> <ul style="list-style-type: none"> • 0 = *OPTIMIZE or *DEFAULT • 1-999,999,999,999 = User specified value • 'FFFFFFFFFFFFFFFF'x = *NONE |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|------------------------------|-------------------|--|
| Statement_Max_Compression | QQF2 | SQL statement maximum compression. Possible values are: <ul style="list-style-type: none"> • 1 - *DEFAULT • 1 - User specified queries • 2 - All queries, user, and system • 3 - System generated internal queries |
| Current_User_Profile | QVC102 | Current user profile name |
| Expression_Evaluator_Used | QVC1E | Expression Evaluator Used (Y/N) |
| Host_Server_Delta | QVP15A | Time not spent within Host Server |
| NTS_Lock_Space_Id | QQC301 | NTS Lock Space Identifier |
| IP_Address | QQC183 | IP Address |
| IP_Type | QFC11 | IP address type <ul style="list-style-type: none"> • '0' = No client IP address • '1' = IPV4 format • '2' = IPV6 format <p>Only applicable for database server jobs.</p> |
| IP_Port_Number | QQSMINT2 | IP Port Number |
| NTS_Transaction_Id | QVC3004 | NTS Transaction Identifier |
| NTS_Format_Id_Length | QQSMINT3 | NTS Format Identified length |
| NTS_Transaction_ID_SubLength | QQSMINT4 | NTS Transaction Identifier sublength. |
| Unique_Refresh_Counter2 | QVRCNT | Unique refresh counter |
| Times_Run | QVP15F | Number of times this Statement was run. If Null, then the statement was run once. <p>Note: While using an SQL Plan Cache snapshot, this value can be set by the database monitor. This value might be null if the query never completed, or was running when the snapshot was created. If there is not a plan cache snapshot, the value is null.</p> |
| Full_Opens | QVP15E | Number of runs that were processed as full opens. If Null, then the refresh count (qvrnt) is used to determine if the open was a full open (0) or a pseudo open (>0). <p>Note: While using an SQL Plan Cache snapshot, this value can be set by the database monitor. This value might be null if the query never completed, or was running when the snapshot was created. If there is not a plan cache snapshot, the value is null.</p> |
| Proc_In_Cache | QVC1F | Procedure definition was found in an internal cache. (Y/N) Only applicable for CALL statements. |
| Combined_Operation | QWC1A | Statement was performed with the processing for another statement. (Y/N) Only applicable for OPEN, FETCH, and CLOSE statements. |
| Client_Applname | QVC3001 | Client Special Register - application name |
| Client_Userid | QVC3002 | Client Special Register - userid |
| Client_Wrkstnname | QVC3003 | Client Special Register - work station name |
| Client_Acctng | QVC3005 | Client Special Register - accounting string |
| Client_Programid | QVC3006 | Client Special Register - program name |
| Interface_Information | QVC5001 | Part of the CLIENT special register information. Three types of info are stored in this char500 column, separated by colons. <ul style="list-style-type: none"> • First part, Interface Name, varchar(127); • Second part, Interface Level, varchar(63); • Third part, Interface Type, varchar(63) |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------|-------------------|--|
| Open_Options | QVC82 | Open options appear as a combination of the following characters, representing the actual capability for the cursor. The character values are left-aligned and padded on the right with blanks. Example 'RU ' indicate that the cursor is both read and update capable. <ul style="list-style-type: none"> • R - Read capable • W - Write capable • U - Update capable • D - Delete capable |
| Extended_Indicators | QWC1D | An Update or Insert statement was enabled to use extended indicators (Y/N). |
| DECFLOAT_Rounding_Mode | QWC1C | Rounding mode to use for DECFLOAT computations and conversions. <ul style="list-style-type: none"> • 'E' = ROUND_HALF_EVEN • 'C' = ROUND_CEILING • 'D' = ROUND_DOWN • 'F' = ROUND_FLOOR • 'G' = ROUND_HALF_DOWN • 'H' = ROUND_HALF_UP • 'U' = ROUND_UP |
| SQL_DECFLOAT_Warnings | QWC1E | DECFLOAT computations and conversions involving division by 0, overflow, underflow, an invalid operand, an inexact result, or a subnormal number results in a warning (Y/N). |
| Worst_Time_Micro | QVP15D | If not null, this time is the time for the slowest single run of this query. Note: When monitor files are created when using an SQL Plan Cache snapshot, this time represents the run time for the longest single run of the query. If the value is null, then the longest run information is not available. In that case, QQI6 might be the next best answer. See documentation for QQI6 for the proper use of that field |
| SQ_Unique_Count | QQINT05 | A unique count used to uniquely identify statements which do not have an ODP but do pass in host variables. If QQUCNT is 0 and the statement passes in host variables, this value is non-zero. An example would be a CALL statement. |
| Concurrent_Access_Res_Used | QFC13 | Specifies what method of concurrent access resolution was used. <ul style="list-style-type: none"> • 'N' = Concurrent access resolution is not applicable. This method applies to read queries with no commit or uncommitted read. • 'S' = SKIP LOCKED DATA clause was specified and rows with incompatible locks held by other transactions are skipped. • 'U' = USE CURRENTLY COMMITTED clause was specified and the currently committed version of data being updated or deleted is used. Data being inserted is skipped. • 'W' = Wait for commit or rollback when data is in the process of being inserted, updated, or deleted. This is the default method when the isolation level does not apply, the query is processed by CQE, or when not specified by the user. |
| SQL_Scalar_UDFs_Not_Inlined | QQSMINT8 | Specifies the number of SQL scalar user-defined functions (UDFs) that were not inlined in an SQL query or expression. |
| Result_Set_Cursor | QVC3007 | Result Set Cursor name. Set by Allocate Cursor, Fetch, and Close. |
| Implicit_XMLPARSE_Option | QFC12 | CURRENT IMPLICIT XMLPARSE OPTION special register. This option is used to specify white-space handling for an implicit parse of serialized XML data. <ul style="list-style-type: none"> • 'S' = STRIP WHITESPACE • 'P' = PRESERVE WHITESPACE |

Table 59. QQQ1000 - SQL Information (continued)

| View Column Name | Table Column Name | Description |
|---------------------|-------------------|--|
| SQL_XML_Data_CCSID | QQSMINT7 | The CCSID used for XML columns, host variables, parameter markers, and expressions if not explicitly specified. |
| OPTIMIZER_USE | QQSMINT5 | Which optimizer was used for the 1000 null = monitor predates this option <ul style="list-style-type: none"> • 0 = Does not apply for this statement • 1 = SQE was used (SQL Query Engine) • 2 = CQE was used (Classic Query Engine) • 3 = CQE direct was used (statements like INSERT W/VALUES) |
| XML_Schema_In_Cache | QFC14 | The XML schema binary used during XMLVALIDATE or decomposition was found in the XML cache. <ul style="list-style-type: none"> • 'Y' = Yes • 'N' = No |

Database monitor view 3000 - Table Scan

Displays the SQL logical view format for database monitor QQQ3000

Create View QQQ3000 as

```
(SELECT QQRID as Row_ID,
        QQTIME as Time_Created,
        QQJFLD as Join_Column,
        QQRDBN as Relational_Database_Name,
        QSYS as System_Name,
        QQJOB as Job_Name,
        QQUSER as Job_User,
        QQJNUM as Job_Number,
        QQI9 as Thread_ID,
        QQUCNT as Unique_Count,
        QQUDEF as User_Defined,
        QQQDTN as Unique_SubSelect_Number,
        QQQDTL as SubSelect_Nested_Level,
        QQMATN as Materialized_View_Subselect_Number,
        QQMATL as Materialized_View_Nested_Level,
        QVP15E as Materialized_View_Union_Level,
        QVP15A as Decomposed_Subselect_Number,
        QVP15B as Total_Number_Decomposed_SubSelects,
        QVP15C as Decomposed_SubSelect_Reason_Code,
        QVP15D as Starting_Decomposed_SubSelect,
        QQTLN as System_Table_Schema,
        QQTFN as System_Table_Name,
        QQTMN as Member_Name,
        QQPTLN as System_Base_Table_Schema,
        QQPTFN as System_Base_Table_Name,
        QQPTMN as Base_Member_Name,
        QQTOTR as Table_Total_Rows,
        QQREST as Estimated_Rows_Selected,
        QQAJN as Estimated_Join_Rows,
        QQEPT as Estimated_Processing_Time,
        QQJNP as Join_Position,
        QQI1 as DataSpace_Number,
        QQC21 as Join_Method,
        QQC22 as Join_Type,
        QQC23 as Join_Operator,
        QQI2 as Index_Advised_Columns_Count,
        QQDSS as DataSpace_Selection,
        QQIDXA as Index_Advised,
        QQRCOD as Reason_Code,
        QQIDXD as Index_Advised_Columns,
        QVQTBL as Table_Name,
        QVQLIB as Table_Schema,
```

```

|         QVPTBL as Base_Table_Name,
|         QVPLIB as Base_Table_Schema,
|         QVBNDY as Bound,
|         QVRCNT as Unique_Refresh_Counter,
|         QVJFANO as Join_Fanout,
|         QVFILES as Join_Table_Count,
|         QVPARPF as Parallel_Prefetch,
|         QVPARPL as Parallel_PreLoad,
|         QVPARD as Parallel_Degree_Requested,
|         QVPARU as Parallel_Degree_Used,
|         QVPARRC as Parallel_Degree_Reason_Code,
|         QVCTIM as Estimated_Cumulative_Time,
|         QQC11 as Skip_Sequential_Table_Scan,
|         QQI3 as Table_Size,
|         QVC3001 as DataSpace_Selection_Columns,
|         QQC14 as Derived_Column_Selection,
|         QVC3002 as Derived_Column_Selection_Columns,
|         QQC18 as Read_Trigger,
|         QVP157 as UDTF_Cardinality,
|         QVC1281 as UDTF_Specific_Name,
|         QVC1282 as UDTF_Specific_Schema,
|         QVP154 as Pool_Size,
|         QVP155 as Pool_Id,
|         QQC13 as MQT_Replacement,
|         QQC15 as InsertTable,
|         QQSMINTF as Plan_Iteration_Number
| FROM UserLib/DBMONTABLE
| WHERE QQRID=3000)

```

Table 60. QQQ3000 - Table Scan

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |

Table 60. QQQ3000 - Table Scan (continued)

| View Column Name | Table Column Name | Description |
|-------------------------------|-------------------|---|
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| System_Table_Schema | QQTLN | Schema of table queried |
| System_Table_Name | QQTFN | Name of table queried |
| Member_Name | QQTMN | Member name of table queried |
| System_Base_Table_Schema | QQPTLN | Schema name of base table |
| System_Base_Table_Name | QQPTFN | Name of base table for table queried |
| Base_Member_Name | QQPTMN | Member name of base table |
| Table_Total_Rows | QQTOTR | Total rows in table |
| Estimated_Rows_Selected | QQREST | Estimated number of rows selected |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI1 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product |
| Index_Advised_Columns_Count | QQI2 | Number of advised columns that use index scan-key positioning |
| DataSpace_Selection | QQDSS | Dataspace selection <ul style="list-style-type: none"> • Y - Yes • N - No |
| Index_Advised | QQIDXA | Index advised <ul style="list-style-type: none"> • Y - Yes • N - No |

Table 60. QQQ3000 - Table Scan (continued)

| View Column Name | Table Column Name | Description |
|----------------------------------|-------------------|---|
| Reason_Code | QQRCOD | Reason code <ul style="list-style-type: none"> • T1 - No indexes exist. • T2 - Indexes exist, but none can be used. • T3 - Optimizer chose table scan over available indexes. |
| Index_Advised_Columns | QQIDXD | Columns for the index advised |
| Table_Name | QVQTBL | Queried table, long name |
| Table_Schema | QVQLIB | Schema of queried table, long name |
| Base_Table_Name | QVPTBL | Base table, long name |
| Base_Table_Schema | QVPLIB | Schema of base table, long name |
| Bound | QVBNDY | I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> • I - I/O bound • C - CPU bound |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (Y/N) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested |
| Parallel_Degree_Used | QVPARU | Parallel degree used |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Skip_Sequential_Table_Scan | QQC11 | Skip sequential table scan (Y/N) |
| Table_Size | QQI3 | Size of table being queried |
| DataSpace_Selection_Columns | QVC3001 | Columns used for dataspace selection |
| Derived_Column_Selection | QQC14 | Derived column selection (Y/N) |
| Derived_Column_Selection_Columns | QVC3002 | Columns used for derived column selection |
| Read_Trigger | QQC18 | Read Trigger (Y/N) |
| UDTF_Cardinality | QVP157 | User-defined table function Cardinality |
| UDTF_Specific_Name | QVC1281 | User-defined table function specific name |
| UDTF_Specific_Schema | QVC1282 | User-defined table function specific schema |
| Pool_Size | QVP154 | Memory pool size |
| Pool_Id | QVP155 | Memory pool ID |
| MQT_Replacement | QQC13 | Materialized Query Table replaced queried table (Y/N) |
| Insert_Table | QQC15 | This is a target table of an insert (Y/N) |

Table 60. QQQ3000 - Table Scan (continued)

| View Column Name | Table Column Name | Description |
|-----------------------|-------------------|--|
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3001 - Index Used

Displays the SQL logical view format for database monitor QQQ3001

```

Create View QQQ3001 as
(SELECT QQRID as Row_ID,
      QQTIME as Time_Created,
      QQJFLD as Join_Column,
      QQRDBN as Relational_Database_Name,
      QQSYS as System_Name,
      QQJOB as Job_Name,
      QQUSER as Job_User,
      QQJNUM as Job_Number,
      QQI9 as Thread_ID,
      QQUCNT as Unique_Count,
      QQUDEF as User_Defined,
      QQQDTN as Unique_SubSelect_Number,
      QQQDTL as SubSelect_Nested_Level,
      QQMATN as Materialized_View_Subselect_Number,
      QQMATL as Materialized_View_Nested_Level,
      QVP15E as Materialized_View_Union_Level,
      QVP15A as Decomposed_Subselect_Number,
      QVP15B as Total_Number_Decomposed_SubSelects,
      QVP15C as Decomposed_SubSelect_Reason_Code,
      QVP15D as Starting_Decomposed_SubSelect,
      QQTLN as System_Table_Schema,
      QQTFN as System_Table_Name,
      QQTMN as Member_Name,
      QQPTLN as System_Base_Table_Schema,
      QQPTFN as System_Base_Table_Name,
      QQPTMN as Base_Member_Name,
      QQILNM as System_Index_Schema,
      QQIFNM as System_Index_Name,
      QQIMNM as Index_Member_Name,
      QQTOTR as Table_Total_Rows,
      QQREST as Estimated_Rows_Selected,
      QQFKEY as Index_Probe_Keys,
      QQKSEL as Index_Scan_Keys,
      QQAJN as Estimated_Join_Rows,
      QQEPT as Estimated_Processing_Time,
      QQJNP as Join_Position,
      QQI1 as DataSpace_Number,
      QQC21 as Join_Method,
      QQC22 as Join_Type,
      QQC23 as Join_Operator,
      QQI2 as Index_Advised_Probe_Count,
      QQKP as Index_Probe_Used,
      QQI3 as Index_Probe_Column_Count,
      QQKS as Index_Scan_Used,
      QQDSS as DataSpace_Selection,
      QQIDXA as Index_Advised,
      QQRCOD as Reason_Code,
      QQIDXD as Index_Advised_Columns,
      QQC11 as Constraint,
      QQ1000 as Constraint_Name,
      QVQTBL as Table_Name,
      QVQLIB as Table_Schema,
      QVPTBL as Base_Table_Name,
      QVPLIB as Base_Table_Schema,
      QVINAM as Index_Name,

```

```

|         QVILIB as Index_Schema,
|         QVBNDY as Bound,
|         QVRCNT as Unique_Refresh_Counter,
|         QVJFANO as Join_Fanout,
|         QVFILES as Join_Table_Count,
|         QVPARPF as Parallel_Prefetch,
|         QVPARPL as Parallel_Preload,
|         QVPARD as Parallel_Degree_Requested,
|         QVPARU as Parallel_Degree_Used,
|         QVPARRC as Parallel_Degree_Reason_Code,
|         QVCTIM as Estimated_Cumulative_Time,
|         QQc14 as Index_Only_Access,
|         QQc12 as Index_Fits_In_Memory,
|         QQC15 as Index_Type,
|         QVC12 as Index_Usage,
|         QQI4 as Index_Entries,
|         QQI5 as Unique_Keys,
|         QQI6 as Percent_Overflow,
|         QQI7 as Vector_Size,
|         QQI8 as Index_Size,
|         QQIA as Index_Page_Size,
|         QVP154 as Pool_Size,
|         QVP155 as Pool_Id,
|         QVP156 as Table_Size,
|         QQC16 as Skip_Sequential_Table_Scan,
|         QVC13 as Tertiary_Indexes_Exist,
|         QVC3001 as DataSpace_Selection_Columns,
|         QQC14 as Derived_Column_Selection,
|         QVC3002 as Derived_Column_Selection_Columns,
|         QVC3003 as Table_Columns_For_Index_Probe,
|         QVC3004 as Table_Columns_For_Index_Scan,
|         QVC3005 as Join_Selection_Columns,
|         QVC3006 as Ordering_Columns,
|         QVC3007 as Grouping_Columns,
|         QQC18 as Read_Trigger,
|         QVP157 as UDTF_Cardinality,
|         QVC1281 as UDTF_Specific_Name,
|         QVC1282 as UDTF_Specific_Schema,
|         QQC13 as MQT_Replacement,
|         QQSMINTF as Plan_Iteration_Number,
|         QVC3008 as Include_Values,
|         QVC15 as Sparse_Index
|
| FROM UserLib/DBMONTTable
| WHERE QQRID=3001)

```

Table 61. QQQ3001 - Index Used

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|---------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User-defined column |

Table 61. QQQ3001 - Index Used (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| System_Table_Schema | QQTLN | Schema of table queried |
| System_Table_Name | QQTFN | Name of table queried |
| Member_Name | QQTMN | Member name of table queried |
| System_Base_Table_Schema | QQPTLN | Schema name of base table |
| System_Base_Table_Name | QQPTFN | Name of base table for table queried |
| Base_Member_Name | QQPTMN | Member name of base table |
| System_Index_Schema | QQILNM | Schema name of index used for access |
| System_Index_Name | QQIFNM | Name of index used for access |
| Index_Member_Name | QQIMNM | Member name of index used for access |
| Table_Total_Rows | QQTOTR | Total rows in base table |
| Estimated_Rows_Selected | QQREST | Estimated number of rows selected |
| Index_Probe_Keys | QQFKEY | Columns selected through index scan-key positioning |
| Index_Scan_Keys | QQKSEL | Columns selected through index scan-key selection |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI1 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |

Table 61. QQQ3001 - Index Used (continued)

| View Column Name | Table Column Name | Description |
|---------------------------|-------------------|--|
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE- Less than or equal • CP - Cartesian product |
| Index_Advised_Probe_Count | QQI2 | Number of advised key columns that use index scan-key positioning |
| Index_Probe_Used | QQKP | Index scan-key positioning <ul style="list-style-type: none"> • Y - Yes • N - No |
| Index_Probe_Column_Count | QQI3 | Number of columns that use index scan-key positioning for the index used |
| Index_Scan_Used | QQKS | Index scan-key selection <ul style="list-style-type: none"> • Y - Yes • N - No |
| DataSpace_Selection | QQDSS | Dataspace selection <ul style="list-style-type: none"> • Y - Yes • N - No |
| Index_Advised | QQIDXA | Index advised <ul style="list-style-type: none"> • Y - Yes • N - No |
| Reason_Code | QQRCD | Reason code <ul style="list-style-type: none"> • I1 - Row selection • I2 - Ordering/Grouping • I3 - Row selection and Ordering/Grouping • I4 - Nested loop join • I5 - Row selection using bitmap processing |
| Index_Advised_Columns | QQIDXD | Columns for index advised |
| Constraint | QQC11 | Index is a constraint (Y/N) |
| Constraint_Name | QQ1000 | Constraint name |
| Table_Name | QVQTBL | Queried table, long name |
| Table_Schema | QVQLIB | Schema of queried table, long name |
| Base_Table_Name | QVPTBL | Base table, long name |
| Base_Table_Schema | QVPLIB | Schema of base table, long name |
| Index_Name | QVINAM | Name of index (or constraint) used, long name |
| Index_Schema | QVILIB | Library of index used, long name |
| Bound | QVBNDY | I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> • I - I/O bound • C - CPU bound |

Table 61. QQQ3001 - Index Used (continued)

| View Column Name | Table Column Name | Description |
|----------------------------------|-------------------|--|
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Join_Fanout | QVJFANO | Join fanout. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_Preload | QVPARPL | Parallel Preload (Y/N) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested |
| Parallel_Degree_Used | QVPARU | Parallel degree used |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Index_Only_Access | QVC14 | Index only access (Y/N) |
| Index_Fits_In_Memory | QQC12 | Index fits in memory (Y/N) |
| Index_Type | QQC15 | Type of Index. Possible values are: <ul style="list-style-type: none"> • B - Binary Radix Index • C - Constraint (Binary Radix) • E - Encoded Vector Index (EVI) • X - Query created temporary index |
| Index_Usage | QVC12 | Index Usage. Possible values are: <ul style="list-style-type: none"> • P - Primary Index • T - Tertiary (AND or OR) Index |
| Index_Entries | QQI4 | Number of index entries |
| Unique_Keys | QQI5 | Number of unique key values |
| Percent_Overflow | QQI6 | Percent overflow |
| Vector_Size | QQI7 | Vector size |
| Index_Size | QQI8 | Index size |
| Index_Page_Size | QQIA | Index page size |
| Pool_Size | QVP154 | Pool size |
| Pool_Id | QVP155 | Pool ID |
| Table_Size | QVP156 | Table size |
| Skip_Sequential_Table_Scan | QQC16 | Skip sequential table scan (Y/N) |
| Tertiary_Indexes_Exist | QVC13 | Tertiary indexes exist (Y/N) |
| DataSpace_Selection_Columns | QVC3001 | Columns used for dataspace selection |
| Derived_Column_Selection | QQC14 | Derived column selection (Y/N) |
| Derived_Column_Selection_Columns | QVC3002 | Columns used for derived column selection |
| Table_Column_For_Index_Probe | QVC3003 | Columns used for index scan-key positioning |
| Table_Column_For_Index_Scan | QVC3004 | Columns used for index scan-key selection |

Table 61. QQQ3001 - Index Used (continued)

| View Column Name | Table Column Name | Description |
|------------------------|-------------------|---|
| Join_Selection_Columns | QVC3005 | Columns used for Join selection |
| Ordering_Columns | QVC3006 | Columns used for Ordering |
| Grouping_Columns | QVC3007 | Columns used for Grouping |
| Read_Trigger | QQC18 | Read Trigger (Y/N) |
| UDTF_Cardinality | QVP157 | Cardinality for user-defined table function. |
| UDTF_Specific_Name | QVC1281 | Specific name for user-defined table function. |
| UDTF_Specific_Schema | QVC1282 | Specific schema for user-defined table function. |
| MQT_Replacement | QQC13 | Materialized Query Table replaced queried table (Y/N) |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |
| Include_Values | QVC3008 | Encoded Vector indexes only. |
| | | Aggregates included as part of index creation and predetermined for Grouping query request. |
| Sparse_Index | QVC15 | Index contains sparse selection or Select/Omit selection criteria (Y/N). |

Database monitor view 3002 - Index Created

Displays the SQL logical view format for database monitor QQQ3002.

```

Create View QQQ3002 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,
         QQUDEF as User_Defined,
         QQQDTN as Unique_SubSelect_Number,
         QQQDTL as SubSelect_Nested_Level,
         QQMATN as Materialized_View_Subselect_Number,
         QQMATL as Materialized_View_Nested_Level,
         QVP15E as Materialized_View_Union_Level,
         QVP15A as Decomposed_Subselect_Number,
         QVP15B as Total_Number_Decomposed_SubSelects,
         QVP15C as Decomposed_SubSelect_Reason_Code,
         QVP15D as Starting_Decomposed_SubSelect,
         QQTLN as System_Table_Schema,
         QQTFN as System_Table_Name,
         QQTMN as Member_Name,
         QQPTLN as System_Base_Table_Schema,
         QQPTFN as System_Base_Table_Name,
         QQPTMN as Base_Member_Name,
         QQILNM as System_Index_Schema,
         QQIFNM as System_Index_Name,
         QQIMNM as Index_Member_Name,
         QQNTNM as NLSS_Table,
         QQNLNM as NLSS_Library,
         QQSTIM as Start_Timestamp,
         QQETIM as End_Timestamp,
         QQTOTR as Table_Total_Rows,
         QQRIDX as Created_Index_Entries,

```

QQREST as Estimated_Rows_Selected,
 QQFKEY as Index_Probe_Keys,
 QQKSEL as Index_Scan_Keys,
 QQAJN as Estimated_Join_Rows,
 QQEPT as Estimated_Processing_Time,
 QQJNP as Join_Position,
 QQI1 as DataSpace_Number,
 QQC21 as Join_Method,
 QQC22 as Join_Type,
 QQC23 as Join_Operator,
 QQI2 as Index_Advised_Probe_Count,
 QQKP as Index_Probe_Used,
 QQI3 as Index_Probe_Column_Count,
 QQKS as Index_Scan_Used,
 QQDSS as DataSpace_Selection,
 QQIDXA as Index_Advised,
 QQRCD as Reason_Code,
 QQIDXD as Index_Advised_Columns,
 QQ1000 as Created_Index_Columns,
 QVQTBL as Table_Name,
 QVQLIB as Table_Schema,
 QVPTBL as Base_Table_Name,
 QVPLIB as Base_Table_Schema,
 QVINAM as Index_Name,
 QVILIB as Index_Schema,
 QVBNDY as Bound,
 QVRCNT as Unique_Refresh_Counter,
 QVJFANO as Join_Fanout,
 QVFILES as Join_Table_Count,
 QVPARPF as Parallel_Prefetch,
 QVPARPL as Parallel_Preload,
 QVPARD as Parallel_Degree_Requested,
 QVPARU as Parallel_Degree_Used,
 QVPARRC as Parallel_Degree_Reason_Code,
 QVCTIM as Estimated_Cumulative_Time,
 QQC101 as Created_Index_Name,
 QQC102 as Created_Index_Schema,
 QQI4 as Created_Index_Page_Size,
 QQI5 as Created_Index_Row_Size,
 QQC14 as Created_Index_Used_ACS_Table,
 QQC103 as Created_Index_ACS_Table,
 QQC104 as Created_Index_ACS_Library,
 QVC13 as Created_Index_Reusable,
 QVC14 as Created_Index_Sparse,
 QVC1F as Created_Index_Type,
 QVP15F as Created_Index_Unique_EVI_Count,
 QVC15 as Permanent_Index_Created,
 QVC16 as Index_From_Index,
 QVP151 as Created_Index_Parallel_Degree_Requested,
 QVP152 as Created_Index_Parallel_Degree_Used,
 QVP153 as Created_Index_Parallel_Degree_Reason_Code,
 QVC17 as Index_Only_Access,
 QVC18 as Index_Fits_In_Memory,
 QVC1B as Index_Type,
 QQI6 as Index_Entries,
 QQI7 as Unique_Keys,
 QVP158 as Percent_Overflow,
 QVP159 as Vector_Size,
 QQI8 as Index_Size,
 QVP156 as Index_Page_Size,
 QVP154 as Pool_Size,
 QVP155 as Pool_ID,
 QVP157 as Table_Size,
 QVC1C as Skip_Sequential_Table_Scan,
 QVC3001 as DataSpace_Selection_Columns,
 QVC1E as Derived_Column_Selection,
 QVC3002 as Derived_Column_Selection_Columns,

```

|         QVC3003 as Table_Column_For_Index_Probe,
|         QVC3004 as Table_Column_For_Index_Scan,
|         QQC18 as Read_Trigger,
|         QQC13 as MQT_Replacement,
|         QQC16 as Reused_Temporary_Index,
|         QQINT03 as Estimated_Storage,
|         QQSMINTF as Plan_Iteration_Number
|   FROM   UserLib/DBMONTAbLe
|   WHERE  QQRID=3002)

```

Table 62. QQQ3002 - Index Created

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| System_Table_Schema | QQTLN | Schema of table queried |
| System_Table_Name | QQTFN | Name of table queried |
| Member_Name | QQTMN | Member name of table queried |
| System_Base_Table_Schema | QQPTLN | Schema name of base table |
| System_Base_Table_Name | QQPTFN | Name of base table for table queried |
| Base_Member_Name | QQPTMN | Member name of base table |
| System_Index_Schema | QQILNM | Schema name of index used for access |
| System_Index_Name | QQIFNM | Name of index used for access |
| Index_Member_Name | QQIMNM | Member name of index used for access |
| NLSS_Table | QQNTNM | NLSS table |
| NLSS_Library | QQNLNM | NLSS library |

Table 62. QQQ3002 - Index Created (continued)

| View Column Name | Table Column Name | Description |
|---------------------------|-------------------|---|
| Start_Timestamp | QQSTIM | Start timestamp, when available. |
| End_Timestamp | QQETIM | End timestamp, when available |
| Table_Total_Rows | QQTOTR | Total rows in table |
| Created_Index_Entries | QQRIDX | Number of entries in index created |
| Estimated_Rows_Selected | QQREST | Estimated number of rows selected |
| Index_Probe_Keys | QQFKEY | Keys selected thru index scan-key positioning |
| Index_Scan_Keys | QQKSEL | Keys selected thru index scan-key selection |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI1 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product |
| Index_Advised_Probe_Count | QQI2 | Number of advised key columns that use index scan-key positioning |
| Index_Probe_Used | QQKP | Index scan-key positioning <ul style="list-style-type: none"> • Y - Yes • N - No |
| Index_Probe_Column_Count | QQI3 | Number of columns that use index scan-key positioning for the index used |
| Index_Scan_Used | QQKS | Index scan-key selection <ul style="list-style-type: none"> • Y - Yes • N - No |
| DataSpace_Selection | QQDSS | Dataspace selection <ul style="list-style-type: none"> • Y - Yes • N - No |

Table 62. QQQ3002 - Index Created (continued)

| View Column Name | Table Column Name | Description |
|------------------------------|-------------------|---|
| Index_Advised | QQIDXA | Index advised <ul style="list-style-type: none"> • Y - Yes • N - No |
| Reason_Code | QQRCOD | Reason code <ul style="list-style-type: none"> • I1 - Row selection • I2 - Ordering/Grouping • I3 - Row selection and Ordering/Grouping • I4 - Nested loop join |
| Index_Advised_Columns | QQIDXD | Key columns for index advised |
| Created_Index_Columns | QQ1000 | Key columns for index created |
| Table_Name | QVQTBL | Queried table, long name |
| Table_Schema | QVQLIB | Schema of queried table, long name |
| Base_Table_Name | QVPTBL | Base table, long name |
| Base_Table_Schema | QVPLIB | Schema of base table, long name |
| Index_Name | QVINAM | Name of index (or constraint) used, long name |
| Index_Schema | QVILIB | Schema of index used, long name |
| Bound | QVBNDY | I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> • I - I/O bound • C - CPU bound |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_Preload | QVPARPL | Parallel Preload (index used) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested (index used) |
| Parallel_Degree_Used | QVPARU | Parallel degree used (index used) |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited (index used) |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Created_Index_Name | QQC101 | Name of index created - when available |
| Created_Index_Schema | QQC102 | Schema of index created - when available |
| Created_Index_Page_Size | QQI4 | Page size of index created |
| Created_Index_Row_Size | QQI5 | Row size of index created |
| Created_Index_Used_ACS_Table | QQC14 | Index Created used Alternate Collating Sequence Table (Y/N) |
| Created_Index_ACS_Table | QQC103 | Alternate Collating Sequence table of index created. |

Table 62. QQQ3002 - Index Created (continued)

| View Column Name | Table Column Name | Description |
|---|-------------------|--|
| Created_Index_ACS_Library | QQC104 | Alternate Collating Sequence library of index created. |
| Created_Index_Reusable | QVC13 | Index created is reusable (Y/N) |
| Created_Index_Sparse | QVC14 | Index created is sparse index (Y/N) |
| Created_Index_Type | QVC1F | Type of index created. Possible values: <ul style="list-style-type: none"> • B - Binary Radix Index • E - Encoded Vector Index (EVI) |
| Created_Index_Unique_EVI_Count | QVP15F | Number of unique values of index created if index created is an EVI index. |
| Permanent_Index_Created | QVC15 | Permanent index created (Y/N) |
| Index_From_Index | QVC16 | Index from index (Y/N) |
| Created_Index_Parallel_Degree_Requested | QVP151 | Parallel degree requested (index created) |
| Created_Index_Parallel_Degree_Used | QVP152 | Parallel degree used (index created) |
| Created_Index_Parallel_Degree_Reason_Code | QVP153 | Reason parallel processing was limited (index created) |
| Index_Only_Access | QVC17 | Index only access (Y/N) |
| Index_Fits_In_Memory | QVC18 | Index fits in memory (Y/N) |
| Index_Type | QVC1B | Type of Index. Possible values are: <ul style="list-style-type: none"> • B - Binary Radix Index • C - Constraint (Binary Radix) • E - Encoded Vector Index (EVI) • T - Tertiary (AND/OR) Index |
| Index_Entries | QQI6 | Number of index entries, index used |
| Unique_Keys | QQI7 | Number of unique key values, index used |
| Percent_Overflow | QVP158 | Percent overflow, index used |
| Vector_Size | QVP159 | Vector size, index used |
| Index_Size | QQI8 | Size of index used. |
| Index_Page_Size | QVP156 | Index page size |
| Pool_Size | QVP154 | Pool size |
| Pool_ID | QVP155 | Pool id |
| Table_Size | QVP157 | Table size |
| Skip_Sequential_Table_Scan | QVC1C | Skip sequential table scan (Y/N) |
| DataSpace_Selection_Columns | QVC3001 | Columns used for dataspace selection |
| Derived_Column_Selection | QVC1E | Derived column selection (Y/N) |
| Derived_Column_Selection_Columns | QVC3002 | Columns used for derived column selection |
| Table_Columns_For_Index_Probe | QVC3003 | Columns used for index scan-key positioning |
| Table_Columns_For_Index_Scan | QVC3004 | Columns used for index scan-key selection |
| Read_Trigger | QQC18 | Read Trigger (Y/N) |
| MQT_Replacement | QQC13 | Materialized Query Table replaced queried table (Y/N) |
| Reused_Temporary_Index | QQC16 | Temporary index reused (Y/N) |
| Estimated_Storage | QQINT03 | Estimated amount of temporary storage used, in megabytes, to create the temporary index. |

Table 62. QQQ3002 - Index Created (continued)

| View Column Name | Table Column Name | Description |
|-----------------------|-------------------|--|
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3003 - Query Sort

Displays the SQL logical view format for database monitor QQQ3003.

```

Create View QQQ3003 as
(SELECT QQRID as Row_ID,
      QQTIME as Time_Created,
      QQJFLD as Join_Column,
      QQRDBN as Relational_Database_Name,
      QQSYS as System_Name,
      QQJOB as Job_Name,
      QQUSER as Job_User,
      QQJNUM as Job_Number,
      QQI9 as Thread_ID,
      QQUCNT as Unique_Count,
      QQUDEF as User_Defined,
      QQQDTN as Unique_SubSelect_Number,
      QQQDTL as SubSelect_Nested_Level,
      QQMATN as Materialized_View_Subselect_Number,
      QQMATL as Materialized_View_Nested_Level,
      QVP15E as Materialized_View_Union_Level,
      QVP15A as Decomposed_Subselect_Number,
      QVP15B as Total_Number_Decomposed_SubSelects,
      QVP15C as Decomposed_SubSelect_Reason_Code,
      QVP15D as Starting_Decomposed_SubSelect,
      QQSTIM as Start_Timestamp,
      QQETIM as End_Timestamp,
      QQRSS as Sorted_Rows,
      QQI1 as Sort_Space_Size,
      QQI2 as Pool_Size,
      QQI3 as Pool_Id,
      QQI4 as Internal_Sort_Buffer_Length,
      QQI5 as External_Sort_Buffer_Length,
      QQRCOD as Reason_Code,
      QQI7 as Union_Reason_Subcode,
      QVBNDY as Bound,
      QVRCNT as Unique_Refresh_Counter,
      QVPPARPF as Parallel_Prefetch,
      QVPPARPL as Parallel_PreLoad,
      QVPPARD as Parallel_Degree_Requested,
      QVPPARU as Parallel_Degree_Used,
      QVPPARRC as Parallel_Degree_Reason_Code,
      QQEPT as Estimated_Processing_Time,
      QVCTIM as Estimated_Cumulative_Time,
      QQAJN as Estimated_Join_Rows,
      QQJNP as Join_Position,
      QQI6 as DataSpace_Number,
      QQC21 as Join_Method,
      QQC22 as Join_Type,
      QQC23 as Join_Operator,
      QVJFANO as Join_Fanout,
      QVFILES as Join_Table_Count,
      QQINT03 as Estimated_Storage,
      QQSMINTF as Plan_Iteration_Number
FROM   UserLib/DBMONTabTe
WHERE  QQRID=3003)

```

| *Table 63. QQQ3003 - Query Sort*

| View Column Name | Table Column Name | Description |
|------------------------------------|--------------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Start_Timestamp | QQSTIM | Start timestamp, when available |
| End_Timestamp | QQETIM | End timestamp, when available |
| Sorted_Rows | QQRSS | Estimated number of rows selected or sorted. |
| Sort_Space_Size | QQI1 | Estimated size of sort space. |
| Pool_Size | QQI2 | Pool size |
| Pool_Id | QQI3 | Pool id |
| Internal_Sort_Buffer_Length | QQI4 | Internal sort buffer length |
| External_Sort_Buffer_Length | QQI5 | External sort buffer length |

Table 63. QQQ3003 - Query Sort (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------|-------------------|---|
| Reason_Code | QQRCOD | Reason code <ul style="list-style-type: none"> F1 - Query contains grouping columns (GROUP BY) from more than one table, or contains grouping columns from a secondary table of a join query that cannot be reordered. F2 - Query contains ordering columns (ORDER BY) from more than one table, or contains ordering columns from a secondary table of a join query that cannot be reordered. F3 - The grouping and ordering columns are not compatible. F4 - DISTINCT was specified for the query. F5 - UNION was specified for the query. F6 - Query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 key columns specified for ordering. |
| Reason_Code (continued) | | <ul style="list-style-type: none"> 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. |
| Union_Reason_Subcode | QQI7 | Reason subcode for Union: <ul style="list-style-type: none"> 51 - Query contains UNION and ORDER BY 52 - Query contains UNION ALL |
| Bound | QVBNDY | I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> I - I/O bound C - CPU bound |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (index used) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested (index used) |
| Parallel_Degree_Used | QVPARU | Parallel degree used (index used) |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited (index used) |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI6 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> NL - Nested loop MF - Nested loop with selection HJ - Hash join |

Table 63. QQQ3003 - Query Sort (continued)

| View Column Name | Table Column Name | Description |
|-----------------------|-------------------|---|
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Estimated_Storage | QQINT03 | Estimated amount of temporary storage used, in megabytes, to create the temporary index. |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3004 - Temp Table

Displays the SQL logical view format for database monitor QQQ3004.

```

Create View QQQ3004 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,
         QQUDEF as User_Defined,
         QQQDTN as Unique_SubSelect_Number,
         QQQDTL as SubSelect_Nested_Level,
         QQMATN as Materialized_View_Subselect_Number,
         QQMATL as Materialized_View_Nested_Level,
         QVP15E as Materialized_View_Union_Level,
         QVP15A as Decomposed_Subselect_Number,
         QVP15B as Total_Number_Decomposed_SubSelects,
         QVP15C as Decomposed_SubSelect_Reason_Code,
         QVP15D as Starting_Decomposed_SubSelect,
         QQTLN as System_Table_Schema,
         QQTFN as System_Table_Name,
         QQTMN as Member_Name,
         QQPTLN as System_Base_Table_Schema,
         QQPTFN as System_Base_Table_Name,
  )

```

```

|         QQPTMN as Base_Member_Name,
|         QQSTIM as Start_Timestamp,
|         QQETIM as End_Timestamp,
|         QQC11 as Has_Default_Values,
|         QQTMPR as Table_Rows,
|         QQRCOD as Reason_Code,
|         QVQTBL as Table_Name,
|         QVQLIB as Table_Schema,
|         QVPTBL as Base_Table_Name,
|         QVPLIB as Base_Table_Schema,
|         QQC101 as Temporary_Table_Name,
|         QQC102 as Temporary_Table_Schema,
|         QVBNDY as Bound,
|         QVRCNT as Unique_Refresh_Counter,
|         QVJFANO as Join_Fanout,
|         QVFILES as Join_Table_Count,
|         QVPARPF as Parallel_Prefetch,
|         QVPARPL as Parallel_Preload,
|         QVPARD as Parallel_Degree_Requested,
|         QVPARU as Parallel_Degree_Used,
|         QVPARRC as Parallel_Degree_Reason_Code,
|         QQEPT as Estimated_Processing_Time,
|         QVCTIM as Estimated_Cumulative_Time,
|         QQAJN as Estimated_Join_Rows,
|         QQJNP as Join_Position,
|         QQI6 as DataSpace_Number,
|         QQC21 as Join_Method,
|         QQC22 as Join_Type,
|         QQC23 as Join_Operator,
|         QQI2 as Temporary_Table_Row_Size,
|         QQI3 as Temporary_Table_Size,
|         QQC12 as Temporary_Query_Result,
|         QQC13 as Distributed_Temporary_Table,
|         QVC3001 as Distributed_Temporary_Data_Nodes,
|         QQI7 as Materialized_Subquery_QDT_Level,
|         QQI8 as Materialized_Union_QDT_Level,
|         QQC14 as View_Contains_Union,
|         QQINT03 as Estimated_Storage,
|         QQSMINTF as Plan_Iteration_Number
| FROM UserLib/DBMONTTable
| WHERE QQRID=3004)

```

Table 64. QQQ3004 - Temp Table

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|---------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |

| Table 64. QQQ3004 - Temp Table (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| System_Table_Schema | QQTLN | Schema of table queried |
| System_Table_Name | QQTFN | Name of table queried |
| Member_Name | QQTMN | Member name of table queried |
| System_Base_Table_Schema | QQPTLN | Schema name of base table |
| System_Base_Table_Name | QQPTFN | Name of base table for table queried |
| Base_Member_Name | QQPTMN | Member name of base table |
| Start_Timestamp | QQSTIM | Start timestamp, when available |
| End_Timestamp | QQETIM | End timestamp, when available |
| Has_Default_Values | QQC11 | Default values may be present in temporary <ul style="list-style-type: none"> • Y - Yes • N - No |
| Table_Rows | QQTMPR | Estimated number of rows in the temporary |

Table 64. QQQ3004 - Temp Table (continued)

| View Column Name | Table Column Name | Description |
|------------------|-------------------|---|
| Reason_Code | QQRCOD | <p>Reason code. Possible values are:</p> <ul style="list-style-type: none"> • F1 - Query contains grouping columns (GROUP BY) from more than one table, or contains grouping columns from a secondary table of a join query that cannot be reordered. • F2 - Query contains ordering columns (ORDER BY) from more than one table, or contains ordering columns from a secondary table of a join query that cannot be reordered. • F3 - The grouping and ordering columns are not compatible. • F4 - DISTINCT was specified for the query. • F5 - UNION was specified for the query. • F6 - Query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 key columns specified for ordering. • F7 - Query optimizer chose to use a sort rather than an index to order the results of the query. • F8 - Perform specified row selection to minimize I/O wait time. • F9 - The query optimizer chose to use a hashing algorithm rather than an index to perform the grouping. • FA - The query contains a join condition that requires a temporary table • FB - The query optimizer creates a run-time temporary file in order to implement certain correlated group by queries. • FC - The query contains grouping fields and there is a read trigger on at least one of the physical files in the query. • FD - The query optimizer creates a runtime temporary file for a static-cursor request. • H1 - Table is a join logical file and its join type does not match the join type specified in the query. • H2 - Format specified for the logical table references more than one base table. • H3 - Table is a complex SQL view requiring a temporary table to contain the 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. |

Table 64. QQQ3004 - Temp Table (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------|-------------------|---|
| Table_Name | QVQTBL | Queried table, long name |
| Table_Schema | QVQLIB | Schema of queried table, long name |
| Base_Table_Name | QVPTBL | Base table, long name |
| Base_Table_Schema | QVPLIB | Library of base table, long name |
| Temporary_Table_Name | QQC101 | Temporary table name |
| Temporary_Table_Schema | QQC102 | Temporary table schema |
| Bound | QVBNDY | I/O or CPU bound. Possible values are: <ul style="list-style-type: none"> • I - I/O bound • C - CPU bound |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (Y/N) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested |
| Parallel_Degree_Used | QVPARU | Parallel degree used |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI6 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |

Table 64. QQQ3004 - Temp Table (continued)

| View Column Name | Table Column Name | Description |
|----------------------------------|-------------------|---|
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product |
| Temporary_Table_Row_Size | QQI2 | Row size of temporary table, in bytes |
| Temporary_Table_Size | QQI3 | Estimated size of temporary table, in bytes. |
| Temporary_Query_Result | QQC12 | Temporary result table that contains the results of the query. (Y/N) |
| Distributed_Temporary_Table | QQC13 | Distributed Table (Y/N) |
| Distributed_Temporary_Data_Nodes | QVC3001 | Data nodes of temporary table |
| Materialized_Subquery_QDT_Level | QQI7 | Materialized subquery QDT level |
| Materialized_Union_QDT_Level | QQI8 | Materialized Union QDT level |
| View_Contains_Union | QQC14 | Union in a view (Y/N) |
| Estimated_Storage | QQINT03 | Estimated amount of temporary storage used, in megabytes, to create the temporary index. |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3005 - Table Locked

Displays the SQL logical view format for database monitor QQQ3005.

Create View QQQ3005 as

```
(SELECT QQRID as Row_ID,
        QQTIME as Time_Created,
        QQJFLD as Join_Column,
        QQRDBN as Relational_Database_Name,
        QQSYS as System_Name,
        QQJOB as Job_Name,
        QQUSER as Job_User,
        QQJNUM as Job_Number,
        QQI9 as Thread_ID,
        QQUCNT as Unique_Count,
        QQUDEF as User_Defined,
        QQQDTN as Unique_SubSelect_Number,
        QQQDTL as SubSelect_Nested_Level,
        QQMATN as Materialized_View_Subselect_Number,
        QQMATL as Materialized_View_Nested_Level,
        QVP15E as Materialized_View_Union_Level,
        QVP15A as Decomposed_Subselect_Number,
        QVP15B as Total_Number_Decomposed_SubSelects,
        QVP15C as Decomposed_SubSelect_Reason_Code,
        QVP15D as Starting_Decomposed_SubSelect,
        QQTLN as System_Table_Schema,
        QQTFN as System_Table_Name,
        QQTMN as Member_Name,
        QQPTLN as System_Base_Table_Schema,
        QQPTFN as System_Base_Table_Name,
        QQPTMN as Base_Member_Name,
        QQC11 as Lock_Success,
        QQC12 as Unlock_Request,
```

```

    QQRCOD as Reason_Code,
    QVQTBL as Table_Name,
    QVQLIB as Table_Schema,
    QVPTBL as Base_Table_Name,
    QVPLIB as Base_Table_Schema,
    QQJNP as Join_Position,
    QQI6 as DataSpace_Number,
    QQC21 as Join_Method,
    QQC22 as Join_Type,
    QQC23 as Join_Operator,
    QVJFANO as Join_Fanout,
    QVFILES as Join_Table_Count,
    QVRCNT as Unique_Refresh_Counter
FROM   UserLib/DBMONTTable
WHERE  QQRID=3005)

```

Table 65. QQQ3005 - Table Locked

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| System_Table_Schema | QQTLN | Schema of table queried |
| System_Table_Name | QQTFN | Name of table queried |
| Member_Name | QQTMN | Member name of table queried |
| System_Base_Table_Schema | QQPTLN | Schema name of base table |
| System_Base_Table_Name | QQPTFN | Name of base table for table queried |
| Base_Member_Name | QQPTMN | Member name of base table |
| Lock_Success | QQC11 | Successful lock indicator (Y/N) |

Table 65. QQQ3005 - Table Locked (continued)

| View Column Name | Table Column Name | Description |
|------------------------|-------------------|--|
| Unlock_Request | QQC12 | Unlock request (Y/N) |
| Reason_Code | QQRCOD | Reason code <ul style="list-style-type: none"> • L1 - UNION with *ALL or *CS with Keep Locks • L2 - DISTINCT with *ALL or *CS with Keep Locks • L3 - No duplicate keys with *ALL or *CS with Keep Locks • L4 - Temporary needed with *ALL or *CS with Keep Locks • L5 - System Table with *ALL or *CS with Keep Locks • L6 - Orderby > 2000 bytes with *ALL or *CS with Keep Locks • L9 - Unknown • LA - User-defined table function with *ALL or *CS with Keep Locks |
| Table_Name | QVQTBL | Queried table, long name |
| Table_Schema | QVQLIB | Schema of queried table, long name |
| Base_Table_Name | QVPTBL | Base table, long name |
| Base_Table_Schema | QVPLIB | Schema of base table, long name |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI6 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |

Database monitor view 3006 - Access Plan Rebuilt

Displays the SQL logical view format for database monitor QQQ3006.

```

| Create View QQQ3006 as
|   (SELECT QQRID as Row_ID,
|          QQTIME as Time_Created,
|          QQJFLD as Join_Column,
|          QQRDBN as Relational_Database_Name,
|          QQSYS as System_Name,
|          QQJOB as Job_Name,
|          QQUSER as Job_User,
|          QQJNUM as Job_Number,
|          QQI9 as Thread_ID,
|          QQUCNT as Unique_Count,
|          QQUDEF as User_Defined,
|          QQQDTN as Unique_SubSelect_Number,
|          QQQDTL as SubSelect_Nested_Level,
|          QQMATN as Materialized_View_Subselect_Number,
|          QQMATL as Materialized_View_Nested_Level,
|          QVP15E as Materialized_View_Union_Level,
|          QVP15A as Decomposed_Subselect_Number,
|          QVP15B as Total_Number_Decomposed_SubSelects,
|          QVP15C as Decomposed_SubSelect_Reason_Code,
|          QVP15D as Starting_Decomposed_SubSelect,
|          QQRCOD as Reason_Code,
|          QQC21 as SubCode,
|          QVRCNT as Unique_Refresh_Counter,
|          QQTIM1 as Last_Access_Plan_Rebuild_Timestamp,
|          QQC11 as Reoptimization_Done,
|          QVC22 as Previous_Reason_Code,
|          QVC23 as Previous_SubCode,
|          QQSMINTF as Plan_Iteration_Number
|   FROM   UserLib/DBMONTAb1e
|   WHERE  QQRID=3006)

```

Table 66. QQQ3006 - Access Plan Rebuilt

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |

Table 66. QQQ3006 - Access Plan Rebuilt (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|---|
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Reason_Code | QQRCOD | Reason code why access plan was rebuilt <ul style="list-style-type: none"> • A1 - A table or member is not the same object as the one referenced when the access plan was last built. Some reasons they might be different are: <ul style="list-style-type: none"> – Object was deleted and recreated. – Object was saved and restored. – Library list was changed. – Object was renamed. – Object was moved. – Object was overridden to a different object. – This is the first run of this query after the object containing the query has been restored. • A2 - Access plan was built to use a reusable Open Data Path (ODP) and the optimizer chose to use a non-reusable ODP for this call. • A3 - Access plan was built to use a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP for this call. • A4 - The number of rows in the table has changed by more than 10% since the access plan was last built. • A5 - A new index exists over one of the tables in the query • A6 - An index that was used for this access plan no longer exists or is no longer valid. • A7 - IBM i Query requires the access plan to be rebuilt because of system programming changes. • A8 - The CCSID of the current job is different than the CCSID of the job that last created the access plan. • A9 - The value of one or more of the following is different for the current job than it was for the job that last created this access plan: <ul style="list-style-type: none"> – date format – date separator – time format – time separator. |

Table 66. QQQ3006 - Access Plan Rebuilt (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|---|
| Reason_Code (continued) | QQRCOD | <ul style="list-style-type: none"> • 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. • 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. • B5 - The query was reoptimized because MQTs have changed. • B6 - The query was reoptimized because the value of a host variable changed and the access plan is no longer valid. • B7 - The query was reoptimized because AQP determined that the query should be reoptimized. |
| SubCode | QQC21 | If the access plan rebuild reason code was A7 this two-byte hex value identifies which specific reason for A7 forced a rebuild. |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Last_Access_Plan_Rebuild_Timestamp | QQTIM1 | Timestamp of last access plan rebuild |
| Reoptimization_Done | QQC11 | Required optimization for this plan. <ul style="list-style-type: none"> • Y - Yes, plan was really optimized. • N - No, the plan was not reoptimized because of the QAQQINI option for the REOPTIMIZE_ACCESS_PLAN parameter value |
| Previous_Reason_Code | QVC22 | Previous reason code |
| Previous_SubCode | QVC23 | Previous reason subcode |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3007 - Optimizer Timed Out

Displays the SQL logical view format for database monitor QQQ3007.

```

| Create View QQQ3007 as
|   (SELECT QQRID as Row_ID,
|         QQTIME as Time_Created,
|         QQJFLD as Join_Column,
|         QQRDBN as Relational_Database_Name,
|         QQSYS as System_Name,
|         QQJOB as Job_Name,
|         QQUSER as Job_User,
|         QQJNUM as Job_Number,
|         QQI9 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQUDEF as User_Defined,
|         QQQDTN as Unique_SubSelect_Number,
|         QQQDTL as SubSelect_Nested_Level,
|         QQMATN as Materialized_View_Subselect_Number,
|         QQMATL as Materialized_View_Nested_Level,
|         QVP15E as Materialized_View_Union_Level,
|         QVP15A as Decomposed_Subselect_Number,
|         QVP15B as Total_Number_Decomposed_SubSelects,
|         QVP15C as Decomposed_SubSelect_Reason_Code,
|         QVP15D as Starting_Decomposed_SubSelect,
|         QQTLN as System_Table_Schema,
|         QQTFN as System_Table_Name,
|         QQTMN as Member_Name,
|         QQPTLN as System_Base_Table_Schema,
|         QQPTFN as System_Base_Table_Name,
|         QQPTMN as Base_Member_Name,
|         QQ1000 as Index_Names,
|         QQC11 as Optimizer_Timed_Out,
|         QQC301 as Reason_Codes,
|         QVQTBL as Table_Name,
|         QVQLIB as Table_Schema,
|         QVPTBL as Base_Table_Name,
|         QVPLIB as Base_Table_Schema,
|         QQJNP as Join_Position,
|         QQI6 as DataSpace_Number,
|         QQC21 as Join_Method,
|         QQC22 as Join_Type,
|         QQC23 as Join_Operator,
|         QVJFANO as Join_Fanout,
|         QVFILES as Join_Table_Count,
|         QVRCNT as Unique_Refresh_Counter,
|         QQIDXNL as Index_Names_2,
|         QQSMINTF as Plan_iteration_number
|   FROM   UserLib/DBMONTAbLe
|   WHERE  QQRID=3007)

```

Table 67. QQQ3007 - Optimizer Timed Out

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |

| *Table 67. QQQ3007 - Optimizer Timed Out (continued)*

| View Column Name | Table Column Name | Description |
|------------------------------------|--------------------------|--|
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| System_Table_Schema | QQTLN | Schema of table queried |
| System_Table_Name | QQTFN | Name of table queried |
| Member_Name | QQTMN | Member name of table queried |
| System_Base_Table_Schema | QQPTLN | Schema name of base table |
| System_Base_Table_Name | QQPTFN | Name of base table for table queried |
| Base_Member_Name | QQPTMN | Member name of base table |

Table 67. QQQ3007 - Optimizer Timed Out (continued)

| View Column Name | Table Column Name | Description |
|------------------|-------------------|--|
| Index_Names | QQ1000 | <p data-bbox="841 289 1321 310">Names of indexes not used and reason code.</p> <ol style="list-style-type: none"> <li data-bbox="854 323 1414 380">1. Access path was not in a valid state. The system invalidated the access path. <li data-bbox="854 392 1386 449">2. Access path was not in a valid state. The user requested that the access path be rebuilt. <li data-bbox="854 462 1446 539">3. Access path is a temporary access path (resides in library QTEMP) and was not specified as the file to be queried. <li data-bbox="854 552 1430 630">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 data-bbox="854 642 1455 821">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 ALWCOPYDTA(*YES or *NO) is specified. <li data-bbox="854 833 1455 890">6. The keys of the access path did not match the fields specified for the join criteria. <li data-bbox="854 903 1451 1005">7. Use of this access path will not minimize delays when reading records from the file. The user requested to minimize delays when reading records from the file. <li data-bbox="854 1018 1446 1163">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 data-bbox="854 1176 1458 1260">9. File 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 data-bbox="854 1272 1455 1346">10. The user specified ignore decimal data errors on the query. This disallows the use of permanent access paths. |

Table 67. QQQ3007 - Optimizer Timed Out (continued)

| View Column Name | Table Column Name | Description |
|-------------------------|-------------------|---|
| Index_Names (continued) | QQ1000 | <ul style="list-style-type: none"> • 11. The access path contains static select/omit selection criteria which is not compatible with the selection in the query. • 12. The access path contains static select/omit selection criteria whose compatibility with the selection in the query cannot be determined. Either the select/omit criteria or the query selection became too complex during compatibility processing. • 13. The access path contains one or more keys which may be changed by the query during an insert or update. • 14. The access path is being deleted or is being created in an uncommitted unit of work in another process. • 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. • 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. • 17. The left-most key of the access path did not match any fields specified for the selection criteria. Therefore, key row positioning cannot be performed, making the cost to use this access path higher than the cost associated with the chosen access method. • 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 cannot be performed, making the cost to use this access path higher than the cost associated with the chosen access method. • 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. |
| Optimizer_Timed_Out | QQC11 | Optimizer timed out (Y/N) |
| Reason_Codes | QQC301 | List of unique reason codes used by the indexes that timed out (each index has a corresponding reason code associated with it) |
| Table_Name | QVQTBL | Queried table, long name |
| Table_Schema | QVQLIB | Schema of queried table, long name |
| Base_Table_Name | QVPTBL | Base table, long name |
| Base_Table_Schema | QVPLIB | Schema of base table, long name |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI6 | Dataspace number |

Table 67. QQQ3007 - Optimizer Timed Out (continued)

| View Column Name | Table Column Name | Description |
|------------------------|-------------------|---|
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> NL - Nested loop MF - Nested loop with selection HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> IN - Inner join PO - Left partial outer join EX - Exception join |
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Index_Names_2 | QQ1000L | Index names when the list will not fit into QQ1000. Set to null otherwise |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3008 - Subquery Processing

Displays the SQL logical view format for database monitor QQQ3008.

```

Create View QQQ3008 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,
         QQUDEF as User_Defined,
         QQQDTN as Unique_SubSelect_Number,
         QQQDTL as SubSelect_Nested_Level,
         QQMATN as Materialized_View_Subselect_Number,
         QQMATL as Materialized_View_Nested_Level,
         QVP15E as Materialized_View_Union_Level,
         QVP15A as Decomposed_Subselect_Number,
         QQI1 as Original_QDT_Count,

```

```

|         QQI2 as Merged_QDT_Count,
|         QQI3 as Final_QDT_Count,
|         QVRCNT as Unique_Refresh_Counter,
|         QQSMINTF as PlanIterNum
|   FROM   UserLib/DBMONTTable
|   WHERE  QQRID=3008)

```

Table 68. QQQ3008 - Subquery Processing

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Original_QDT_Count | QQI1 | Original number of QDTs |
| Merged_QDT_Count | QQI2 | Number of QDTs merged |
| Final_QDT_Count | QQI3 | Final number of QDTs |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| PlanIterNum | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3010 - Host Variable & ODP Implementation

Displays the SQL logical view format for database monitor QQQ3010.

```

| Create View QQQ3010 as
|   (SELECT QQRID as Row_ID,
|         QQTIME as Time_Created,
|         QQJFLD as Join_Column,
|         QQRDBN as Relational_Database_Name,
|         QSYS as System_Name,
|         QQJOB as Job_Name,
|         QQUSER as Job_User,
|         QQJNUM as Job_Number,
|         QQI9 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQI5 as Unqie_Refresh_Counter2,
|         QQUDEF as User_Defined,
|         QQC11 as ODP_Implementation,
|         QQC12 as Host_Variable_Implementation,

```

```

|         QQ1000 as Host_Variable Values,
|         QVRCNT as Unique_Refresh_Counter,
|         QQDBCLOB1 as DBCLOB_CCSID,
|         QQI7 as DBCLOB_Length,
|         QQINT05 as SQ_Unique_Count,
|         QVC11 as HV_Truncated
|   FROM   UserLib/DBMONTTable
|   WHERE  QQRID=3010)

```

Table 69. QQQ3010 - HostVar & ODP Implementation

| View Column Name | Table Column Name | Description |
|------------------------------|-------------------|---|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| Unquie_Refresh_Counter2 | QQI5 | Unique refresh counter |
| User_Defined | QQUDEF | User defined column |
| ODP_Implementation | QQC11 | ODP implementation <ul style="list-style-type: none"> • R - Reusable ODP • N - Nonreusable ODP • '' - Column not used |
| Host_Variable_Implementation | QQC12 | Host variable implementation <ul style="list-style-type: none"> • I - Interface supplied values (ISV) • V - Host variables treated as literals (V2) • U - Table management row positioning (UP) • S - SQL Insert/Update host variable value |
| Host_Variable_Values | QQ1000 | Host variable values |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| DBCLOB_CCSID | QQDBCLOB1 | Host variables values in a DBCLOB CCSID 1200 field |
| DBCLOB_Length | QQI7 | Length of host variables in the DBCLOB column. |
| SQ_Unique_Count | QQINT05 | A unique count used to uniquely identify statements which do not have an ODP but do pass in host variables. If QQUCNT is 0 and the statement passes in host variables, this value will be non-zero. An example would be a CALL statement. |
| HV_Truncated | QVC11 | Host variable has been truncated (Y/N). |

Database monitor view 3011 - Array Host Variables

Displays the SQL logical view format for database monitor QQQ3011.

```

| Create View QQQ3011 as
|   (SELECT QQRID as Row_ID,
|         QQTIME as Time_Created,
|         QQJFLD as Join_Column,
|         QQRDBN as Relational_Database_Name,

```

```

|         QSYS as System_Name,
|         QQJOB as Job_Name,
|         QQUSER as Job_User,
|         QQJNUM as Job_Number,
|         QQ19 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQUDEF as User_Defined,
|         QQC11 as ODP_Implementation,
|         QQC12 as Array_Variable_Implementation,
|         QQC101 as Array_Name,
|         QVRCNT as Unique_Refresh_Counter,
|         QQDBCLOB1 as Array_Values,
|         QQINT05 as SQ_Unique_Count,
|         QVC11 AS HV_Truncated,
|         QVC1281 as Array_Name,
|         QVC1282 as Array_Library,
|         QQI1 as Max_Cardinality,
|         QQI2 as Cur_Cardinality,
|         QQI3 as Index_Position
|   FROM   UserLib/DBMONTable
|   WHERE  QQRID=3011)

```

Table 70. QQQ3011 - Array Host Variables

| View Column Name | Table Column Name | Description |
|-------------------------------|-------------------|---|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| ODP_Implementation | QQC11 | ODP implementation: <ul style="list-style-type: none"> • R - Reusable ODP • N - Nonreusable ODP • ' ' - Column not used |
| Array_Variable_Implementation | QQC12 | Array variable implementation: <ul style="list-style-type: none"> • I - Interface supplied values (ISV) • S- SQL Insert/Update array variable value |
| Array_Name | QQC101 | Array name generated by the optimizer. Matches the array value in the QQ1000 QQHVAR field in the 3010 record. |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter. |
| Array_Values | QQDBCLOB1 | Array variables values in a DBCLOB CCSID 1200 field (max 1 MB). |
| SQ_Unique_Count | QQINT05 | A unique count used to uniquely identify statements which do not have an ODP but do pass in Arrays. If QQUCNT is 0 and the statement passes in Arrays, this value will be non-zero. An example would be a CALL statement. |
| HV_Truncated | QVC11 | Host variable has been truncated (Y/N). |

Table 70. QQQ3011 - Array Host Variables (continued)

| View Column Name | Table Column Name | Description |
|------------------|-------------------|---|
| Array_Name | QVC1281 | Name of Array UDT. |
| Array_Library | QVC1282 | Library of Array UDT. |
| Max_Cardinality | QQI1 | Maximum cardinality of Array. |
| Cur_Cardinality | QQI2 | Current cardinality of Array. |
| Index_Position | QQI3 | Index position in the Array designated in the QQ1000 QQHVAR field in the 3010 record. |

Database monitor view 3012 - Global Variables

Displays the SQL logical view format for database monitor QQQ3012.

```

Create View QQQ3012 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,
         QQI5 as Unique_Refresh_Counter2,
         QQUDEF as User_Defined,
         QVRCNT as Unique_Refresh_Counter,
         QQDBCLOB1 as DBCLOB_Global_Variable,
         QQINT05 as SQ_Unique_Count,
         QVC11 as GV_Truncated
  FROM   UserLib/DBMONTTable
 WHERE  QQRID=3012)

```

Table 71. QQQ3012 - Global Variables

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| Unique_Refresh_Counter2 | QQI5 | Unique refresh counter |
| User_Defined | QQUDEF | User-defined column |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| DBCLOB_Global_Variable | QQDBCLOB1 | Global session variable values in a DBCLOB CCSID 1200 field. |

Table 71. QQQ3012 - Global Variables (continued)

| View Column Name | Table Column Name | Description |
|------------------|-------------------|--|
| SQ_Unique_Count | QQINT05 | A unique count used to uniquely identify statements which do not have an ODP but do pass in global variables. If QQUCNT is 0 and the statement passes in global variables, this value is non-zero. An example would be a CALL statement. |
| GV_Truncated | QVC11 | Host variable has been truncated (Y/N). |

Database monitor view 3014 - Generic QQ Information

Displays the SQL logical view format for database monitor QQQ3014.

Create View QQQ3014 as

```
(SELECT QQRID as Row_ID,
        QQTIME as Time_Created,
        QQJFLD as Join_Column,
        QQRDBN as Relational_Database_Name,
        QQSYS as System_Name,
        QQJOB as Job_Name,
        QQUSER as Job_User,
        QQJNUM as Job_Number,
        QQI9 as Thread_ID,
        QQUCNT as Unique_Count,
        QQUDEF as User_Defined,
        QQQDTN as Unique_SubSelect_Number,
        QQQDTL as SubSelect_Nested_Level,
        QQMATN as Materialized_View_Subselect_Number,
        QQMATL as Materialized_View_Nested_Level,
        QVP15E as Materialized_View_Union_Level,
        QVP15A as Decomposed_Subselect_Number,
        QVP15B as Total_Number_Decomposed_SubSelects,
        QVP15C as Decomposed_SubSelect_Reason_Code,
        QVP15D as Starting_Decomposed_SubSelect,
        QQREST as Estimated_Rows_Selected,
        QQEPT as Estimated_Processing_Time,
        QQI1 as Open_Time,
        QQORDG as Has_Ordering,
        QQGRPG as Has_Grouping,
        QQJNG as Has_Join,
        QQC22 as Join_Type,
        QQUNIN as Has_Union,
        QQSUBQ as Has_Subquery,
        QWC1F as Has_Scalar_Subselect,
        QQHSTV as Has_Host_Variables,
        QQRCDS as Has_Row_Selection,
        QQC11 as Query_Governor_Enabled,
        QQC12 as Stopped_By_Query_Governor,
        QQC101 as Open_Id,
        QQC102 as Query_Options_Library,
        QQC103 as Query_Options_Table_Name,
        QQC13 as Early_Exit,
        QVRCNT as Unique_Refresh_Counter,
        QQI5 as Optimizer_Time,
        QQTIM1 as Access_Plan_Timestamp,
        QVC11 as Ordering_Implementation,
        QVC12 as Grouping_Implementation,
        QVC13 as Join_Implementation,
        QVC14 as Has_Distinct,
        QVC15 as Is_Distributed,
        QVC3001 as Distributed_Nodes,
        QVC105 as NLSS_Table,
        QVC106 as NLSS_Library,
        QVC16 as ALWCPYDATA,
        QVC21 as Access_Plan_Reason_Code,
```

```

|         QVC22 as Access_Plan_Reason_SubCode,
|         QVC3002 as Summary,
|         QWC16 as Last_Union_Subselect,
|         QVP154 as Query_PoolSize,
|         QVP155 as Query_PoolID,
|         QQI2 as Query_Time_Limit,
|         QVC81 as Parallel_Degree,
|         QQI3 as Max_Number_of_Tasks,
|         QVC17 as Apply_CHGQRYA_Remote,
|         QVC82 as Async_Job_Usage,
|         QVC18 as Force_Join_Order_Indicator,
|         QVC19 as Print_Debug_Messages,
|         QVC1A as Parameter_Marker_Conversion,
|         QQI4 as UDF_Time_Limit,
|         QVC1283 as Optimizer_Limitations,
|         QVC1E as Reoptimize_Requested,
|         QVC87 as Optimize_All_Indexes,
|         QQC14 as Has_Final_Decomposed_QDT,
|         QQC15 as Is_Final_Decomposed_QDT,
|         QQC18 as Read_Trigger,
|         QQC81 as Star_Join,
|         SUBSTR(QVC23,1,1) as Optimization_Goal,
|         SUBSTR(QVC24,1,1) as VE_Diagram_Type,
|         SUBSTR(QVC24,2,1) as Ignore_Like_Redunant_Shifts,
|         QQC23 as Union_QDT,
|         QQC21 as Unicode_Normalization,
|         QVP153 as Pool_Fair_Share,
|         QQC82 as Force_Join_Order_Requested,
|         QVP152 as Force_Join_Order_Dataspace1,
|         QQI6 as No_Parameter_Marker_Reason_Code,
|         QVP151 as Hash_Join_Reason_Code,
|         QQI7 as MQT_Refresh_Age,
|         SUBSTR(QVC42,1,1) as MQT_Usage,
|         QVC43 as SQE_NotUsed_Reason_Code,
|         QVP156 as Estimated_IO_Count,
|         QVP157 as Estimated_Processing_Cost,
|         QVP158 as Estimated_CPU_Cost,
|         QVP159 as Estimated_IO_Cost,
|         SUBSTR(QVC44,1,1) as Has_Implicit_Numeric_Conversion,
|         QVCTIM as Accumulated_Est_Process_Time,
|         QQINT01 as Query_Gov_Storage_Limit,
|         QQINT02 as Estimated_Storage,
|         QQINT03 as Adjusted_Temp_Storage,
|         QQINT04 as Original_Cost_Estimate,
|         QQI8 as Parallel_Degree_Percentage,
|         QFC12 as FieldProc_Encoded_Comparison,
|         QFC13 as Allow_Array_Changes_INI_Opt,
|         QFC11 as SQL_Concurrent_Access_Resolution,
|         QQSMINTF as Plan_Iteration_Number,
|         QXC11 as Warm_IO_Requested,
|         QXC12 as Warm_IO_Used,
|         QXC13 as Optimization_Goal_Override,
|         QXC1E as Plan_Signature_Match
|
| FROM UserLib/DBMONTTable
| WHERE QQRID=3014)

```

Table 72. QQQ3014 - Generic QQ Information

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |

Table 72. QQQ3014 - Generic QQ Information (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|---|
| System_Name | QSYSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User-defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Estimated_Rows_Selected | QQREST | Estimated number of rows selected |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Open_Time | QQI1 | Time spent to open cursor, in milliseconds |
| Has_Ordering | QQORDG | Ordering (Y/N) |
| Has_Grouping | QQGRPG | Grouping (Y/N) |
| Has_Join | QQJNG | Join Query (Y/N) |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |
| Has_Union | QQUNIN | Union Query (Y/N) |
| Has_Subquery | QQSUBQ | Subquery (Y/N) |
| Has_Scalar_Subselect | QWC1F | Scalar Subselects (Y/N) |
| Has_Host_Variables | QQHSTV | Host variables (Y/N) |
| Has_Row_Selection | QQRCDS | Row selection (Y/N) |
| Query_Governor_Enabled | QQC11 | Query governor enabled (Y/N) |
| Stopped_By_Query_Governor | QQC12 | Query governor stopped the query (Y/N) |
| Open_Id | QQC101 | Query open ID |
| Query_Options_Library | QQC102 | Query Options library name |
| Query_Options_Table_Name | QQC103 | Query Options file name |
| Early_Exit | QQC13 | Query early exit value |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |

Table 72. QQQ3014 - Generic QQ Information (continued)

| View Column Name | Table Column Name | Description |
|----------------------------|-------------------|--|
| Optimizer_Time | QQI5 | Time spent in optimizer, in milliseconds |
| Access_Plan_Timestamp | QQTIM1 | Access Plan rebuilt timestamp, last time access plan was rebuilt. |
| Ordering_Implementation | QVC11 | Ordering implementation. Possible values are: <ul style="list-style-type: none"> • I - Index • S - Sort |
| Grouping_Implementation | QVC12 | Grouping implementation. Possible values are: <ul style="list-style-type: none"> • I - Index • H - Hash grouping |
| Join_Implementation | QVC13 | Join Implementation. Possible values are: <ul style="list-style-type: none"> • N - Nested Loop join • H - Hash join • C - Combination of Nested Loop and Hash |
| Has_Distinct | QVC14 | Distinct query (Y/N) |
| Is_Distributed | QVC15 | Distributed query (Y/N) |
| Distributed_Nodes | QVC3001 | Distributed nodes |
| NLSS_Table | QVC105 | Sort Sequence Table |
| NLSS_Library | QVC106 | Sort Sequence Library |
| ALWCPYDATA | QVC16 | ALWCPYDTA setting |
| Access_Plan_Reason_Code | QVC21 | Reason code why access plan was rebuilt |
| Access_Plan_Reason_SubCode | QVC22 | Subcode why access plan was rebuilt |
| Summary | QVC3002 | Summary of query implementation. Shows dataspace number and name of index used for each table being queried. |
| Last_Union_Subselect | QWC16 | Last part (last QDT) of Union (Y/N) |
| Query_PoolSize | QVP154 | Pool size |
| Query_PoolID | QVP155 | Pool id |
| Query_Time_Limit | QQI2 | Query time limit |
| Parallel_Degree | QVC81 | Parallel Degree <ul style="list-style-type: none"> • *SAME - Do not change current setting • *NONE - No parallel processing is allowed • *I/O - Any number of tasks might be used for I/O processing. SMP parallel processing is not allowed. • *OPTIMIZE - The optimizer chooses the number of tasks to use for either I/O or SMP parallel processing. • *MAX - The optimizer chooses to use either I/O or SMP parallel processing. • *SYSVAL - Use the current system value to process the query. • *ANY - Has the same meaning as *I/O. • *NBRTASKS - The number of tasks for SMP parallel processing is specified in column QVTASKN. |
| Max_Number_of_Tasks | QQI3 | Max number of tasks |

Table 72. QQQ3014 - Generic QQ Information (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------|-------------------|---|
| Apply_CHGQRYA_Remote | QVC17 | Apply CHGQRYA remotely (Y/N) |
| Async_Job_Usage | QVC82 | Asynchronous job usage <ul style="list-style-type: none"> *SAME - Do not change current setting *DIST - Asynchronous jobs might be used for queries with distributed tables *LOCAL - Asynchronous jobs might be used for queries with local tables only *ANY - Asynchronous jobs might be used for any database query *NONE - No asynchronous jobs are allowed |
| Force_Join_Order_Indicator | QVC18 | Force join order (Y/N) |
| Print_Debug_Messages | QVC19 | Print debug messages (Y/N) |
| Parameter_Marker_Conversion | QVC1A | Parameter marker conversion (Y/N) |
| UDF_Time_Limit | QQI4 | User Defined Function time limit |
| Optimizer_Limitations | QVC1283 | Optimizer limitations. Possible values: <ul style="list-style-type: none"> *PERCENT followed by 2 byte integer containing the percent value *MAX_NUMBER_OF_RECORDS followed by an integer value that represents the maximum number of rows |
| Reoptimize_Requested | | Reoptimize access plan requested. Possible values are: <ul style="list-style-type: none"> O - Only reoptimize the access plan when required. Do not reoptimize for subjective reasons. Y - Yes, force the access plan to be reoptimized. N - No, do not reoptimize the access plan, unless optimizer determines that it is necessary. May reoptimize for subjective reasons. |
| Optimize_All_Indexes | | Optimize all indexes requested <ul style="list-style-type: none"> *SAME - Do not change current setting *YES - Examine all indexes *NO - Allow optimizer to time out *TIMEOUT - Force optimizer to time out |
| Has_Final-Decomposed_QDT | QQC14 | Final decomposed QDT built indicator (Y/N) |
| Is_Final-Decomposed_QDT | QQC15 | The final decomposed QDT indicator (Y/N) |
| Read_Trigger | QQC18 | One of the files contains a read trigger (Y/N) |
| Star_Join | QQC81 | Star join optimization requested. <ul style="list-style-type: none"> *NO - Star join optimization is not performed. *COST - The optimizer determines if any EVIs can be used for star join optimization. *FORCE - The optimizer adds any EVIs that can be used for star join optimization. |
| Optimization_Goal | QVC23 | Byte 1 = Optimization goal. Possible values are: <ul style="list-style-type: none"> F - First I/O, optimize the query to return the first screen full of rows as quickly as possible. A - All I/O, optimize the query to return all rows as quickly as possible. |

Table 72. QQQ3014 - Generic QQ Information (continued)

| View Column Name | Table Column Name | Description |
|---------------------------------|-------------------|---|
| VE_Diagram_Type | QVC24 | Byte 1 = Type of Visual Explain diagram. Possible values are: <ul style="list-style-type: none"> • D - Detail • B - Basic |
| Ignore_Like_Redunant_Shifts | QVC24 | Byte 2 - Ignore LIKE redundant shifts. Possible values are: <ul style="list-style-type: none"> • O - Optimize, the query optimizer determines which redundant shifts to ignore. • A - All redundant shifts are ignored. |
| Union_QDT | 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). |
| Unicode_Normalization | QQC21 | Unicode data normalization requested (Y/N) |
| Pool_Fair_Share | QVP153 | Fair share of the pool size as determined by the optimizer |
| Force_Join_Order_Requested | QQC82 | Force Join Order requested. Possible values are: <ul style="list-style-type: none"> • *NO - The optimizer was allowed to reorder join files • *YES - The optimizer was not allowed to reorder join files as part of its optimization process • *SQL - The optimizer only forced the join order for those queries that used the SQL JOIN syntax • *PRIMARY - The optimizer was only required to force the primary dial for the join. |
| Force_Join_Order_Dataspace1 | QVP152 | Primary dial to force if Force_Join_Order_Indicator is *PRIMARY. |
| No_Parameter_Marker_Reason_Code | QQI6 | Reason code for why Parameter Marker Conversion was not performed: <ol style="list-style-type: none"> 1. Argument of function must be a literal 2. LOCALTIME or LOCALTIMESTAMP 3. Duration literal in arithmetic expression 4. UPDATE query with no WHERE clause 5. BLOB literal 6. Special register in UPDATE or INSERT with values 7. Result expression for CASE 8. GROUP BY expression 9. ESCAPE character 10. Double Negative value -(-1) 11. INSERT or UPDATE with a mix of literals, parameter markers, and NULLs 12. UPDATE with a mix of literals and parameter markers 13. INSERT with VALUES containing NULL value and expressions 14. UPDATE with list of expressions 99. Parameter marker conversion disabled by QAQQINI |
| Hash_Join_Reason_Code | QVP151 | Reason code why hash join was not used. |

Table 72. QQQ3014 - Generic QQ Information (continued)

| View Column Name | Table Column Name | Description |
|---------------------------------|-------------------|--|
| MQT_Refresh_Age | QQI7 | Value of the MATERIALIZED_QUERY_TABLE_REFRESH_AGE duration. If the QAQQINI parameter value is set to *ANY, the timestamp duration is 99999999999999. |
| MQT_Usage | QVC42,1,1 | Byte 1 - Contains the MATERIALIZED_QUERY_TABLE_USAGE. Possible values are: <ul style="list-style-type: none"> N - *NONE - no materialized query tables used in query optimization and implementation A - *ALL - User-maintained. Refresh-deferred query tables can be used. U - *USER - Only user-maintained materialized query tables can be used. |
| SQE_NotUsed_Reason_Code | QVC43 | SQE Not Used Reason Code. Possible values: <ul style="list-style-type: none"> LF - DDS logical file specified in query definition DK - An index with derived key or select/omit was found over a queried table NF - Too many tables in query NS - Not an SQL query or query not run through an SQL interface DF - Distributed table in query RT - Read Trigger defined on queried table PD - Program described file in query WC - WHERE CURRENT OF a partition table IO - Simple INSERT query CV - Create view statement |
| Estimated_IO_Count | QVP156 | Estimated I/O count |
| Estimated_Processing_Cost | QVP157 | Estimated processing cost in milliseconds |
| Estimated_CPU_Cost | QVP158 | Estimated CPU cost in milliseconds |
| Estimated_IO_Cost | QVP159 | Estimated I/O cost in milliseconds |
| Has_Implicit_Numeric_Conversion | QVC44 | Byte 1: Implicit numeric conversion (Y/N) |
| Accumulated_Est_Process_Time | QVCTIM | Accumulated estimated processing time across all subselects, in seconds. |
| Query_Gov_Storage_Limit | QQINT01 | Specified query governor storage limit, in megabytes |
| Estimated_Storage | QQINT02 | Original estimated temporary storage used, in megabytes. |
| Adjusted_Temp_Storage | QQINT03 | Adjusted temporary storage used, in Adjusted megabytes. This value accumulates the actual time and storage it took to create any temporary indexes and temporary tables. Set by CQE only. |
| Original_Cost_Estimate | QQINT04 | Original cost estimate as determined by the CQE query optimizer. Set by CQE only. |
| Parallel_Degree_Percentage | QQI8 | Percentage specified on Parallel_Degree *OPTIMIZE and *MAX. |

Table 72. QQQ3014 - Generic QQ Information (continued)

| View Column Name | Table Column Name | Description |
|----------------------------------|-------------------|--|
| FieldProc_Encoded_Comparison | QFC12 | <p>FIELDPROC_ENCODED_COMPARISON option active for this query. Specifies the amount of optimization that the optimizer might use when queried columns have attached field procedures.</p> <ul style="list-style-type: none"> • 'N' - NONE • 'E' - ALLOW_EQUAL • 'R' - ALLOW_RANGE • 'A' - ALL |
| Allow_Array_Change_INI_Opt | QFC13 | <p>ALLOW_ARRAY_VALUE_CHANGES QAQQINI option active for this query.</p> <ul style="list-style-type: none"> • 'N' - Do not allow changes to values in arrays referenced in the query to be visible after the query is opened. • 'Y' - Allow changes to values in arrays to be visible to the query while the query is running. |
| SQL_Concurrent_Access_Resolution | QFC11 | <p>SQL_CONCURRENT_ACCESS_RESOLUTION QAQQINI option active for this query.</p> <ul style="list-style-type: none"> • 'U' - USE CURRENTLY COMMITTED • 'W' - WAIT FOR OUTCOME |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number; original optimization = 1 |
| Warm_IO_Requested | QXC11 | <p>Warm I/O value that was requested.</p> <ul style="list-style-type: none"> • 'Y' - Yes, use Warm I/O • 'N' - No, do not use Warm I/O • 'D' - Default |
| Warm_IO_Used | QXC12 | <p>Warm I/O values used to implement the query.</p> <ul style="list-style-type: none"> • 'Y' - Yes, use Warm I/O • 'N' - No, do not use Warm I/O • 'D' - Default |
| Optimization_Goal_Override | QXC13 | <p>Optimization Goal Override.</p> <ul style="list-style-type: none"> • 'O' - Override the specified Optimize For N Rows value and use Optimize For All Rows. • 'D' - Default, use the specified Optimize For N Rows value. |
| Plan_Signature_Match | QXC1E | <p>Plan signature match.</p> <ul style="list-style-type: none"> • 'Y' - New plan matched old plan it replaced; same plan signature. • 'N' - New plan different from old plan it replaced; different plan signature. |

Database monitor view 3015 - Statistics Information

Displays the SQL logical view format for database monitor QQQ3015.

```

Create View QQQ3015 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,

```

```

|         QQJNUM as Job_Number,
|         QQI9 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQUDEF as User_Defined,
|         QQQDTN as Unique_SubSelect_Number,
|         QQQDTL as SubSelect_Nested_Level,
|         QQMATN as Materialized_View_Subselect_Number,
|         QQMATL as Materialized_View_Nested_Level,
|         QVP15E as Materialized_View_Union_Level,
|         QVP15A as Decomposed_Subselect_Number,
|         QVP15B as Total_Number_Decomposed_SubSelects,
|         QVP15C as Decomposed_SubSelect_Reason_Code,
|         QVP15D as Starting_Decomposed_SubSelect,
|         QQTLN as System_Table_Schema,
|         QQTFN as System_Table_Name,
|         QQTMN as Member_Name,
|         QQPTLN as System_Base_Table_Schema,
|         QQPTFN as System_Base_Table_Name,
|         QQPTMN as Base_Member_Name,
|         QVQTBL as Table_Name,
|         QVQLIB as Table_Schema,
|         QVPTBL as Base_Table_Name,
|         QVPLIB as Base_Table_Schema,
|         QQNTNM as NLSS_Table,
|         QQNLNM as NLSS_Library,
|         QQC11 as Statistic_Status,
|         QQI2 as Statistic_Importance,
|         QQ1000 as Statistic_Columns,
|         QVC1000 as Statistic_ID,
|         QQSMINTF as Plan_Iteration_Number
|
| FROM UserLib/DBMONTTable
| WHERE QQRID=3015)

```

Table 73. QQQ3015 - Statistic Information

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |

Table 73. QQQ3015 - Statistic Information (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| System_Table_Schema | QQTLN | Schema of table queried |
| System_Table_Name | QQTFN | Name of table queried |
| Member_Name | QQTMN | Member name of table queried |
| System_Base_Table_Schema | QQPTLN | Schema name of base table |
| System_Base_Table_Name | QQPTFN | Name of the base table queried |
| Base_Member_Name | QQPTMN | Member name of base table |
| Table_Name | QVQTBL | Queried table, long name |
| Table_Schema | QVQLIB | Schema of queried table, long name |
| Base_Table_Name | QVPTBL | Base table, long name |
| Base_Table_Schema | QVPLIB | Schema of base table, long name |
| NLSS_Table | QQNTNM | NLSS table |
| NLSS_Library | QQNLNM | NLSS library |
| Statistic_Status | QQC11 | Statistic Status. Possible values are: <ul style="list-style-type: none"> • 'N' - No statistic • 'S' - Stale statistic • '' - Unknown |
| Statistic_Importance | QQI2 | Importance of this statistic |
| Statistic_Columns | QQ1000 | Columns for the statistic advised |
| Statistic_ID | QVC1000 | Statistic identifier |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3018 - STRDBMON/ENDDBMON

Displays the SQL logical view format for database monitor QQQ3018.

```

Create View QQQ3018 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQC11 as Monitored_Job_type,
         QQC12 as Monitor_Command,
         QQC301 as Monitor_Job_Information,
         QQ1000L as STRDBMON_Command_Text,
         QQC101 as Monitor_ID,
         QQC102 as Version_Release_Mod,
         QQC103 as Group_PTF,
         QVC11 as Initial_AQP_Processing
  FROM   UserLib/DBMONTAbTe
  WHERE  QQRID=3018)

```

| *Table 74. QQQ3018 - STRDBMON/ENDDBMON*

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Monitored_Job_type | QQC11 | Type of job monitored <ul style="list-style-type: none"> • C - Current • J - Job name • A - All |
| Monitor_Command | QQC12 | Command type <ul style="list-style-type: none"> • S - STRDBMON • E - ENDDBMON |
| Monitor_Job_Information | QQC301 | Monitored job information <ul style="list-style-type: none"> • * - Current job • Job number/User/Job name • *ALL - All jobs |
| STRDBMON_Command_Text | QQ1000L | STRDBMON command text. |
| Monitor_ID | QQC101 | Monitor ID |
| Version_Release_Mod | QQC102 | Version Release and modification level |
| Group_PTF | QQC103 | Installed Group PTF number and level. For example, 'SF99601 3' indicates that Database Group release V6R1M0, version 3 is installed. |
| Initial_AQP_Processing | QVC11 | Initial AQP processing. <p>Y indicates that the monitor file has already been processed to handle the AQP plan iteration number. All iterations other than the last one have been changed to a negative number.</p> <p>All other values indicate that the monitor file has not been processed.</p> |

Database monitor view 3019 - Rows retrieved

Displays the SQL logical view format for database monitor QQQ3019.

```

Create View QQQ3019 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,

```

QQUCNT as Unique_Count,
 QQUDEF as User_Defined,
 QQQDTN as Unique_SubSelect_Number,
 QQQDTL as SubSelect_Nested_Level,
 QQMATN as Materialized_View_Subselect_Number,
 QQMATL as Materialized_View_Nested_Level,
 QVP15E as Materialized_View_Union_Level,
 QVP15A as Decomposed_Subselect_Number,
 QVP15B as Total_Number_Decomposed_SubSelects,
 QVP15C as Decomposed_SubSelect_Reason_Code,
 QVP15D as Starting_Decomposed_SubSelect,
 QQI1 as CPU_Time_to_Return_All_Rows,
 QQI2 as Clock_Time_to_Return_All_Rows,
 QQI3 as Number_Synchronous_Database_Reads,
 QQI4 as Number_Synchronous_Database_Writes,
 QQI5 as Number_Asynchronous_Database_Reads,
 QQI6 as Number_Asynchronous_Database_Writes,
 QVP151 as Number_Page_Faults,
 QQI7 as Number_Rows_Returned,
 QQI8 as Number_of_Calls_for_Returned_Rows,
 QVP15F as Number_of_Times_Statement_was_Run,
 QQINT03 as Temporary_Storage,
 QQC11 as DBMON_Temp_Result_Reused,
 QQC21 as DBMON_Temp_Reused_RC,
 QQINT01 as DBMON_Temp_Reuse_Count,
 QQIA as Skip_Lock_Row_Count,
 QQINT05 as Skip_Lock_Row_Runs,
 QQINT06 as Skip_Lock_On_Runs,
 QQF1 as Adjusted_Average_Run_Time,
 QVRCNT as Unique_Refresh_Counter,
 QVP152 as Committed_Journal_Search_Requests,
 QVP153 as Committed_Journal_Search_Failures,
 QVP154 as Committed_Journal_Search_Time_Limit
FROM UserLib/DBMONTTable
WHERE QQRID=3019)

Table 75. QQQ3019 - Rows retrieved

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|------------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |

Table 75. QQQ3019 - Rows retrieved (continued)

| View Column Name | Table Column Name | Description |
|-------------------------------------|-------------------|---|
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| CPU_Time_to_Return_All_Rows | QQI1 | CPU time to return all rows, in milliseconds |
| Clock_Time_to_Return_All_Rows | QQI2 | Clock time to return all rows, in milliseconds |
| Number_Synchronous_Database_Reads | QQI3 | Number of synchronous database reads |
| Number_Synchronous_Database_Writes | QQI4 | Number of synchronous database writes |
| Number_Asynchronous_Database_Reads | QQI5 | Number of asynchronous database reads |
| Number_Asynchronous_Database_Writes | QQI6 | Number of asynchronous database writes |
| Number_Page_Faults | QVP151 | Number of page faults |
| Number_Rows_Returned | QQI7 | Number of rows returned |
| Number_of_Calls_for_Returned_Rows | QQI8 | Number of calls to retrieve rows returned |
| Number_of_Times_Statement_was_Run | QVP15F | Number of times this Statement was run. If Null, then the statement was run once. |
| Temporary_Storage | QQINT03 | Amount of temporary storage used. |
| DBMON_Temp_Result_Reused | QQC11 | Indicates if the DBMON temporary result was reused (Y/N). |
| DBMON_Temp_Reused_RC | QQC21 | Reason code why the DBMON temporary result was reused. |
| DBMON_Temp_Reuse_Count | QQINT01 | Number of times the DBMON temporary result was reused. |
| Skip_Lock_Row_Count | QQIA | Number of locked rows that were skipped. |
| Skip_Lock_Row_Runs | QQINT05 | Number of runs where some rows were skipped. |
| Skip_Lock_On_Runs | QQINT06 | Number of runs where Skip Lock was active. |
| Adjusted_Average_Run_Time | QQF1 | Average runtime for the query, adjusted to not include individual runs that are well outside the norm. Units in microseconds. |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Committed_Journal_Search_Requests | QVP152 | Number of times the database manager searched the journal for the currently committed version of a record. |
| Committed_Journal_Search_Failures | QVP153 | Number of times the database manager failed to find the currently committed version of a record in the journal. |
| Committed_Journal_Search_Time_Limit | QVP154 | Maximum amount of time allowed to search the journal for the currently committed version of a record. |

Database monitor view 3020 - Index advised (SQE)

Displays the SQL logical view format for database monitor QQQ3020.

```

| Create View QQQ3020 as
|   (SELECT QQRID as Row_ID,
|         QQTIME as Time_Created,

```

```

|         QQJFLD as Join_Column,
|         QQRDBN as Relational_Database_Name,
|         QQSYS as System_Name,
|         QQJOB as Job_Name,
|         QQUSER as Job_User,
|         QQJNUM as Job_Number,
|         QQI9 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQUDEF as User_Defined,
|         QQQDTN as Unique_SubSelect_Number,
|         QQQDTL as SubSelect_Nested_Level,
|         QQMATN as Materialized_View_Subselect_Number,
|         QQMATL as Materialized_View_Nested_Level,
|         QVP15E as Materialized_View_Union_Level,
|         QVP15A as Decomposed_Subselect_Number,
|         QVP15B as Total_Number_Decomposed_SubSelects,
|         QVP15C as Decomposed_SubSelect_Reason_Code,
|         QVP15D as Starting_Decomposed_SubSelect,
|         QQTLN as System_Table_Schema,
|         QQTFN as System_Table_Name,
|         QQTMN as Member_Name,
|         QQPTLN as System_Base_Table_Schema,
|         QQPTFN as System_Base_Table_Name,
|         QQPTMN as Base_Member_Name,
|         QVPLIB as Base_Table_Schema,
|         QVPTBL as Base_Table_Name,
|         QQTOTR as Table_Total_Rows,
|         QQEPT as Estimated_Processing_Time,
|         QQIDXA as Index_is_Advised,
|         QQIDXN as Index_Advised_Columns_Short_List,
|         QQ1000L as Index_Advised_Columns_Long_List,
|         QQI1 as Number_of_Advised_Columns,
|         QQI2 as Number_of_Advised_Primary_Columns,
|         QQRCOD as Reason_Code,
|         QVRCNT as Unique_Refresh_Counter,
|         QVC1F as Type_of_Index_Advised,
|         QQNTNM as NLSS_Table,
|         QQNLNM as NLSS_Library,
|         QQSMINTF as Plan_Iteration_Number
| FROM UserLib/DBMONTTable
| WHERE QQRID=3020)

```

Table 76. QQQ3020 - Index advised (SQE)

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|---------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |

Table 76. QQQ3020 - Index advised (SQE) (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|---|
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| System_Table_Schema | QQTLN | Schema of table queried |
| System_Table_Name | QQTFN | Name of table queried |
| Member_Name | QQTMN | Member name of table queried |
| System_Base_Table_Schema | QQPTLN | Schema name of base table |
| System_Base_Table_Name | QQPTFN | Name of base table for table queried |
| Base_Member_Name | QQPTMN | Member of base table |
| Base_Table_Schema | QVPLIB | Schema of base table, long name |
| Base_Table_Name | QVPTBL | Base table, long name |
| Table_Total_Rows | QQTOTR | Number of rows in the table |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Index_is_Advised | QQIDXA | Index advised (Y/N) |
| Index_Advised_Columns_Short_List | QQIDXD | Columns for the index advised, first 1000 bytes |
| Index_Advised_Columns_Long_List | QQ1000L | Column for the index advised |
| Number_of_Advised_Columns | QQI1 | Number of indexes advised |
| Number_of_Advised_Primary_Columns | QQI2 | Number of advised columns that use index scan-key positioning |
| Reason_Code | QQRCOD | Reason code <ul style="list-style-type: none"> • I1 - Row selection • I2 - Ordering/Grouping • I3 - Row selection and Ordering/Grouping • I4 - Nested loop join • I5 - Row selection using bitmap processing |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Type_of_Index_Advised | QVC1F | Type of index advised. Possible values are: <ul style="list-style-type: none"> • B - Radix index • E - Encoded vector index |
| NLSS_Table | QQNTNM | Sort Sequence Table |
| NLSS_Library | QQNLNM | Sort Sequence Library |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Related reference:

“Index advisor” on page 137

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.

Database monitor view 3021 - Bitmap Created

Displays the SQL logical view format for database monitor QQQ3021.

```

| Create View QQQ3021 as
|   (SELECT QQRID as Row_ID,
|         QQTIME as Time_Created,
|         QQJFLD as Join_Column,
|         QQRDBN as Relational_Database_Name,
|         QQSYS as System_Name,
|         QQJOB as Job_Name,
|         QQUSER as Job_User,
|         QQJNUM as Job_Number,
|         QQI9 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQUDEF as User_Defined,
|         QQQDTN as Unique_SubSelect_Number,
|         QQQDTL as SubSelect_Nested_Level,
|         QQMATN as Materialized_View_Subselect_Number,
|         QQMATL as Materialized_View_Nested_Level,
|         QVP15E as Materialized_View_Union_Level,
|         QVP15A as Decomposed_Subselect_Number,
|         QVP15B as Total_Number_Decomposed_SubSelects,
|         QVP15C as Decomposed_SubSelect_Reason_Code,
|         QVP15D as Starting_Decomposed_SubSelect,
|         QVRCNT as Unique_Refresh_Counter,
|         QVPPARPF as Parallel_Prefetch,
|         QVPPARPL as Parallel_PreLoad,
|         QVPPARD as Parallel_Degree_Requested,
|         QVPPARU as Parallel_Degree_Used,
|         QVPPARRC as Parallel_Degree_Reason_Code,
|         QQEPT as Estimated_Processing_Time,
|         QVCTIM as Estimated_Cumulative_Time,
|         QQREST as Estimated_Rows_Selected,
|         QQAJN as Estimated_Join_Rows,
|         QQJNP as Join_Position,
|         QQI6 as DataSpace_Number,
|         QQC21 as Join_Method,
|         QQC22 as Join_Type,
|         QQC23 as Join_Operator,
|         QVJFANO as Join_Fanout,
|         QVFILES as Join_Table_Count,
|         QQI2 as Bitmap_Size,
|         QVP151 as Bitmap_Count,
|         QVC3001 as Bitmap_IDs,
|         QQINT03 as Storage_Estimate,
|         QQSMINTF as Plan_Iteration_Number
|   FROM   UserLib/DBMONTTable
|   WHERE  QQRID=3021)

```

Table 77. QQQ3021 - Bitmap Created

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |

Table 77. QQQ3021 - Bitmap Created (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (index used) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested (index used) |
| Parallel_Degree_Used | QVPARU | Parallel degree used (index used) |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited (index used) |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Rows_Selected | QQREST | Estimated rows selected |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI6 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |

Table 77. QQQ3021 - Bitmap Created (continued)

| View Column Name | Table Column Name | Description |
|-----------------------|-------------------|---|
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Bitmap_Size | QQI2 | Bitmap size |
| Bitmap_Count | QVP151 | Number of bitmaps created |
| Bitmap_IDs | QVC3001 | Internal bitmap IDs |
| Storage_Estimate | QQINT03 | Estimated amount of temporary storage used, in megabytes, to create the temporary index. |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1. |

Database monitor view 3022 - Bitmap Merge

Displays the SQL logical view format for database monitor QQQ3022

```

Create View QQQ3022 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,
         QQUDEF as User_Defined,
         QQQDTN as Unique_SubSelect_Number,
         QQQDTL as SubSelect_Nested_Level,
         QQMATN as Materialized_View_Subselect_Number,
         QQMATL as Materialized_View_Nested_Level,
         QVP15E as Materialized_View_Union_Level,
         QVP15A as Decomposed_Subselect_Number,
         QVP15B as Total_Number_Decomposed_SubSelects,
         QVP15C as Decomposed_SubSelect_Reason_Code,
         QVP15D as Starting_Decomposed_SubSelect,
         QVRCNT as Unique_Refresh_Counter,
         QVPARPF as Parallel_Prefetch,
         QVPARPL as Parallel_PreLoad,
         QVPARD as Parallel_Degree_Requested,
         QVPARU as Parallel_Degree_Used,

```

```

|         QVPARRC as Parallel_Degree_Reason_Code,
|         QQEPT as Estimated_Processing_Time,
|         QVCTIM as Estimated_Cumulative_Time,
|         QQREST as Estimated_Rows_Selected,
|         QQAJN as Estimated_Join_Rows,
|         QQJNP as Join_Position,
|         QQI6 as DataSpace_Number,
|         QQC21 as Join_Method,
|         QQC22 as Join_Type,
|         QQC23 as Join_Operator,
|         QVJFANO as Join_Fanout,
|         QVFILES as Join_Table_Count,
|         QQI2 as Bitmap_Size,
|         QVC101 as Bitmap_ID,
|         QVC3001 as Bitmaps_Merged,
|         QQINT03 as Storage_Estimate,
|         QQSMINTF as Plan_Iteration_Number
| FROM   UserLib/DBMONTTable
| WHERE  QQRID=3022)

```

Table 78. QQQ3022 - Bitmap Merge

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (index used) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested (index used) |
| Parallel_Degree_Used | QVPARU | Parallel degree used (index used) |

Table 78. QQQ3022 - Bitmap Merge (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------|-------------------|---|
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited (index used) |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Rows_Selected | QQREST | Estimated rows selected |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI6 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> NL - Nested loop MF - Nested loop with selection HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> IN - Inner join PO - Left partial outer join EX - Exception join |
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Bitmap_Size | QQI2 | Bitmap size |
| Bitmap_ID | QVC101 | Internal bitmap ID |
| Bitmaps_Merged | QVC3001 | IDs of bitmaps merged together |
| Storage_Estimate | QQINT03 | Estimated amount of temporary storage used, in megabytes, to create the final bitmap. Only set by CQE |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3023 - Temp Hash Table Created

Displays the SQL logical view format for database monitor QQQ3023.

```

Create View QQQ3023 as
(SELECT QQRID as Row_ID,
      QQTIME as Time_Created,
      QQJFLD as Join_Column,

```

```

|         QQRDBN as Relational_Database_Name,
|         QQSYS as System_Name,
|         QQJOB as Job_Name,
|         QQUSER as Job_User,
|         QQJNUM as Job_Number,
|         QQI9 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQUDEF as User_Defined,
|         QQQDTN as Unique_SubSelect_Number,
|         QQQDTL as SubSelect_Nested_Level,
|         QQMATN as Materialized_View_Subselect_Number,
|         QQMATL as Materialized_View_Nested_Level,
|         QVP15E as Materialized_View_Union_Level,
|         QVP15A as Decomposed_Subselect_Number,
|         QVP15B as Total_Number_Decomposed_SubSelects,
|         QVP15C as Decomposed_SubSelect_Reason_Code,
|         QVP15D as Starting_Decomposed_SubSelect,
|         QVRCNT as Unique_Refresh_Counter,
|         QVPARPF as Parallel_Prefetch,
|         QVPARPL as Parallel_PreLoad,
|         QVPARD as Parallel_Degree_Requested,
|         QVPARU as Parallel_Degree_Used,
|         QVPARRC as Parallel_Degree_Reason_Code,
|         QQEPT as Estimated_Processing_Time,
|         QVCTIM as Estimated_Cumulative_Time,
|         QQREST as Estimated_Rows_Selected,
|         QQAJN as Estimated_Join_Rows,
|         QQJNP as Join_Position,
|         QQI6 as DataSpace_Number,
|         QQC21 as Join_Method,
|         QQC22 as Join_Type,
|         QQC23 as Join_Operator,
|         QVJFANO as Join_Fanout,
|         QVFILES as Join_Table_Count,
|         QVC1F as HashTable_ReasonCode,
|         QQI2 as HashTable_Entries,
|         QQI3 as HashTable_Size,
|         QQI4 as HashTable_Row_Size,
|         QQI5 as HashTable_Key_Size,
|         QQIA as HashTable_Element_Size,
|         QQI7 as HashTable_PoolSize,
|         QQI8 as HashTable_PoolID,
|         QVC101 as HashTable_Name,
|         QVC102 as HashTable_Library,
|         QVC3001 as HashTable_Columns,
|         QQINT03 as Storage_Estimate,
|         QQSMINTF as Plan_Iteration_Number
|
| FROM   UserLib/DBMONTAbLe
| WHERE  QQRID=3023)

```

Table 79. QQQ3023 - Temp Hash Table Created

| View Column Name | Table Column Name | Description |
|--------------------------|-------------------|------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQIFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |

Table 79. QQQ3023 - Temp Hash Table Created (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (index used) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested (index used) |
| Parallel_Degree_Used | QVPARU | Parallel degree used (index used) |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited (index used) |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Rows_Selected | QQREST | Estimated rows selected |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI6 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |

Table 79. QQQ3023 - Temp Hash Table Created (continued)

| View Column Name | Table Column Name | Description |
|------------------------|-------------------|---|
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| HashTable_ReasonCode | QVC1F | Hash table reason code <ul style="list-style-type: none"> • J - Created for hash join • G - Created for hash grouping |
| HashTable_Entries | QQI2 | Hash table entries |
| HashTable_Size | QQI3 | Hash table size |
| HashTable_Row_Size | QQI4 | Hash table row size |
| HashTable_Key_Size | QQI5 | Hash table key size |
| HashTable_Element_Size | QQIA | Hash table element size |
| HashTable_PoolSize | QQI7 | Hash table pool size |
| HashTable_PoolID | QQI8 | Hash table pool ID |
| HashTable_Name | QVC101 | Hash table internal name |
| HashTable_Library | QVC102 | Hash table library |
| HashTable_Columns | QVC3001 | Columns used to create hash table |
| Storage_Estimate | QQINT03 | Estimated amount of temporary storage used, in megabytes, to create the hash table. Only set by CQE. |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3025 - Distinct Processing

Displays the SQL logical view format for database monitor QQQ3025.

```

Create View QQQ3025 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,

```

```

|         QQUDEF as User_Defined,
|         QQQDTN as Unique_SubSelect_Number,
|         QQQDTL as SubSelect_Nested_Level,
|         QQMATN as Materialized_View_Subselect_Number,
|         QQMATL as Materialized_View_Nested_Level,
|         QVP15E as Materialized_View_Union_Level,
|         QVP15A as Decomposed_Subselect_Number,
|         QVP15B as Total_Number_Decomposed_SubSelects,
|         QVP15C as Decomposed_SubSelect_Reason_Code,
|         QVP15D as Starting_Decomposed_SubSelect,
|         QVRCNT as Unique_Refresh_Counter,
|         QVPARPF as Parallel_Prefetch,
|         QVPARPL as Parallel_Preload,
|         QVPARD as Parallel_Degree_Requested,
|         QVPARU as Parallel_Degree_Used,
|         QVPARRC as Parallel_Degree_Reason_Code,
|         QQEPT as Estimated_Processing_Time,
|         QVCTIM as Estimated_Cumulative_Time,
|         QQREST as Estimated_Rows_Selected,
|         QQSMINTF as Plan_Iteration_Number
| FROM UserLib/DBMONTAbLe
| WHERE QQRID=3025)

```

Table 80. QQQ3025 - Distinct Processing

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_Preload | QVPARPL | Parallel Preload (index used) |

Table 80. QQQ3025 - Distinct Processing (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------|-------------------|--|
| Parallel_Degree_Requested | QVPARD | Parallel degree requested (index used) |
| Parallel_Degree_Used | QVPARU | Parallel degree used (index used) |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited (index used) |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Rows_Selected | QQREST | Estimated rows selected |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3026 - Set operation

Displays the SQL logical view format for database monitor QQQ3026.

```

Create View QQQ3026 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,
         QQUDEF as User_Defined,
         QQQDTN as Unique_SubSelect_Number,
         QQQDTL as SubSelect_Nested_Level,
         QQMATN as Materialized_View_Subselect_Number,
         QQMATL as Materialized_View_Nested_Level,
         QVP15E as Materialized_View_Union_Level,
         QVP15A as Decomposed_Subselect_Number,
         QVP15B as Total_Number_Decomposed_SubSelects,
         QVP15C as Decomposed_SubSelect_Reason_Code,
         QVP15D as Starting_Decomposed_SubSelect,
         QVRCNT as Unique_Refresh_Counter,
         QVPARPF as Parallel_Prefetch,
         QVPARPL as Parallel_PreLoad,
         QVPARD as Parallel_Degree_Requested,
         QVPARU as Parallel_Degree_Used,
         QVPARRC as Parallel_Degree_Reason_Code,
         QQEPT as Estimated_Processing_Time,
         QVCTIM as Estimated_Cumulative_Time,
         QQREST as Estimated_Rows_Selected,
         QQC11 as Union_Type,
         QVFILES as Join_Table_Count,
         QQUNIN as Has_Union,
         QWC16 as Last_Union_Subselect,
         QQC23 as Set_in_a_View,
         QQC22 as Set_Operator,
         QQSMINTF as Plan_Iteration_Number
  FROM   UserLib/DBMONTAb1e
  WHERE  QQRID=3026)

```

Table 81. QQQ3026 - Set operation

| View Column Name | Table Column Name | Description |
|------------------|-------------------|--------------------|
| Row_ID | QQRID | Row identification |

Table 81. QQQ3026 - Set operatoin (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (Y/N) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested |
| Parallel_Degree_Used | QVPARU | Parallel degree used |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Rows_Selected | QQREST | Estimated number of rows selected |
| Union_Type | QQC11 | Type of union. Possible values are: <ul style="list-style-type: none"> • A - Union All • U - Union |
| Join_Table_Count | QVFILES | Number of tables queried |
| Has_Union | QQUNIN | Union subselect (Y/N) |
| Last_Union_Subselect | QWC16 | This is the last subselect, or only subselect, for the query. (Y/N) |

Table 81. QQQ3026 - Set operatoin (continued)

| View Column Name | Table Column Name | Description |
|-----------------------|-------------------|--|
| Set_in_a_View | QQC23 | Set operation within a view. <ul style="list-style-type: none"> Byte 1 of 2 (Y/N): This subselect is part of a query that is contained within a view and it contains a set operation (for example, Union). Byte 2 of 2 (Y/N): This is the last subselect of the query that is contained within a view. |
| Set_Operator | QQC22 | Type of set operation. Possible values are: <ul style="list-style-type: none"> UU - Union UA - Union All UR - Union Recursive EE - Except EA - Except All II - Intersect IA - Intersect All |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3027 - Subquery Merge

Displays the SQL logical view format for database monitor QQQ3027.

```

Create View QQQ3027 as
(SELECT QQRID as Row_ID,
      QQTIME as Time_Created,
      QQJFLD as Join_Column,
      QQRDBN as Relational_Database_Name,
      QQSYS as System_Name,
      QQJOB as Job_Name,
      QQUSER as Job_User,
      QQJNUM as Job_Number,
      QQI9 as Thread_ID,
      QQUCNT as Unique_Count,
      QQUDEF as User_Defined,
      QQQDTN as Unique_SubSelect_Number,
      QQQDTL as SubSelect_Nested_Level,
      QQMATN as Materialized_View_Subselect_Number,
      QQMATL as Materialized_View_Nested_Level,
      QVP15E as Materialized_View_Union_Level,
      QVP15A as Decomposed_Subselect_Number,
      QVP15B as Total_Number_Decomposed_SubSelects,
      QVP15C as Decomposed_SubSelect_Reason_Code,
      QVP15D as Starting_Decomposed_SubSelect,
      QVRCNT as Unique_Refresh_Counter,
      QVPARPF as Parallel_Prefetch,
      QVPARPL as Parallel_PreLoad,
      QVPARD as Parallel_Degree_Requested,
      QVPARU as Parallel_Degree_Used,
      QVPARRC as Parallel_Degree_Reason_Code,
      QQEPT as Estimated_Processing_Time,
      QVCTIM as Estimated_Cumulative_Time,
      QQREST as Estimated_Rows_Selected,
      QQAJN as Estimated_Join_Rows,
      QQJNP as Join_Position,
      QQI1 as DataSpace_Number,
      QQC21 as Join_Method,
      QQC22 as Join_Type,
      QQC23 as Join_Operator,
      QVJFANO as Join_Fanout,
      QVFILES as Join_Table_Count,

```

```

|         QVP151 as Subselect_Number_of_Inner_Subquery,
|         QVP152 as Subselect_Level_of_Inner_Subquery,
|         QVP153 as Materialized_View_Subselect_Number_of_Inner,
|         QVP154 as Materialized_View_Nested_Level_of_Inner,
|         QVP155 as Materialized_View_Union_Level_of_Inner,
|         QQC101 as Subquery_Operator,
|         QVC21 as Subquery_Type,
|         QQC11 as Has_Correlated_Columns,
|         QVC3001 as Correlated_Columns,
|         QQSMINTF as Plan_Iteration_Number
| FROM   UserLib/DBMONTAb1e
| WHERE  QQRID=3027)

```

Table 82. QQQ3027 - Subquery Merge

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Subselect number for outer subquery |
| SubSelect_Nested_Level | QQQDTL | Subselect level for outer subquery |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number for outer subquery |
| Materialized_View_Nested_Level | QQMATL | Materialized view subselect level for outer subquery |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (index used) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested (index used) |
| Parallel_Degree_Used | QVPARU | Parallel degree used (index used) |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited (index used) |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Rows_Selected | QQREST | Estimated rows selected |

Table 82. QQQ3027 - Subquery Merge (continued)

| View Column Name | Table Column Name | Description |
|---|-------------------|---|
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Join_Position | QQJNP | Join position - when available |
| DataSpace_Number | QQI6 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> NL - Nested loop MF - Nested loop with selection HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> IN - Inner join PO - Left partial outer join EX - Exception join |
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| Subselect_Number_of_Inner_Subquery | QVP151 | Subselect number for inner subquery |
| Subselect_Level_of_Inner_Subquery | QVP152 | Subselect level for inner subquery |
| Materialized_View_Subselect_Number_of_Inner | QVP153 | Materialized view subselect number for inner subquery |
| Materialized_View_Nested_Level_of_Inner | QVP154 | Materialized view subselect level for inner subquery |
| Materialized_View_Union_Level_of_Inner | QVP155 | Materialized view union level for inner subquery |

Table 82. QQQ3027 - Subquery Merge (continued)

| View Column Name | Table Column Name | Description |
|------------------------|-------------------|--|
| Subquery_Operator | QQC101 | Subquery operator. Possible values are: <ul style="list-style-type: none"> • EQ - Equal • NE - Not Equal • LT - Less Than or Equal • LT - Less Than • GE - Greater Than or Equal • GT - Greater Than • IN • LIKE • EXISTS • NOT - Can precede IN, LIKE or EXISTS |
| Subquery_Type | QVC21 | Subquery type. Possible values are: <ul style="list-style-type: none"> • SQ - Subquery • SS - Scalar subselect • SU - Set Update |
| Has_Correlated_Columns | QQC11 | Correlated columns exist (Y/N) |
| Correlated_Columns | QVC3001 | List of correlated columns with corresponding QDT number |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3028 - Grouping

Displays the SQL logical view format for database monitor QQQ3028.

```

Create View QQQ3028 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QQSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,
         QQUDEF as User_Defined,
         QQQDTN as Unique_SubSelect_Number,
         QQQDTL as SubSelect_Nested_Level,
         QQMATN as Materialized_View_Subselect_Number,
         QQMATL as Materialized_View_Nested_Level,
         QVP15E as Materialized_View_Union_Level,
         QVP15A as Decomposed_Subselect_Number,
         QVP15B as Total_Number_Decomposed_SubSelects,
         QVP15C as Decomposed_SubSelect_Reason_Code,
         QVP15D as Starting_Decomposed_SubSelect,
         QVRCNT as Unique_Refresh_Counter,
         QVPARPF as Parallel_Prefetch,
         QVPARPL as Parallel_PreLoad,
         QVPARD as Parallel_Degree_Requested,
         QVPARU as Parallel_Degree_Used,
         QVPARRC as Parallel_Degree_Reason_Code,
         QQEPT as Estimated_Processing_Time,
         QVCTIM as Estimated_Cumulative_Time,
         QQREST as Estimated_Rows_Selected,
         QQAJN as Estimated_Join_Rows,

```

```

|         QQJNP as Join_Position,
|         QQI1 as DataSpace_Number,
|         QQC21 as Join_Method,
|         QQC22 as Join_Type,
|         QQC23 as Join_Operator,
|         QVJFANO as Join_Fanout,
|         QVFILES as Join_Table_Count,
|         QQC11 as GroupBy_Implementation,
|         QQC101 as GroupBy_Index_Name,
|         QQC102 as GroupBy_Index_Library,
|         QVINAM as GroupBy_Index_Long_Name,
|         QVILIB as GroupBy_Index_Long_Library,
|         QQC12 as Has_Having_Selection,
|         QQC13 as Having_to_Where_Selection_Conversion,
|         QQI2 as Estimated_Number_of_Groups,
|         QQI3 as Average_Number_Rows_per_Group,
|         QVC3001 as GroupBy_Columns,
|         QVC3002 as MIN_Columns,
|         QVC3003 as MAX_Columns,
|         QVC3004 as SUM_Columns,
|         QVC3005 as COUNT_Columns,
|         QVC3006 as AVG_Columns,
|         QVC3007 as STDDEV_Columns,
|         QVC3008 as VAR_Columns,
|         QQSMINTF as Plan_Iteration_Number
| FROM UserLib/DBMONTAbTe
| WHERE QQRID=3028)

```

Table 83. QQQ3028 - Grouping

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job user |
| Job_Number | QQJNUM | Job number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |

Table 83. QQQ3028 - Grouping (continued)

| View Column Name | Table Column Name | Description |
|-----------------------------|-------------------|---|
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (index used) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested (index used) |
| Parallel_Degree_Used | QVPARU | Parallel degree used (index used) |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited (index used) |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Rows_Selected | QQREST | Estimated rows selected |
| Estimated_Join_Rows | QQAJN | Estimated number of joined rows |
| Join_Position | QQJNP | Join position |
| DataSpace_Number | QQI1 | Dataspace number |
| Join_Method | QQC21 | Join method - when available <ul style="list-style-type: none"> • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join |
| Join_Type | QQC22 | Join type - when available <ul style="list-style-type: none"> • IN - Inner join • PO - Left partial outer join • EX - Exception join |
| Join_Operator | QQC23 | Join operator - when available <ul style="list-style-type: none"> • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product |
| Join_Fanout | QVJFANO | Join fan out. Possible values are: <ul style="list-style-type: none"> • N - Normal join situation where fanout is allowed and each matching row of the join fanout is returned. • D - Distinct fanout. Join fanout is allowed however none of the join fanout rows are returned. • U - Unique fanout. Join fanout is not allowed. Error situation if join fanout occurs. |
| Join_Table_Count | QVFILES | Number of tables joined |
| GroupBy_Implementation | QQC11 | Group by implementation <ul style="list-style-type: none"> • ' ' - No grouping • I - Index • H - Hash |
| GroupBy_Index_Name | QQC101 | Index, or constraint, used for grouping |
| GroupBy_Index_Library | QQC102 | Library of index used for grouping |

Table 83. QQQ3028 - Grouping (continued)

| View Column Name | Table Column Name | Description |
|--------------------------------------|-------------------|--|
| GroupBy_Index_Long_Name | QVINAM | Long name of index, or constraint, used for grouping |
| GroupBy_Index_Long_Library | QVILIB | Long name of index, or constraint, library used for grouping |
| Has_Having_Selection | QQC12 | Having selection exists (Y/N) |
| Having_to_Where_Selection_Conversion | QQC13 | Having to Where conversion (Y/N) |
| Estimated_Number_of_Groups | QQI2 | Estimated number of groups |
| Average_Number_Rows_per_Group | QQI3 | Average number of rows in each group |
| GroupBy_Columns | QVC3001 | Grouping columns |
| MIN_Columns | QVC3002 | MIN columns |
| MAX_Columns | QVC3003 | MAX columns |
| SUM_Columns | QVC3004 | SUM columns |
| COUNT_Columns | QVC3005 | COUNT columns |
| AVG_Columns | QVC3006 | AVG columns |
| STDDEV_Columns | QVC3007 | STDDEV columns |
| VAR_Columns | QVC3008 | VAR columns |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3030 - Materialized query tables

Displays the SQL logical view format for database monitor QQQ3030.

```

Create View QQQ3030 as
  (SELECT QQRID as Row_ID,
         QQTIME as Time_Created,
         QQJFLD as Join_Column,
         QQRDBN as Relational_Database_Name,
         QSYS as System_Name,
         QQJOB as Job_Name,
         QQUSER as Job_User,
         QQJNUM as Job_Number,
         QQI9 as Thread_ID,
         QQUCNT as Unique_Count,
         QQUDEF as User_Defined,
         QQQDTN as Unique_SubSelect_Number,
         QQQDTL as SubSelect_Nested_Level,
         QQMATN as Materialized_View_Subselect_Number,
         QQMATL as Materialized_View_Nested_Level,
         QVP15E as Materialized_View_Union_Level,
         QVP15A as Decomposed_Subselect_Number,
         QVP15B as Total_Number_Decomposed_SubSelects,
         QVP15C as Decomposed_SubSelect_Reason_Code,
         QVP15D as Starting_Decomposed_SubSelect,
         QVRCNT as Unique_Refresh_Counter,
         QQ1000 as Materialized_Query_Tables,
         QQC301 as MQT_Reason_Codes,
         QQSMINTF as Plan_Iteration_Number
   FROM   UserLib/DBMONTAb1e
  WHERE  QQRID=3030)

```

Table 84. QQQ3030 - Materialized query tables

| View Column Name | Table Column Name | Description |
|------------------|-------------------|--------------------|
| Row_ID | QQRID | Row identification |

Table 84. QQQ3030 - Materialized query tables (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|--|
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job User |
| Job_Number | QQJNUM | Job Number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Materialized_Query_Tables | QQ1000 | Materialized query tables examined and reason why used or not used: <ul style="list-style-type: none"> • 0 - The materialized query table was used • 1 - The cost to use the materialized query table, as determined by the optimizer, was higher than the cost associated with the chosen implementation. • 2 - The join specified in the materialized query was not compatible with the query. • 3 - The materialized query table had predicates that were not matched in the query. • 4 - The grouping specified in the materialized query table is not compatible with the grouping specified in the query. |

Table 84. QQQ3030 - Materialized query tables (continued)

| View Column Name | Table Column Name | Description |
|---------------------------------------|-------------------|---|
| Materialized_Query_Tables (continued) | | <ul style="list-style-type: none"> • 5 - The query specified columns that were not in the select-list of the materialized query table. • 6 - The materialized query table query contains functionality that is not supported by the query optimizer. • 7 - The materialized query table specified the DISABLE QUERY OPTIMIZATION clause. • 8 - The ordering specified in the materialized query table is not compatible with the ordering specified in the query. • 9 - The query contains functionality that is not supported by the materialized query table matching algorithm. • 10 - Materialized query tables may not be used for this query. • 11 - The refresh age of this materialized query table exceeds the duration specified by the MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option. • 12 - The commit level of the materialized query table is lower than the commit level specified for the query. • 13 - The distinct specified in the materialized query table is not compatible with the distinct specified in the query. • 14 - The FETCH FOR FIRST n ROWS clause of the materialized query table is not compatible with the query. • 15 - The QAQQINI options used to create the materialized query table are not compatible with the QAQQINI options used to run this query. • 16 - The materialized query table is not usable. • 17 - The union specified in the materialized query table is not compatible with the query. • 18 - The constants specified in the materialized query table are not compatible with host variable values specified in the query. • 19 - The Materialized query table is in check pending status. • 20 - The UDTF specified in the materialized query table was not compatible with the query. • 21 - The VALUES clause specified in the materialized query table was not compatible with the query. • 22 - The UNNEST clause specified in the materialized query table was not compatible with the query. |
| MQT_Reason_Codes | QQC301 | List of unique reason codes used by the materialized query tables (each materialized query table has a corresponding reason code associated with it) |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Database monitor view 3031 - Recursive common table expressions

Displays the SQL logical view format for database monitor QQQ3031.

```

| Create View QQQ3031 as
|   (SELECT QQRID as Row_ID,
|         QQTIME as Time_Created,
|         QQJFLD as Join_Column,
|         QQRDBN as Relational_Database_Name,
|         QQSYS as System_Name,
|         QQJOB as Job_Name,
|         QQUSER as Job_User,
|         QQJNUM as Job_Number,
|         QQI9 as Thread_ID,
|         QQUCNT as Unique_Count,
|         QQUDEF as User_Defined,
|         QQQDTN as Unique_SubSelect_Number,
|         QQQDTL as SubSelect_Nested_Level,
|         QQMATN as Materialized_View_Subselect_Number,
|         QQMATL as Materialized_View_Nested_Level,
|         QVP15E as Materialized_View_Union_Level,
|         QVP15A as Decomposed_Subselect_Number,
|         QVP15B as Total_Number_Decomposed_SubSelects,
|         QVP15C as Decomposed_SubSelect_Reason_Code,
|         QVP15D as Starting_Decomposed_SubSelect,
|         QVRCNT as Unique_Refresh_Counter,
|         QVPARPF as Parallel_Prefetch,
|         QVPARPL as Parallel_Preload,
|         QVPARD as Parallel_Degree_Requested,
|         QVPARU as Parallel_Degree_Used,
|         QVPARRC as Parallel_Degree_Reason_Code,
|         QQEPT as Estimated_Processing_Time,
|         QVCTIM as Estimated_Cumulative_Time,
|         QQREST as Estimated_Rows_Selected,
|         QQC11 as Recursive_Query_Cycle_Check,
|         QQC15 as Recursive_Query_Search_Option,
|         QQI2 as Number_of_Recursive_Values,
|         QQSMINTF as Plan_Iteration_Number
|   FROM UserLib/DBMONTabTe
|   WHERE QQRID=3031)

```

Table 85. QQQ3031 - Recursive common table expressions

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|------------------------------------|
| Row_ID | QQRID | Row identification |
| Time_Created | QQTIME | Time row was created |
| Join_Column | QQJFLD | Join column (unique per job) |
| Relational_Database_Name | QQRDBN | Relational database name |
| System_Name | QQSYS | System name |
| Job_Name | QQJOB | Job name |
| Job_User | QQUSER | Job User |
| Job_Number | QQJNUM | Job Number |
| Thread_ID | QQI9 | Thread identifier |
| Unique_Count | QQUCNT | Unique count (unique per query) |
| User_Defined | QQUDEF | User defined column |
| Unique_SubSelect_Number | QQQDTN | Unique subselect number |
| SubSelect_Nested_Level | QQQDTL | Subselect nested level |
| Materialized_View_Subselect_Number | QQMATN | Materialized view subselect number |
| Materialized_View_Nested_Level | QQMATL | Materialized view nested level |
| Materialized_View_Union_Level | QVP15E | Materialized view union level |

Table 85. QQQ3031 - Recursive common table expressions (continued)

| View Column Name | Table Column Name | Description |
|------------------------------------|-------------------|---|
| Decomposed_Subselect_Number | QVP15A | Decomposed query subselect number, unique across all decomposed subselects |
| Total_Number_Decomposed_SubSelects | QVP15B | Total number of decomposed subselects |
| Decomposed_SubSelect_Reason_Code | QVP15C | Decomposed query subselect reason code |
| Starting_Decomposed_SubSelect | QVP15D | Decomposed query subselect number for the first decomposed subselect |
| Unique_Refresh_Counter | QVRCNT | Unique refresh counter |
| Parallel_Prefetch | QVPARPF | Parallel Prefetch (Y/N) |
| Parallel_PreLoad | QVPARPL | Parallel Preload (Y/N) |
| Parallel_Degree_Requested | QVPARD | Parallel degree requested |
| Parallel_Degree_Used | QVPARU | Parallel degree used |
| Parallel_Degree_Reason_Code | QVPARRC | Reason parallel processing was limited |
| Estimated_Processing_Time | QQEPT | Estimated processing time, in seconds |
| Estimated_Cumulative_Time | QVCTIM | Estimated cumulative time, in seconds |
| Estimated_Rows_Selected | QQREST | Estimated number of rows selected |
| Recursive_Query_Cycle_Check | QQC11 | CYCLE option: <ul style="list-style-type: none"> • Y - checking for cyclic data • N - not checking for cyclic data |
| Recursive_Query_Search_Option | QQC15 | SEARCH option: <ul style="list-style-type: none"> • N - None specified • D - Depth first • B - Breadth first |
| Number_of_Recursive_Values | QQI2 | Number of values put on queue to implement recursion. Includes values necessary for CYCLE and SEARCH options. |
| Plan_Iteration_Number | QQSMINTF | AQP Plan iteration number, original optimization = 1 |

Query optimizer messages reference

See the following for query optimizer message reference:

Query optimization performance information messages

You can evaluate the structure and performance of the SQL statements in a program using informational messages. These messages are 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 could send any of the following messages when appropriate. The ampersand variables (&1, &X) are replacement variables that contain either an object name or other substitution value when you see the message in the job log. These messages provide feedback on how a query was run. In some cases, the messages indicate improvements you can make 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.

| CPI4321 - Access path built for &18 &19 | |
|--|--|
| Message Text: | Access path built for &18 &19. |
| Cause Text: | <p>A temporary access path was built to access records from member &6 of &18 &19 in library &5 for reason code &10. This process took &11 minutes and &12 seconds. The access path built contains &15 entries. The access path was built using &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"> 1 - Perform specified ordering/grouping criteria. 2 - Perform specified join criteria. 3 - Perform specified record selection to minimize I/O wait time. <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>&17.</p> <p>A key field of *MAP indicates the key field is an expression (derived field).</p> <p>The access path was built using sequence table &13 in library &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 &18 &19 in library &5 is a logical file then the access path is built over member &9 of physical file &7 in library &8.</p> <p>A file name starting with *QUERY or *N indicates the access path was built over a temporary file.</p> |
| 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 &13 in library &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> |

| CPI4322 - Access path built from keyed file &1 | |
|---|---------------------------------------|
| Message Text: | Access path built from keyed file &1. |

| CPI4322 - Access path built from keyed file &1 | |
|---|---|
| Cause Text: | <p>A temporary access path was built using the access path from member &3 of keyed file &1 in library &2 to access records from member &6 of file &4 in library &5 for reason code &10. This process took &11 minutes and &12 seconds. The access path built contains &15 entries. The reason codes and their meanings follow:</p> <ol style="list-style-type: none"> 1 - Perform specified ordering/grouping criteria. 2 - Perform specified join criteria. 3 - Perform specified record selection to minimize I/O wait time. <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>&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 &13 in library &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 &4 in library &5 is a logical file then the temporary access path is built over member &9 of physical file &7 in library &8. Creating an access path from a keyed file generally results in improved performance.</p> |
| 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 &13 in library &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> |

| CPI4323 - The query access plan has been rebuilt | |
|---|---|
| Message Text: | The query access plan has been rebuilt. |

| CPI4323 - The query access plan has been rebuilt | |
|---|---|
| Cause Text: | <p>The access plan was rebuilt for reason code &13. The reason codes and their meanings follow:</p> <p>0 - A new access plan was created.</p> <p>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 recreated, restored, or overridden to a new object.</p> <p>2 - Access plan was using a reusable Open Data Path (ODP), and the optimizer chose to use a non-reusable ODP.</p> <p>3 - Access plan was using a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP.</p> <p>4 - The number of records in member &3 of file &1 in library &2 has changed by more than 10%.</p> <p>5 - A new access path exists over member &6 of file &4 in library &5.</p> <p>6 - An access path over member &9 of file &7 in library &8 that was used for this access plan no longer exists or is no longer valid.</p> <p>7 - The query access plan had to be rebuilt because of system programming changes.</p> <p>8 - The CCSID (Coded Character Set Identifier) of the current job is different than the CCSID used in the access plan.</p> <p>9 - The value of one of the following is different in the current job: date format, date separator, time format, or time separator.</p> <p>10 - The sort sequence table specified has changed.</p> <p>11 - The number of active processors or the size or paging option of the storage pool has changed.</p> <p>12 - The system feature DB2 Symmetric Multiprocessing has either been installed or removed.</p> <p>13 - The value of the degree query attribute has changed either by the CHGSYSVAL or CHGQRYA CL commands or with the query options file &15 in library &16.</p> <p>14 - A view is either being opened by a high level language open, or is being materialized.</p> <p>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.</p> <p>16 - Query attributes have been specified from the query options file &15 in library &16.</p> <p>17 - The access plan was generated with a commitment control level that is different in the current job.</p> <p>18 - The access plan was generated with a different static cursor answer set size.</p> <p>19 - This is the first run of the query since a prepare or compile.</p> <p>20 and greater -- View the second level message text of the next message issued (CPI4351) for an explanation of these reason codes.</p> <p>If the reason code is 4, 5, 6, 20, or 21 and the file specified in the reason code explanation is a logical file, then member &12 of physical file &10 in library &11 is the file with the specified change.</p> |
| Recovery Text: | Excessive rebuilds should be avoided and may indicate an application design problem. |

| CPI4324 - Temporary file built for file &1 | |
|---|-----------------------------------|
| Message Text: | Temporary file built for file &1. |

| CPI4324 - Temporary file built for file &1 | |
|---|---|
| Cause Text: | <p>A temporary file was built for member &3 of file &1 in library &2 for reason code &4. This process took &5 minutes and &6 seconds. The temporary file was required in order for the query to be processed. The reason codes and their meanings follow:</p> <p>1 - The file is a join logical file and its join-type (JDFTVAL) does not match the join-type specified in the query.</p> <p>2 - The format specified for the logical file references more than one physical file.</p> <p>3 - The file is a complex SQL view, or nested table expression, or common table expression, or is a data change table reference that requires a temporary file.</p> <p>4 - For an update-capable query, a subselect references a field in this file which matches one of the fields being updated.</p> <p>5 - For an update-capable query, a subselect references SQL view &1, which is based on the file being updated.</p> <p>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.</p> <p>7 - The file is user-defined table function &8 in &2, and all the records were retrieved from the function. The processing time is not returned for this reason code.</p> <p>8 - The file is a partition file requiring a temporary file for processing the grouping or join.</p> |
| Recovery Text: | You may want to change the query to refer to a file that does not require a temporary file to be built. |

| CPI4325 - Temporary result file built for query | |
|--|--|
| Message Text: | Temporary result file built for query. |

| CPI4325 - Temporary result file built for query | |
|--|--|
| Cause Text: | <p>A temporary result file was created to contain the results of the query for reason code &4. This process took &5 minutes and &6 seconds. The temporary file created contains &7 records. The reason codes and their meanings follow:</p> <p>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.</p> <p>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.</p> <p>3 - The grouping and ordering fields are not compatible.</p> <p>4 - DISTINCT was specified for the query.</p> <p>5 - Set operator (UNION, EXCEPT, or INTERSECT) was specified for the query.</p> <p>6 - The query had to be implemented using a sort. More than 120 key fields specified for ordering.</p> <p>7 - The query optimizer chose to use a sort rather than an access path to order the results of the query.</p> <p>8 - Perform specified record selection to minimize I/O wait time.</p> <p>9 - The query optimizer chose to use a hashing algorithm rather than an access path to perform the grouping for the query.</p> <p>10 - The query contains a join condition that requires a temporary file.</p> <p>11 - The query optimizer creates a run-time temporary file in order to implement certain correlated group by queries.</p> <p>12 - The query contains grouping fields (GROUP BY, MIN/MAX, COUNT, etc.) and there is a read trigger on one or more of the underlying physical files in the query.</p> <p>13 - The query involves a static cursor or the SQL FETCH FIRST clause.</p> |
| Recovery Text: | For more information on why a temporary result was used, refer to "Data access methods" on page 8. |

| CPI4325 - Temporary result file built for query | |
|--|---|
| Message Text: | &12 &13 processed in join position &10. |

| CPI4325 - Temporary result file built for query | |
|--|--|
| Cause Text: | <p>Access path for member &5 of file &3 in library &4 was used to access records in member &2 of file &13 in library &1 for reason code &9. The reason codes and their meanings follow:</p> <p>1 - Perform specified record selection.</p> <p>2 - Perform specified ordering/grouping criteria.</p> <p>3 - Record selection and ordering/grouping criteria.</p> <p>4 - Perform specified join criteria.</p> <p>If file &13 in library &1 is a logical file then member &8 of physical file &6 in library &7 is the actual file in join position &10.</p> <p>A file name starting with *TEMPX for the access path indicates it is a temporary access path built over file &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: &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 &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 the "Data access methods" on page 8 for additional tips on optimizing a query's join order and index only access.</p> <p>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.</p> |

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

| CPI4326 - &12 &13 processed in join position &10 | |
|--|--|
| Cause Text: | <p>Access path for member &5 of file &3 in library &4 was used to access records in member &2 of file &13 in library &1 for reason code &9. The reason codes and their meanings follow:</p> <p>1 - Perform specified record selection.</p> <p>2 - Perform specified ordering/grouping criteria.</p> <p>3 - Record selection and ordering/grouping criteria.</p> <p>4 - Perform specified join criteria.</p> <p>If file &13 in library &1 is a logical file then member &8 of physical file &6 in library &7 is the actual file in join position &10.</p> <p>A file name starting with *TEMPX for the access path indicates it is a temporary access path built over file &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: &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 &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 the "Data access methods" on page 8 for additional tips on optimizing a query's join order and index only access.</p> <p>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.</p> |

This message provides the join position of the specified table when an index is used to access the table data. **Join position** pertains to the order in which the tables are joined.

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

| CPI4327 - File &12 &13 processed in join position &10 | |
|---|--|
| 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 the "Data access methods" on page 8 for additional tips on optimizing a query's join order.</p> |

| CPI4328 - Access path of file &3 was used by query | |
|--|---|
| Message Text: | Access path of file &3 was used by query. |
| Cause Text: | <p>Access path for member &5 of file &3 in library &4 was used to access records from member &2 of &12 &13 in library &1 for reason code &9. The reason codes and their meanings follow:</p> <p>1 - Record selection.</p> <p>2 - Ordering/grouping criteria.</p> <p>3 - Record selection and ordering/grouping criteria.</p> <p>If file &13 in library &1 is a logical file then member &8 of physical file &6 in library &7 is the actual file being accessed.</p> <p>Index only access was used for this query: &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 &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 the "Data access methods" on page 8. for additional tips on index only access.</p> |

| CPI4329 - Arrival sequence access was used for &12 &13 | |
|--|---|
| Message Text: | Arrival sequence access was used for &12 &13. |
| Cause Text: | <p>Arrival sequence access was used to select records from member &2 of file &13 in library &1.</p> <p>If file &13 in library &1 is a logical file then member &8 of physical file &6 in library &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 "Data access methods" on page 8.</p> |

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

| CPI432A - Query optimizer timed out for file &1 | |
|--|--|
| Cause Text: | <p>The query optimizer timed out before it could consider all access paths built over member &3 of file &1 in library &2.</p> <p>The list below shows the access paths considered before the optimizer timed out. If file &1 in library &2 is a logical file then the access paths specified are actually built over member &9 of physical file &7 in library &8. Following each access path name in the list is a reason code which explains how the optimizer considered the access path.</p> <p>&11.</p> <p>The reason codes and their meanings follow:</p> <p>0 - The access path was used to implement the query.</p> <p>1 - Access path was not in a valid state. The system invalidated the access path.</p> <p>2 - Access path was not in a valid state. The user requested that the access path be rebuilt.</p> <p>3 - Access path is a temporary access path (resides in library QTEMP) and was not specified as the file to be queried.</p> <p>4 - The cost to use this access path, as determined by the optimizer, was higher than the cost associated with the chosen access method.</p> <p>5 - The keys of the access path did not match the fields specified for the ordering/grouping criteria.</p> <p>6 - The keys of the access path did not match the fields specified for the join criteria.</p> <p>7 - Use of this access path would not minimize delays when reading records from the file as the user requested.</p> <p>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.</p> <p>9 - File &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.</p> <p>10 and greater - View the second level message text of the next message issued (CPI432D) for an explanation of these reason codes.</p> |
| 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> |

| CPI432B - Subselects processed as join query | |
|---|--|
| 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 | |
|---|---|
| Message Text: | All access paths were considered for file &1. |

| CPI432C - All access paths were considered for file &1 | |
|---|--|
| Cause Text: | <p>The query optimizer considered all access paths built over member &3 of file &1 in library &2.</p> <p>The list below shows the access paths considered. If file &1 in library &2 is a logical file then the access paths specified are actually built over member &9 of physical file &7 in library &8. Following each access path name in the list is a reason code which explains how the optimizer considered the access path.</p> <p>&11.</p> <p>The reason codes and their meanings follow:</p> <p>0 - The access path was used to implement the query.</p> <p>1 - Access path was not in a valid state. The system invalidated the access path.</p> <p>2 - Access path was not in a valid state. The user requested that the access path be rebuilt.</p> <p>3 - Access path is a temporary access path (resides in library QTEMP) and was not specified as the file to be queried.</p> <p>4 - The cost to use this access path, as determined by the optimizer, was higher than the cost associated with the chosen access method.</p> <p>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.</p> <p>6 - The keys of the access path did not match the fields specified for the join criteria.</p> <p>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.</p> <p>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.</p> <p>9 - File &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.</p> <p>10 and greater - View the second level message text of the next message issued (CPI432D) for an explanation of these reason codes.</p> |
| Recovery Text: | The user may want to delete any access paths no longer needed. |

| CPI432D - Additional access path reason codes were used | |
|--|--|
| Message Text: | Additional access path reason codes were used. |

| CPI432D - 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> <p>The reason codes and their meanings follow:</p> <p>10 - The user specified ignore decimal data errors on the query. This disallows the use of permanent access paths.</p> <p>11 - The access path contains static select/omit selection criteria which is not compatible with the selection in the query.</p> <p>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.</p> <p>13 - The access path cannot be used because it contains one or more keys which may be changed by the query during an insert or update.</p> <p>14 - The access path is being deleted or is being created in an uncommitted unit of work in another process.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>99 - The access path was used to gather statistics information for the query optimizer.</p> |
| Recovery Text: | See prior message CPI432A or CPI432C for more information. |

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

| CPI432E - Selection fields mapped to different attributes | |
|--|--|
| Message Text: | Selection fields mapped to different attributes. |

| CPI432E - Selection fields mapped to different attributes | |
|--|---|
| Cause Text: | <p>The data type, digits, decimal position, or length of each of the following selection fields was changed so that the field could be properly compared to the literal, host variable, or field operand associated with it. Therefore, an access path cannot be used to process that selection, since no key field has attributes that match the new attributes of the field. &1.</p> <p>The data type of the field may have been changed to match the comparison operand. For a numeric field, the number of total digits or fractional digits of the comparison operand may have exceeded that of the field.</p> |
| Recovery Text: | <p>You may want to change each comparison operand as follows:</p> <ol style="list-style-type: none"> 1 - For a literal, change the literal value so that its attributes match the field's attributes. Normally, an attributes mismatch is caused by a numeric literal that has non-significant leading or trailing zeroes. 2 - For a host variable, either change the host variable's definition to match the field's definition or define a new host variable that matches the field's definition. 3 - For a field, change the attributes of one of the fields to match the other's attributes. |

| CPI432F - Access path suggestion for file &1 | |
|---|---|
| Message Text: | Access path suggestion for file &1. |
| Cause Text: | <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 &3 of file &1 in library &2.</p> <p>In the list of key fields that follow, the query optimizer is recommending the first &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> <p>&11.</p> <p>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.</p> |
| Recovery Text: | <p>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.</p> <p>For more information, refer to "Data access methods" on page 8.</p> |

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

| CPI4330 - &6 tasks used for parallel &10 scan of file &1 | |
|---|--|
| Cause Text: | <p>&6 is the average numbers of tasks used for a &10 scan of member &3 of file &1 in library &2.</p> <p>If file &1 in library &2 is a logical file, then member &9 of physical file &7 in library &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 &5 which was limited for reason code &4. The reason code definitions are:</p> <ol style="list-style-type: none"> 1 - The *NBRTASKS parameter value was specified for the DEGREE parameter of the CHGQRYA CL command. 2 - The optimizer calculated the number of tasks which would use all of the central processing units (CPU). 3 - The optimizer calculated the number of tasks which can efficiently run in this job's share of the memory pool. 4 - The optimizer calculated the number of tasks which can efficiently run using the entire memory pool. 5 - The optimizer limited the number of tasks to equal the number of disk units which contain the file's data. <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 &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"> 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 &5. 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. 3 - Specify *MAX for the query attribute DEGREE. 4 - Increase the size of the memory pool. 5 - Use the CHGPF CL command or the SQL ALTER statement to redistribute the file's data across more disk units. |
| CPI4331 - &6 tasks used for parallel index created over file | |
| Message Text: | &6 tasks used for parallel index created over file &1. |

| CPI4331 - &6 tasks used for parallel index created over file | |
|---|--|
| Cause Text: | <p>&6 is the average numbers of tasks used for an index created over member &3 of file &1 in library &2.</p> <p>If file &1 in library &2 is a logical file, then member &9 of physical file &7 in library &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 &5 which was limited for reason code &4. The definition of reason codes are:</p> <ol style="list-style-type: none"> 1 - The *NBRTASKS parameter value was specified for the DEGREE parameter of the CHGQRYA CL command. 2 - The optimizer calculated the number of tasks which would use all of the central processing units (CPU). 3 - The optimizer calculated the number of tasks which can efficiently run in this job's share of the memory pool. 4 - The optimizer calculated the number of tasks which can efficiently run using the entire memory pool. <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"> 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 &5 to see if a performance improvement is achieved. 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. 3 - Specify *MAX for the query attribute degree. 4 - Increase the size of the memory pool. |

| CPI4332 - &1 host variables used in query | |
|--|----------------------------------|
| Message Text: | &1 host variables used in query. |

| CPI4332 - &1 host variables used in query | |
|---|--|
| Cause Text: | <p>There were &1 host variables defined for use in the query. The values used for the host variables for this open of the query follow: &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"> - DBCS data is displayed in hex format. - *N denotes a value of NULL. - *Z denotes a zero length string. - *L denotes a value too long to display in the replacement text. - *U denotes a value that could not be displayed. |
| Recovery Text: | None |

| CPI4333 - Hashing algorithm used to process join | |
|--|---|
| 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: | The hash join method is usually a good implementation choice, however, if you want to disallow the use of this method specify ALWCPYDTA(*YES). |

| CPI4334 - Query implemented as reusable ODP | |
|---|--|
| 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 follow | |
|---|--|
| Message Text: | Optimizer debug messages for hash join step &1 follow: |
| 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 "Data access methods" on page 8 for more information about hashing algorithm for join processing. |

| CPI4336 - Group processing generated | |
|---|---|
| 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 "Data access methods" on page 8 |

| CPI4337 - Temporary hash table build for hash join step &1 | |
|---|---|
| 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 "Data access methods" on page 8 for more information about hashing algorithm for join processing. |

| CPI4338 - &1 Access path(s) used for bitmap processing of file &2 | |
|--|--|
| 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 &4 of file &2 in library &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>&8</p> <p>If file &2 in library &3 is a logical file then member &7 of physical file &5 in library &6 is the actual file being accessed.</p> |
| Recovery Text: | Refer to "Data access methods" on page 8 for more information about bitmap processing. |

| CPI433A - Unable to retrieve query options file | |
|--|--|
| Message Text: | Unable to retrieve query options file. |

| CPI433A - Unable to retrieve query options file | |
|--|---|
| Cause Text: | <p>Unable to retrieve the query options from member &3 in file &2 in library &1 for reason code &4. The reason codes and their meanings follow:</p> <p>1 - Library &1 was not found.</p> <p>2 - File &2 in library &1 was not found.</p> <p>3 - The file was damaged.</p> <p>4 - The file was locked by another process which prevented successful retrieval of the query options.</p> <p>5 - File &2 and the internal query options structures are out of sync.</p> <p>6 - An unexpected error occurred while trying to retrieve the options file.</p> <p>The query options file is used by the Query Optimizer to determine how a query will be implemented.</p> |
| Recovery Text: | <p>Default query options will be used, unless one of the following actions are taken, based on the reason code above.</p> <p>1 - Either create the library (CRTLIB command) or correct the library name and then try the request again.</p> <p>2 - Either specify the library name that contains the query options file or create a duplicate object (CRTDUPOBJ command) of file &2 from library QSYS into the specified library.</p> <p>4 - Wait for lock on file &2 in library &1 to be released and try the request again.</p> <p>3, 5, or 6 - Delete query options file &2 in library &1 and then duplicate it from QSYS. If the problem still persists, report the problem (ANZPRB command).</p> |

| CPI433B - Unable to update query options file | |
|--|---|
| Message Text: | Unable to update query options file. |
| Cause Text: | <p>An error occurred while trying to update the query options from member &3, file &2, library &1 for reason code &4. The reason codes and their meanings follow:</p> <p>1 - The library &1 was not found.</p> <p>2 - The file &2 in library &1 was not found.</p> <p>3 - The parameter &5 was not found.</p> <p>4 - The value &6 for parameter &5 was not valid.</p> <p>5 - An unexpected error occurred while trying to update the options file.</p> |
| Recovery Text: | <p>Do one of the following actions based on the reason code above.</p> <p>1 - Either create the library (CRTLIB) command or correct the library name and then try the request again.</p> <p>2 - Either specify the library name that contains the query options file or create duplicate object (CRTDUPOBJ) command of QAQQINI from library QSYS into the specified library.</p> <p>3 - Either specify a valid parameter or correct the parameter name and then try the request again.</p> <p>4 - Either specify a valid parameter value or correct the parameter value and then try the request again. (WRKJOB) command.</p> |

| | |
|---|---|
| CPI433C - Library &1 not found | |
| Message Text: | Library &1 not found. |
| Cause Text: | The specified library does not exist, or the name of the library is not spelled correctly. |
| Recovery Text: | Correct the spelling of the library name, or specify the name of an existing library. Then try the request again. |

| | |
|--|---|
| CPI433D - Query options used to build the query access plan | |
| Message Text: | Query options used to build the 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 |

| | |
|---|---|
| CPI433E - User-defined function &4 found in library &1 | |
| Message Text: | User-defined function &4 found in library &1. |
| Cause Text: | Function &4 was resolved to library &1. The specific name of the function is &5. If the function is defined to use an external program, the associated program or service program is &3 in library &2. |
| Recovery Text: | Refer to the SQL programming topic collection, for more information on user-defined functions. |

| | |
|---|---|
| CPI433F - Multiple join classes used to process join | |
| Message Text: | Multiple join classes used to process join. |
| Cause Text: | Multiple join classes are used when join queries are written that have conflicting operations or cannot be implemented as a single query. 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. 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. |
| Recovery Text: | Refer to "Join optimization" on page 55 for more information about join classes. |

| | |
|---|---|
| CPI4340 - Optimizer debug messages for join class step &1 follow | |
| Message Text: | Optimizer debug messages for join class step &1 follow: |
| 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 55 for more information about join classes. |

| | |
|---|---|
| CPI4341 - Performing distributed query | |
| 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 topic collection. |

| CPI4342 - Performing distributed join for query | |
|--|--|
| 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: &1.</p> <p>The library, file and member names of each file involved in the join follow: &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 | |
|--|--|
| 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 | |
|---|--|
| Message Text: | Temporary distributed result file &3 built for query. |
| Cause Text: | <p>Temporary distributed result file &3 was created to contain the intermediate results of the query for reason code &6. The reason codes and their meanings follow:</p> <p>1 - Data from member &2 of &7 &8 in library &1 was directed to other nodes.</p> <p>2 - Data from member &2 of &7 &8 in library &1 was broadcast to all nodes.</p> <p>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.</p> <p>4 - Query contains join criteria over a distributed file and the query was processed in multiple steps.</p> <p>A library and member name of *N indicates the data comes from a query temporary distributed file.</p> <p>File &3 was built on nodes: &9.</p> <p>It was built using partitioning keys: &10.</p> <p>A partitioning key of *N indicates no partitioning keys were used when building the temporary distributed result file.</p> |

| CPI4345 - Temporary distributed result file &3 built for query | |
|---|---|
| Recovery Text: | <p>If the reason code is:</p> <p>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.</p> <p>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.</p> <p>3 - Better performance may be achieved if grouping fields are specified that match the partitioning keys.</p> <p>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.</p> <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 | |
|---|--|
| 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 | |
|--|--|
| Message Text: | Query being processed in multiple steps. |
| 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 | |
|---|---|
| Message Text: | The ODP associated with the cursor was hard closed. |

| CPI4348 - 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 &1. The reason codes and their meanings follow:</p> <p>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.</p> <p>2 - An additional wildcard was specified in the LIKE pattern on this invocation of the cursor.</p> <p>3 - SQL indicated to the query optimizer that the cursor cannot be refreshed.</p> <p>4 - The system code could not obtain a lock on the file being queried.</p> <p>5 - The length of the host variable value is too large for the the host variable as determined by the query optimizer.</p> <p>6 - The size of the ODP to be refreshed is too large.</p> <p>7 - Refresh of the local ODP of a distributed query failed.</p> <p>8 - SQL hard closed the cursor prior to the fast path refresh code.</p> |
| 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 | |
|---|--|
| Message Text: | Fast past refresh of the host variable values is not possible. |

| CPI4349 - Fast past refresh of the host variables values is not possible | |
|---|--|
| Cause Text: | <p>The Open Data Path (ODP) for this statement or cursor could not invoke the fast past refresh code for reason code &1. The reason codes and their meanings follow:</p> <p>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.</p> <p>2 - The attributes of the new host variable value are not the same as the attributes of the old host variable value.</p> <p>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.</p> <p>4 - The host variable has a data type of IGC ONLY and the the length is not even or is less than 2 bytes.</p> <p>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.</p> <p>6 - A translate table with substitution characters was used.</p> <p>7 - The host variable contains DBCS data and a CCSID translate table with substitution characters is required.</p> <p>8 - The host variable contains DBCS that is not well formed. That is, a shift-in without a shift-out or visa versa.</p> <p>9 - The host variable must be translated with a sort sequence table and the sort sequence table contains substitution characters.</p> <p>10 - The host variable contains DBCS data and must be translated with a sort sequence table that contains substitution characters.</p> <p>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.</p> |
| 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. |

| CPI434 - Member &3 was opened with fewer open options than were specified | |
|--|---|
| Message Text: | Member &3 was opened with fewer open options than were specified. |
| Cause Text: | An INSTEAD OF trigger is being used for some of the open options. However there is an additional INSTEAD OF trigger on an underlying SQL view file whose trigger actions cannot be used. An open request can support INSTEAD OF triggers from only one SQL view file. The member could not be opened with the following open options: &4. |
| Recovery Text: | When adding an INSTEAD OF trigger, specify trigger actions for all of the requested open options. |

| CPI434E - Query could not be run using SQE | |
|---|---|
| Message Text: | Query could not be run using SQE. |
| Cause Text: | <p>The query was run using CQE (Current Query Engine). The query could not be run using SQE (SQL Query Engine) for reason code &1. The reason codes and their meanings follow:</p> <p>1 -- Sort sequence table &2 in library &3 is an ICU (International Components of Unicode) sort sequence table that is not supported by SQE.</p> |
| Recovery Text: | Recovery for reason code 1: To run the query using SQE, specify a version of the ICU sort sequence table that is &4 or later. |

| CPI4350 - Materialized query tables were considered for optimization | |
|---|---|
| Message Text: | Materialized query tables were considered for optimization. |
| Cause Text: | <p>The query optimizer considered usage of materialized query tables for this query.</p> <p>Following each materialized query table name in the list is a reason code which explains why the materialized query table was not used. A reason code of 0 indicates that the materialized query table was used to implement the query.</p> <p>The reason codes and their meanings follow:</p> <p>1 - The cost to use the materialized query table, as determined by the optimizer, was higher than the cost associated with the chosen implementation.</p> <p>2 - The join specified in the materialized query was not compatible with the query.</p> <p>3 - The materialized query table had predicates that were not matched in the query.</p> <p>4 - The grouping or distinct specified in the materialized query table is not compatible with the grouping or distinct specified in the query.</p> <p>5 - The query specified columns that were not in the select-list of the materialized query table.</p> <p>6 - The materialized query table query contains functionality that is not supported by the query optimizer.</p> <p>7 - The materialized query table specified the DISABLE QUERY OPTIMIZATION clause.</p> <p>8 - The ordering specified in the materialized query table is not compatible with the ordering specified in the query.</p> <p>9 - The query contains functionality that is not supported by the materialized query table matching algorithm.</p> <p>10 - Materialized query tables may not be used for this query.</p> <p>11 - The refresh age of this materialized query table exceeds the duration specified by the MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option.</p> <p>12 - The commit level of the materialized query table is lower than the commit level specified for the query.</p> <p>14 - The FETCH FOR FIRST n ROWS clause of the materialized query table is not compatible with the query.</p> <p>15 - The QAQQINI options used to create the materialized query table are not compatible with the QAQQINI options used to run this query.</p> <p>16 - The materialized query table is not usable.</p> <p>17 - The UNION specified in the materialized query table is not compatible with the query.</p> <p>18 - The constants specified in the materialized query table are not compatible with host variable values specified in the query.</p> <p>19 - The materialized query table is in Check Pending status and cannot be used.</p> <p>20 - The UDTF specified in the materialized query table is not compatible with UDTF in the query.</p> <p>21 - The Values clause specified in the materialized query table is not compatible with Values specified in the query.</p> |
| Recovery Text: | The user may want to delete any materialized query tables that are no longer needed. |

| CPI4351 - Additional reason codes for query access plan has been rebuilt | |
|---|--|
| Message Text: | Additional reason codes for query access plan has been rebuilt. |
| Cause Text: | <p>Message CPI4323 was issued immediately before this message. Because of message length restrictions, some of the reason codes used by message CPI4323 are explained below rather than in that message. The CPI4323 message was issued for reason code &13. The additional reason codes and their meaning follow:</p> <p>20 - Referential or check constraints for member &19 of file &17 in library &18 have changed since the access plan was generated.</p> <p>21 - Materialized query tables for member &22 of file &20 in library &21 have changed since the access plan was generated. If the file is *N then the file name is not available.</p> <p>22 - The value of a host variable changed and the access plan is no longer valid.</p> <p>23 - Adaptive Query Processing (AQP) determined that a new access plan is needed.</p> |
| Recovery Text: | See the prior message CPI4323 for more information. |

| CPI436A - Database monitor started for job &1, monitor ID &2 | |
|---|--|
| Message Text: | Database monitor started for job &1, monitor ID &2. |
| Cause Text: | <p>The database monitor was started for job &1. The system generated monitor ID for this database monitor is &2.</p> <p>If multiple monitors have been started using the same generic job name, the monitor ID is needed to uniquely identify which monitor is to be ended with the ENDDDBMON command.</p> |
| Recovery Text: | If multiple monitors have been started using the same generic job name, remember the monitor ID. The monitor ID will be required when using the ENDDDBMON command to end this specific monitor. |

Query optimization performance information messages and open data paths

Several of the following SQL runtime 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 run, in many cases, the database manager saves the associated ODPs of the SQL operations. These ODPs are then reused the next time the statement is run. For example, an SQL CLOSE statement could close the SQL cursor, but leave the ODP available to use again the next time the cursor is opened. This technique 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.

| SQL7910 - All SQL cursors closed | |
|---|---------------------|
| Message Text: | SQL cursors closed. |

| SQL7910 - All 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. |
| Recovery Text: | <p>To keep cursors, ODPs, prepared statements, and locks available after the completion of a program, use the CLOSQLCSR precompile parameter.</p> <p>-- The *ENDJOB option will allow the user to keep the SQL resources active for the duration of the job.</p> <p>-- 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.</p> <p>-- 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.</p> <p>-- The *ENDMOD option causes all SQL resources to only be accessible by the same invocation of the module.</p> <p>-- 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.</p> |

| SQL7911 - ODP reused | |
|-----------------------|---|
| 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 |

| SQL7912 - ODP created | |
|-----------------------|---|
| Message Text: | ODP created. |
| Cause Text: | <p>An Open Data Path (ODP) has been created. No reusable ODP could be found. This occurs in the following cases:</p> <p>-- This is the first time the statement has been run.</p> <p>-- A RCLRSC has been issued since the last run of this statement.</p> <p>-- The last run of the statement caused the ODP to be deleted.</p> <p>-- If this is an OPEN statement, the last CLOSE of this cursor caused the ODP to be deleted.</p> <p>-- The Application Server (AS) has been changed by a CONNECT statement.</p> |
| 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. |

The first time that the statement is run or the cursor is opened for a process, an ODP must always be created. However, if this message appears on every statement run or cursor open, use the tips recommended in “Retaining cursor positions for non-ILE program calls” on page 234 in your application.

| SQL7913 - ODP deleted | |
|------------------------------|--|
| 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. |

| SQL7914 - ODP not deleted | |
|----------------------------------|--|
| 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 |

| SQL7915 - Access plan for SQL statement has been built | |
|---|---|
| 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: -- The program has been restored from a different release and this is the first time this statement has been run. -- All the files required for the statement did not exist at precompile time, and this is the first time this statement has been run. -- The program was precompiled using SQL naming mode, and the program owner has changed since the last time the program was called. |
| Recovery Text: | This is normal processing for SQL. Once the access plan is built, it will be used on subsequent runs of the statement. |

| SQL7916 - Blocking used for query | |
|--|--|
| 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. |

| SQL7917 - Access plan not updated | |
|--|--|
| 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. |

| | |
|--|---|
| SQL7917 - Access plan not 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. |

| | |
|---------------------------------------|---|
| SQL7918 - Reusable ODP deleted | |
| Message Text: | Reusable ODP deleted. Reason code &1. |
| Cause Text: | <p>An existing Open Data Path (ODP) was found for this statement, but it could not be reused for reason &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> <p>1 -- Commitment control isolation level is not compatible.</p> <p>2 -- The statement contains SQL special register USER, CURRENT DEBUG MODE, CURRENT DECFLOAT ROUNDING MODE, or CURRENT TIMEZONE, and the value for one of these registers has changed.</p> <p>3 -- The PATH used to locate an SQL function has changed.</p> <p>4 -- The job default CCSID has changed.</p> <p>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.</p> <p>6 -- The file, library, or member for the original ODP was changed with an override.</p> <p>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.</p> <p>8 -- An OVRDBF or DLTOVR command has been issued, causing different override options, such as different SEQONLY or WAITRCD values.</p> <p>9 -- An error occurred when attempting to verify the statement override information is compatible with the reusable ODP information.</p> <p>10 -- The query optimizer has determined the ODP cannot be reused.</p> <p>11 -- The client application requested not to reuse ODPs.</p> |
| 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. |

| | |
|---|---|
| SQL7919 - Data conversion required on FETCH or embedded SELECT | |
| Message Text: | Data conversion required on FETCH or embedded SELECT. |

| SQL7919 - Data conversion required on FETCH or embedded SELECT | |
|--|--|
| Cause Text: | <p>Host variable &2 requires conversion. The data retrieved for the FETCH or embedded SELECT statement can not 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 &1.</p> <p>-- Reason 1 - host variable &2 is a character or graphic string of a different length than the value being retrieved.</p> <p>-- Reason 2 - host variable &2 is a numeric type that is different than the type of the value being retrieved.</p> <p>-- Reason 3 - host variable &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.</p> <p>-- Reason 4 - host variable &2 is a variable length string and the value being retrieved is not.</p> <p>-- Reason 5 - host variable &2 is not a variable length string and the value being retrieved is.</p> <p>-- Reason 6 - host variable &2 is a variable length string whose maximum length is different than the maximum length of the variable length value being retrieved.</p> <p>-- Reason 7 - a data conversion was required on the mapping of the value being retrieved to host variable &2, such as a CCSID conversion.</p> <p>-- Reason 8 - a DRDA connection was used to get the value being retrieved into host variable &2. The value being retrieved is either null capable or varying-length, is contained in a partial row, or is a derived expression.</p> <p>-- Reason 10 - the length of host variable &2 is too short to hold a TIME or TIMESTAMP value being retrieved.</p> <p>-- Reason 11 - host variable &2 is of type DATE, TIME or TIMESTAMP, and the value being retrieved is a character string.</p> <p>-- Reason 12 - too many host variables were specified and records are blocked. Host variable &2 does not have a corresponding column returned from the query.</p> <p>-- 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.</p> <p>-- Reason 14 - a LOB Locator was used and the commitment control level of the process was not *ALL.</p> |
| Recovery Text: | To get better performance, attempt to use host variables of the same type and length as their corresponding result columns. |

| SQL7939 - Data conversion required on INSERT or UPDATE | |
|--|---|
| Message Text: | Data conversion required on INSERT or UPDATE. |

| SQL7939 - Data conversion required on INSERT or UPDATE | |
|---|---|
| Cause Text: | <p>The INSERT or UPDATE values can not 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 &1.</p> <p>-- Reason 1 is that the INSERT or UPDATE value is a character or graphic string of a different length than column &2.</p> <p>-- Reason 2 is that the INSERT or UPDATE value is a numeric type that is different than the type of column &2.</p> <p>-- Reason 3 is that the INSERT or UPDATE value is a variable length string and column &2 is not.</p> <p>-- Reason 4 is that the INSERT or UPDATE value is not a variable length string and column &2 is.</p> <p>-- 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 &2.</p> <p>-- Reason 6 is that a data conversion was required on the mapping of the INSERT or UPDATE value to column &2, such as a CCSID conversion.</p> <p>-- Reason 7 is that the INSERT or UPDATE value is a character string and column &2 is of type DATE, TIME, or TIMESTAMP.</p> <p>-- Reason 8 is that the target table of the INSERT is not a SQL table.</p> |
| Recovery Text: | To get better performance, try to use values of the same type and length as their corresponding columns. |

PRTSQLINF message reference

The following messages are returned from **PRTSQLINF**.

| SQL400A - Temporary distributed result file &1 was created to contain join result | |
|--|--|
| 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. |
| Recovery Text: | For more information about processing of distributed files, refer to the Distributed database programming topic collection. |

| SQL400B - Temporary distributed result file &1 was created to contain join result | |
|--|--|
| 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 topic collection. |

| SQL400C - Optimizer debug messages for distributed query step &1 and &2 follow | |
|---|--|
| Message Text: | Optimizer debug messages for distributed query step &1 of &2 follow: |

| SQL400C - 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 topic collection. |

| SQL400D - GROUP BY processing generated | |
|--|--|
| 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 collection. |

| SQL400E - Temporary distributed result file &1 was created while processing distributed subquery | |
|---|---|
| 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. |
| Recovery Text: | 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. For more information about processing of distributed files, refer to the Distributed database programming topic collection. |

| SQL4001 - Temporary result created | |
|---|--|
| 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"> -- The table is a join logical file and its join type (JDFTVAL) does not match the join-type specified in the query. -- The format specified for the logical file refers to more than one physical table. -- The table is a complex SQL view requiring a temporary table to contain the results of the SQL view. -- 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. |
| Recovery Text: | Performance may be improved if the query can be changed to avoid temporary results. |

| SQL4002 - Reusable ODP sort used | |
|---|-------------------------|
| Message Text: | Reusable ODP sort used. |

| SQL4002 - 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"> -- 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. -- The grouping and ordering columns are not compatible. -- DISTINCT was specified for the query. -- UNION was specified for the query. -- 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. -- The query optimizer chose to use a sort rather than an index to order the results of the query. |
| Recovery Text: | A reusable ODP generally results in improved performance when compared to a non-reusable ODP. |

| SQL4003 - UNION | |
|-----------------------|--|
| Message Text: | UNION, EXCEPT, or INTERSECT. |
| 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 | |
|-----------------------|---|
| 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 | |
|--|---|
| 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 | |
|---|--------------------------------------|
| Message Text: | All indexes considered for table &1. |

| SQL4006 - 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 | |
|--|--|
| Message Text: | Query implementation for join position &1 table &2. |
| 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 55 for more information about join optimization and tips to influence join order. |

| SQL4008 - Index &1 used for table &2 | |
|--------------------------------------|--|
| 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"> -- Row selection. -- Join criteria. -- Ordering/grouping criteria. -- Row selection and ordering/grouping criteria. <p>The table number refers to the relative position of this table in the query.</p> <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 | |
|--------------------------------------|--|
| 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"> -- Perform specified ordering/grouping criteria. -- Perform specified join criteria. <p>The table number refers to the relative position of this table in the query.</p> |
| Recovery Text: | 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. 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. |

| SQL401A - Processing grouping criteria for query containing a distributed table | |
|---|--|
| Message Text: | Processing grouping criteria for query containing a distributed table. |

| SQL401A - Processing grouping criteria for query containing a distributed table | |
|--|--|
| 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 topic collection. |

| SQL401B - Temporary distributed result table &1 was created while processing grouping criteria | |
|---|---|
| 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 topic collection. |

| SQL401C - Performing distributed join for query | |
|--|--|
| 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 topic collection. |

| SQL401D - Temporary distributed result table &1 was created because table &2 was directed | |
|--|---|
| 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 topic collection. |

| SQL401E - Temporary distributed result table &1 was created because table &2 was broadcast | |
|---|---|
| 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. |
| Recovery Text: | 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. For more information about processing of distributed tables, refer to the Distributed database programming topic collection. |

| SQL401F - Table &1 used in distributed join | |
|--|------------------------------------|
| Message Text: | Table &1 used in distributed join. |

| SQL401F - 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 topic collection. |

| SQL4010 - Table scan access for table &1 | |
|--|--|
| 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 | |
|---|--|
| Message Text: | Index scan-key row positioning used on table &1. |
| Cause Text: | 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. The table number refers to the relative position of this table in the query. |
| Recovery Text: | Refer to "Data access methods" on page 8 for more information about index scan-key row positioning. |

| SQL4012 - Index created from index &1 for table &2 | |
|--|---|
| Message Text: | Index created from index &1 for table &2. |
| Cause Text: | A temporary index was created using the specified index to access rows from the queried table for one of the following reasons: -- Perform specified ordering/grouping criteria. -- Perform specified join criteria. The table number refers to the relative position of this table in the query. |
| Recovery Text: | 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. |

| SQL4013 - Access plan has not been built | |
|--|--|
| Message Text: | Access plan has not been built. |
| Cause Text: | An access plan was not created for this query. Possible reasons may include: -- Tables were not found when the program was created. -- The query was complex and required a temporary result table. -- Dynamic SQL was specified. |
| 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 | |
|--|--|
| 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 on index scan-key row positioning.</p> |

| SQL4015 - From-column &1.&2, to-column &3.&4, join operator &5, join predicate &6 | |
|---|--|
| 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> <p>-- &1: The join 'from table' number. The table number refers to the relative position of this table in the query.</p> <p>-- &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).</p> <p>-- &3: The join 'to table' number. The table number refers to the relative position of this table in the query.</p> <p>-- &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).</p> <p>-- &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).</p> <p>-- &6: The join predicate number. Identifies the join predicate within this set of join pairs.</p> |
| Recovery Text: | Refer to "Join optimization" on page 55 for more information about joins. |

| SQL4016 - Subselects processed as join query | |
|--|---|
| 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 | |
|--|---|
| Message Text: | Host variables implemented as reusable ODP. |

| SQL4017 - 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 | |
|--|---|
| 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 | |
|--|--|
| 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 | |
|--|---|
| Message Text: | Hashing algorithm used to process join. |
| Cause Text: | The hash join algorithm is typically used for longer running join queries. The original query will be subdivided into hash join steps. 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 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. |
| Recovery Text: | The hash join method is usually a good implementation choice, however, if you want to disallow the use of this method specify ALWCPYDTA(*YES). Refer to the &qryopt. for more information on hashing algorithm for join processing. |

| SQL402B - Table &1 used in hash join step &2 | |
|--|---|
| Message Text: | Table &1 used in hash join step &2. |
| Cause Text: | 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. If there are two or more of these messages for the same hash join step, then that step is a nested loop join. 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. |
| Recovery Text: | Refer to "Data access methods" on page 8 for more information about hashing. |

| SQL402C - Temporary table created for hash join results | |
|---|--|
| Message Text: | Temporary table created for hash join results. |

| SQL402C - Temporary table created for hash join results | |
|---|--|
| Cause Text: | 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: GROUP BY or summary functions ORDER BY DISTINCT Expression containing columns from more than one table Complex row selection involving columns from more than one table |
| Recovery Text: | Refer to "Data access methods" on page 8 for more information about the hashing algorithm for join processing. |

| SQL402D - Query attributes overridden from query options file &2 in library &1 | |
|--|---|
| 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 | |
|--|--|
| 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 | |
|--|---|
| 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 | |
|---|--|
| 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 | |
|--|---|
| 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"> -- The query attribute degree was specified with an option of *IO or *ANY for the application process. -- 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. <p>For more information about parallel table prefetch, refer to "Data access methods" on page 8.</p> |

SQL4024 - Parallel index preload access method used

| | |
|-----------------------|---|
| 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">-- The query attribute degree was specified with an option of *IO or *ANY for the application process.-- 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. <p>For more information about parallel table prefetch, refer to "Data access methods" on page 8.</p> |

SQL4025 - Parallel table preload access method used

| | |
|-----------------------|---|
| Message Text: | Parallel table preload access method used. |
| 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">-- The query attribute degree must have been specified with an option of *IO or *ANY for the application process.-- 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. <p>For more information about parallel table prefetch, refer to "Data access methods" on page 8.</p> |

SQL4026 - Index only access used on table number &1

| | |
|-----------------------|--|
| Message Text: | Index only access used on table number &1. |
| Cause Text: | 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. The table number refers to the relative position of this table in the query. |
| Recovery Text: | Refer to "Data access methods" on page 8 for more information about index only access. |

SQL4027 - Access plan was saved with DB2 Symmetric Multiprocessing installed on the system

| | |
|-----------------------|--|
| Message Text: | Access plan was saved with DB2 Symmetric Multiprocessing installed on the system. |
| Cause Text: | The access plan saved was created while the system feature DB2 Symmetric Multiprocessing was installed on the system. The access plan may have been influenced by the presence of this system feature. Having this system feature installed may cause the implementation of the query to change. |
| Recovery Text: | For more information about how the system feature DB2 Symmetric Multiprocessing can influence a query, refer to the "Controlling parallel processing for queries" on page 184 |

| SQL4028 - The query contains a distributed table | |
|---|--|
| Message Text: | The query contains a distributed table. |
| Cause Text: | 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. 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 topic collection. |

| SQL4029 - Hashing algorithm used to process the grouping | |
|---|---|
| Message Text: | Hashing algorithm used to process the grouping. |
| Cause Text: | The grouping specified within the query was implemented with a hashing algorithm. |
| Recovery Text: | 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 ALWCOPYDTA(*YES). Refer to "Data access methods" on page 8 for more information about the hashing algorithm. |

| SQL4030 - &1 tasks specified for parallel scan on table &2 | |
|---|--|
| Message Text: | &1 tasks specified for parallel scan on table &2. |
| Cause Text: | The query optimizer has calculated the optimal number of tasks for this query based on the query attribute degree. The table number refers to the relative position of this table in the query. |
| Recovery Text: | 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. Refer to "Data access methods" on page 8 for more information about parallel scan. |

| SQL4031 - &1 tasks specified for parallel index create over table &2 | |
|---|--|
| Message Text: | &1 tasks specified for parallel index create over table &2. |
| Cause Text: | The query optimizer has calculated the optimal number of tasks for this query based on the query attribute degree. The table number refers to the relative position of this table in the query. |
| Recovery Text: | 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. Refer to "Data access methods" on page 8 for more information about parallel index create. |

| SQL4032 - Index &1 used for bitmap processing of table &2 | |
|--|---|
| Message Text: | Index &1 used for bitmap processing of table &2. |
| Cause Text: | 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. 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. The table number refers to the relative position of this table in the query. |
| Recovery Text: | The query can be run in debug mode (STRDBG) to determine more specific information. Also, refer to "Data access methods" on page 8 for more information about bitmap processing. |

| | |
|---|---|
| SQL4033 - &1 tasks specified for parallel bitmap create using &2 | |
| 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: | 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. Refer to "Data access methods" on page 8 for more information about parallel scan. |

| | |
|---|---|
| SQL4034 - Multiple join classes used to process join | |
| Message Text: | Multiple join classes used to process join. |
| Cause Text: | Multiple join classes are used when join queries are written that have conflicting operations or cannot be implemented as a single query. Each join class will be optimized and processed as a separate step of the query with the results written out to a temporary table. 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. |
| Recovery Text: | Refer to "Join optimization" on page 55 for more information about join classes. |

| | |
|---|--|
| SQL4035 - Table &1 used in join class &2 | |
| Message Text: | Table &1 used in join class &2. |
| Cause Text: | 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. 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. 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. |
| Recovery Text: | Refer to "Join optimization" on page 55 for more information about join classes. |

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

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

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

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

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

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

- | Intellectual Property Licensing
- | Legal and Intellectual Property Law
- | IBM Japan, Ltd.
- | 3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

| © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This Database performance and query optimization publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

- | IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business
- | Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be
- | trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at
- | Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Other company, product, or service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.



Printed in USA