IBM Systems - iSeries

DB2 Universal Database for iSeries Database Performance and Query Optimization

Version 5 Release 4

IBM Systems - iSeries

DB2 Universal Database for iSeries Database Performance and Query Optimization

Version 5 Release 4

Note

Before using this information and the product it supports, read the information in "Notices," on page 321.

Sixth Edition (February 2006)

This edition applies to version 5, release 4, modification 0 of IBM i5/OS (product number 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

Contents

Performance and query optimization

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

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

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: Read the "Code license and disclaimer information" on page 320 for important legal information.

What's new for V5R4

The following information was added or updated in this release of the information:
- "Recursive query optimization" on page 72
- "Viewing the plan cache with iSeries Navigator" on page 84
- "Query optimizer index advisor" on page 110
- Query resource governor
- "Encoded vector index symbol table scan" on page 16
- Queue data access method
- "Using iSeries Navigator with detailed monitors" on page 107
- QAQQINI updates
- Set Current Degree
- Visual Explain updates
- "Display information with Database Health Center" on page 140
- New database monitor format

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.

Printable PDF

Use this to 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 7303 KB).

Other information

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

Saving PDF files

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

1. Right-click the PDF in your browser (right-click the link above).
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

Query Engine Overview

DB2® UDB for iSeries provides two query engines to process queries: the Classic Query Engine (CQE) and the SQL Query Engine (SQE).

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

Along with the new query engine, several more components were created and other existing components were updated. Additionally, new data access methods were created for SQE.

Related information

Embedded SQL programming

SQL programming

Query (QQQQRY) API

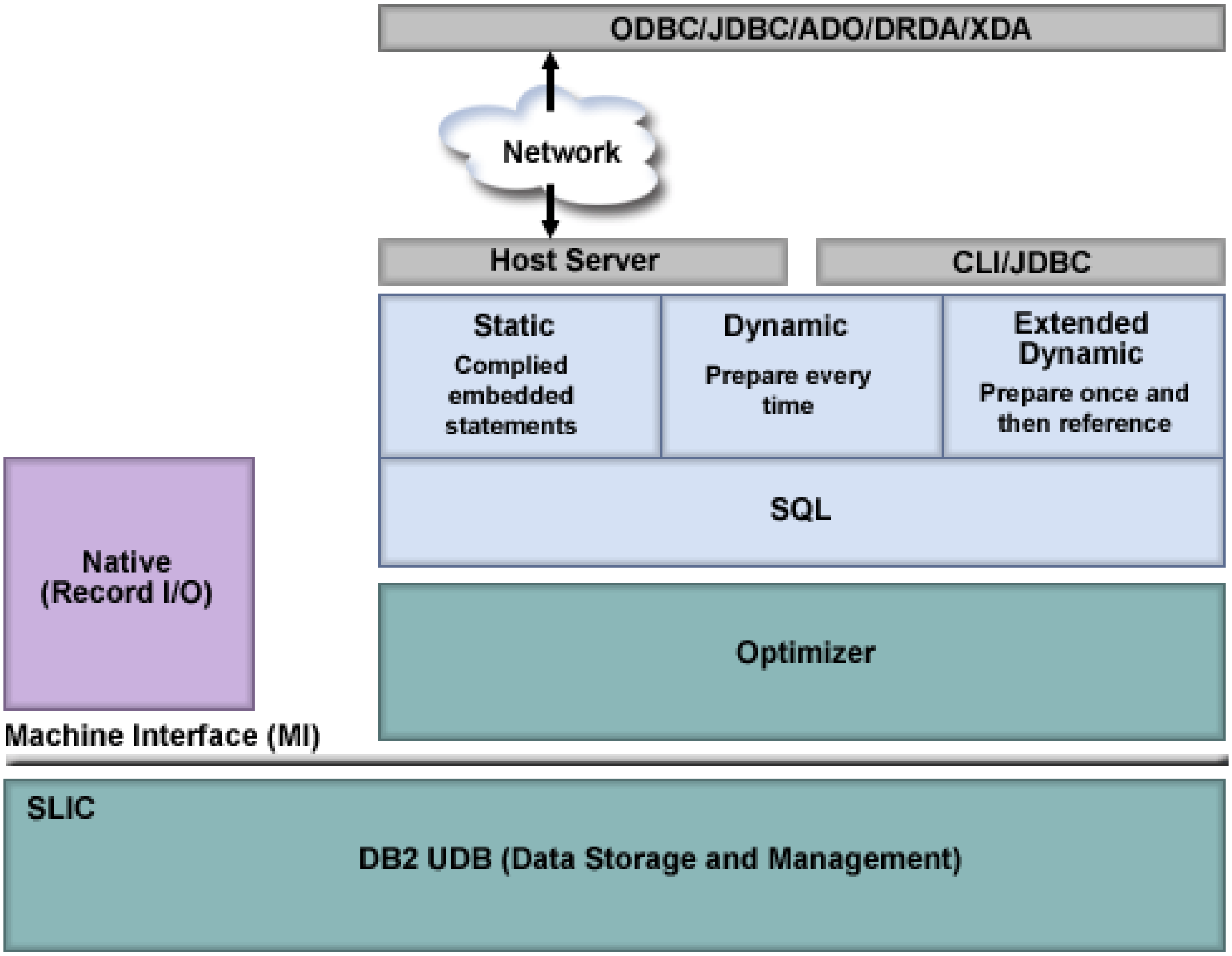
Open Query File (OPNQRYF) command

Run SQL Statements (RUNSQLSTM) command

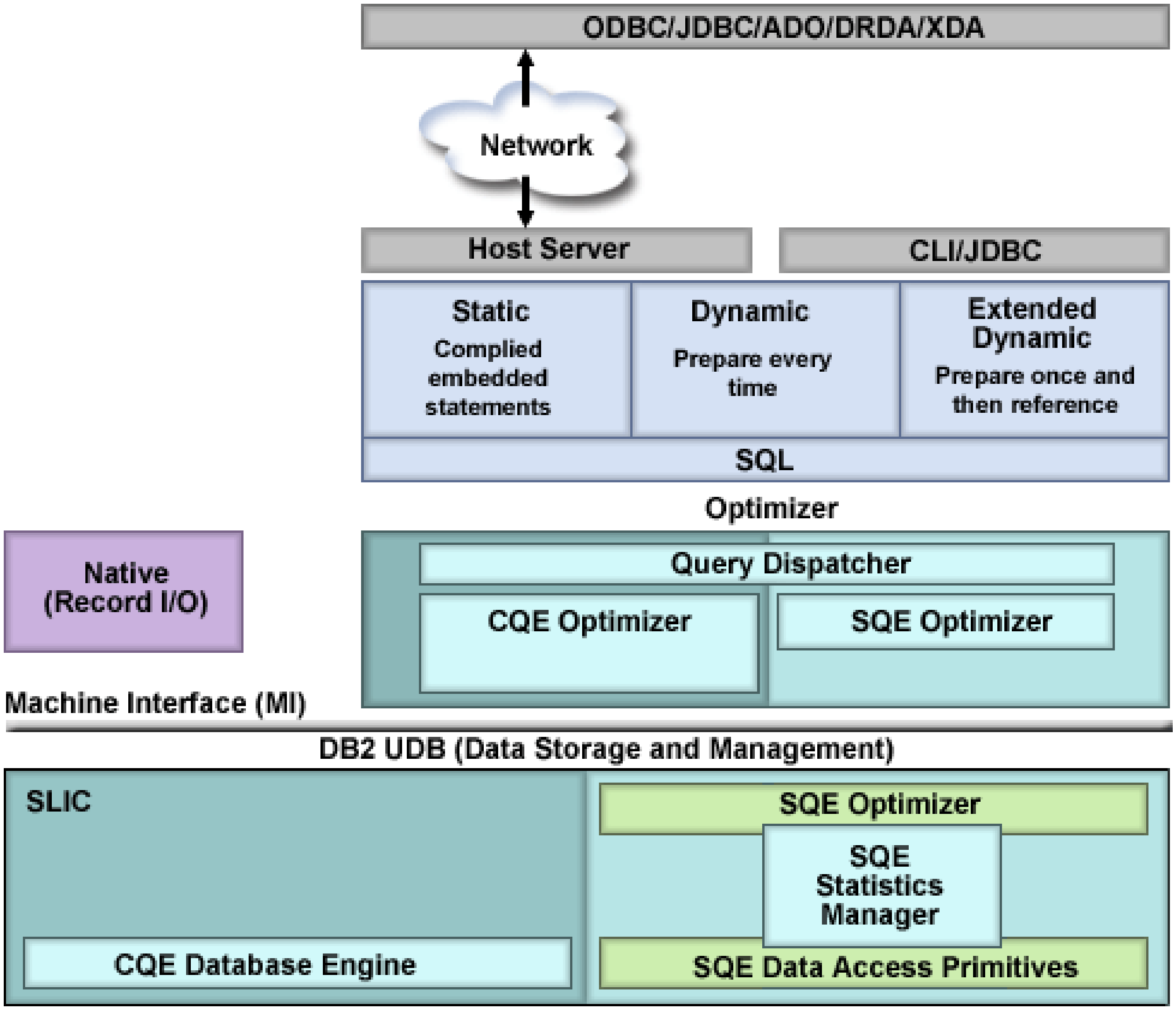
SQE and CQE Engines

To fully understand the implementation of query management and processing in DB2 UDB for iSeries on i5/OS™ V5R2 and subsequent releases, it is important to see how the queries were implemented in releases of i5/OS previous to V5R2.

The figure below shows a high-level overview of the architecture of DB2 UDB for iSeries before i5/OS V5R2. The optimizer and database engine are implemented at different layers of the operating system. The interaction between the optimizer and the database engine occurs across the Machine Interface (MI).



The figure below shows an overview of the DB2 UDB for iSeries architecture on i5/OS V5R3 and where each SQE component fits. The functional separation of each SQE component is clearly evident. In line with design objectives, this division of responsibility enables IBM® to more easily deliver functional enhancements to the individual components of SQE, as and when required. Notice that most of the SQE Optimizer components are implemented below the MI. This translates into enhanced performance efficiency.



Query Dispatcher

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

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

- INSERT WITH VALUES statement or the target of an INSERT with subselect statement
- NLSS or CCSID translation between columns
- Lateral correlation
- Logical files
- Datalink columns
- Tables with Read Triggers
- User-defined table functions
- Read-only queries with more than 1000 dataspaces or updateable queries with more than 256 dataspaces.
- DB2 Multisystem tables

- non-SQL queries, for example the QQQQry API, Query/400, or OPNQRYF

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

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

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

Related reference

"MQT supported function" on page 64
Although a MQT can contain almost any query, the optimizer only supports a limited set of query functions when matching MQTs to user specified queries. The user specified query and the MQT query must both be supported by the SQE optimizer.

Statistics Manager

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

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

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

Cardinality of values
: The number of unique or distinct occurrences of a specific value in a single column or multiple columns of a table.

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

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

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

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

The Statistics Manager uses a hybrid approach to manage database statistics. The majority of this information can be obtained from existing indexes. In cases where the required statistics cannot be gathered from existing indexes, statistical information is constructed of single columns of a table and stored internally as part of the table. By default, this information is collected automatically by the system, but you can manually control the collection of statistics. Unlike indexes, however, statistics are not maintained immediately as data in the tables change.

Related reference

"Collecting statistics with the Statistics Manager" on page 135
As stated earlier, the collection of statistics is handled by a separate component called the Statistics Manager. Statistical information can be used by the query optimizer to determine the best access plan for a query. Since the query optimizer bases its choice of access plan on the statistical information found in the table, it is important that this information be current.

Plan Cache

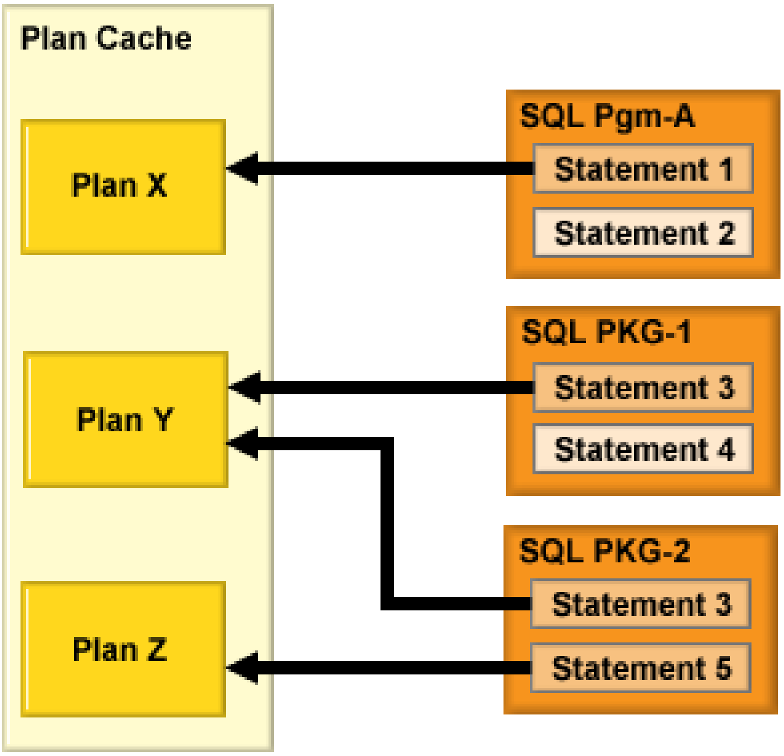
The Plan Cache is a repository that contains the access plans for queries that were optimized by SQE.

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

Once an access plan is created, it is available for use by all users and all queries, regardless of where the query originates. Furthermore, when an access plan is tuned, when creating an index for example, all queries can benefit from this updated access plan. This eliminates the need to reoptimize the query, resulting in greater efficiency.

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



As shown above, the Plan Cache is interrogated each time a query is executed in order to determine if a valid access plan exists that satisfies the requirements of the query. If a valid access plan is found, it is used to implement the query. Otherwise a new access plan is created and stored in the Plan Cache for future use. The Plan Cache is automatically updated with new query access plans as they are created, or is updated for an existing plan (the next time the query is run) when new statistics or indexes become available. 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 Megabytes (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 executables and temporary storage containers (hash tables, sorts, temporary indexes, and so on) used to run the query. All systems are currently configured with the same size Plan Cache, regardless of the server size or the hardware configuration.

When the Plan Cache exceeds its designated size, a background task is automatically scheduled to remove plans from the Plan Cache. Access plans are deleted based upon the age of the access plan, how frequently it is being used and how much cumulative resources (CPU/IO) were consumed by the runs of the query. The total number of access plans stored in the Plan Cache depends largely upon the complexity of the SQL statements that are being executed. In certain test environments, there have been typically 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 can be maintained for a single SQL statement. Although the SQL statement itself is the primary hash key to the Plan Cache, different environmental settings can cause different access plans to be stored in the Plan Cache. Examples of these environmental settings include:

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

- Different ALWCPYDTA settings

Currently, the Plan Cache can maintain a maximum of 3 different access plans for the same SQL statement. As new access plans are created for the same SQL statement, 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 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 172
Some complex queries can perform better by using a sort or hashing method to evaluate the query instead of using or creating an index.

"Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command" on page 115
You can modify different types of attributes of the queries that you will execute during a certain job with the Change Query Attributes (CHGQRYA) CL command, or by using the iSeries Navigator Change Query Attributes interface.

"Viewing the plan cache with iSeries Navigator" on page 84
The Plan Cache contains a wealth of information about the SQE queries being run through the database. Its contents are viewable through the iSeries Navigator GUI interface.

Data access on DB2 UDB for iSeries: data access paths and 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 queries
- The executable instructions or operations to retrieve and transform the data into usable information

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

Permanent objects and access methods

The database objects and access methods used by the query engine can be broken down into three basic types of operations that are used to manipulate the permanent and temporary objects -- Create, Scan, and Probe.

The following table lists each object and the access methods that can be performed against that object. The symbols shown in the table are the icons used by Visual Explain.

Table 1. Permanent object's 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 will determine the most efficient way to extract the data from the table in order to satisfy the query. This may include scanning or probing the table or using an index to extract the data.

Visual explain icon:

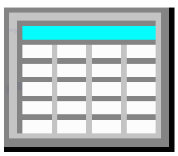


Table scan:

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

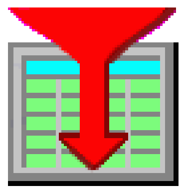
A table scan operation requests large I/Os to bring as many rows as possible into main memory for processing. It also asynchronously pre-fetches the data to make sure that the table scan operation is never waiting for rows to be paged into memory. Table scan however, has a disadvantage in it has to process all of the rows in order to satisfy the query. The scan operation itself is very efficient if it does not need to perform the I/O synchronously.

Table 2. Table scan attributes

Data access method	**Table scan**
Description	Reads all of the rows from the table and applies the selection criteria to each of the rows within the table. The rows in the table are processed in no guaranteed order, but typically they are processed sequentially.
Advantages	• 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	• 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 will be selected. You can reorganize the table to remove deleted rows.
Likely to be used	• When expecting a large number of 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

Example SQL statement	`SELECT * FROM Employee`
	`WHERE WorkDept BETWEEN 'A01'AND 'E01'`
	`OPTIMIZE FOR ALL ROWS`

Table 2. Table scan attributes (continued)

Data access method	Table scan
Messages indicating use	• 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 46
DB2 Universal Database for iSeries provides a **nested loop** join method. For this method, the processing of the tables in the join are ordered. This order is called the **join order**. The first table in the final join order is called the **primary table**. The other tables are called **secondary tables**. Each join table position is called a **dial**.

Table probe:

A table probe operation is used to retrieve a specific row from a table based upon its row number. The row number is provided to the table probe access method by some other operation that generates a row number for the table.

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

Table 3. Table probe attributes

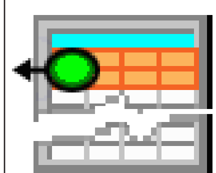
Data access method	Table probe
Description	Reads a single row from the table based upon a specific row number. A random I/O is performed against the table to extract the row.
Advantages	• Requests smaller I/Os to prevent paging rows into memory that are not needed • Can be used in conjunction with any 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 a large number of rows are selected

Table 3. Table probe attributes (continued)

Data access method	Table probe
Likely to be used	• When row numbers (either from indexes or temporary row number lists) are being used, but data from the underlying table rows are required for further processing of the query • When processing any remaining selection or projection of the values

Example SQL statement	`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`

Messages indicating use	There is no specific message that indicates the use of a table probe. The messages in this example illustrate the use of a data access method that generates a row number that is used to perform the table probe operation.
	• 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	

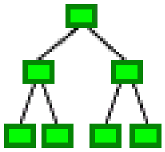
Radix index

An SQL index (or keyed sequence access path) is a permanent object that is created over a table and used by the optimizer to provide a sequenced view of the data for a scan or probe operation.

The rows in the tables are sequenced in the index based upon the key columns specified on the creation of the object. When the key columns are matched up by the optimizer to a query, it gives the optimizer the ability to use the radix index to help satisfy any selection, ordering, grouping or join requirements.

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

Visual Explain icon:



Radix index scan:

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

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

Table 4. Radix index scan attributes

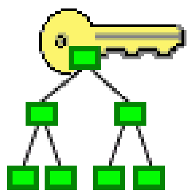
Data access method	Radix index scan
Description	Sequentially scan and process all of the keys associated with the index. Any selection is applied to every key value of the index before a table row
Advantages	• Only those index entries that match any selection continue to be processed • Potential to extract all of the data from the index keys' 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 a large number of rows are selected because of the random I/O associated with the Table Probe.
Likely to be used	• 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	`CREATE INDEX X1 ON Employee (LastName, WorkDept)`
	
	`SELECT * FROM Employee`
	`WHERE WorkDept BETWEEN 'A01' AND 'E01'`
	`ORDER BY LastName`
	`OPTIMIZE FOR 30 ROWS`

Messages indicating use	• Optimizer Debug:
	`CPI4328 -- Access path of file X1 was used by query.`
	• PRTSQLINF:
	`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 ALWCPYDTA parameter on database performance" on page 172
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 Radix Index Scan is that the rows being returned must first be identified by a probe operation to subset the rows being retrieved.

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

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

Table 5. Radix index probe attributes

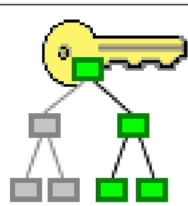
Data access method	Radix index probe
Description	The index is quickly probed based upon the selection criteria that were rewritten into a series of ranges. Only those keys that satisfy the selection will be used to generate a table row number.
Advantages	• Only those index entries that match any selection continue to be processed • Provides very quick access to the selected rows • Potential to extract all of the data from the index keys' 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 a large number of rows are selected because of the random I/O associated with the Table Probe.
Likely to be used	• 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	`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`

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

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

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

Related concepts

"Nested loop join implementation" on page 46
DB2 Universal Database for iSeries provides a **nested loop** join method. For this method, the processing of the tables in the join are ordered. This order is called the **join order**. The first table in the final join order is called the **primary table**. The other tables are called **secondary tables**. Each join table position is called a **dial**.

Related reference

"Effects of the ALWCPYDTA parameter on database performance" on page 172
Some complex queries can perform better by using a sort or hashing method to evaluate the query instead of using or creating an index.

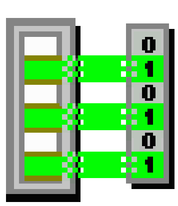
Encoded vector index

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

The size of the vector will match the number of rows in the underlying table. Each vector entry represents the table row number in the same position. The codes generated to represent the distinct key values can be 1, 2 or 4 bytes in length, depending upon the number of distinct values that need to be represented. Because of their compact size and relative simplicity, the EVI can be used to process large amounts of data very efficiently.

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

Visual explain icon:



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 will be sequentially scanned, though the sequence of the resulting entries is not in any guaranteed order. The symbol table can be used by the optimizer to satisfy group by or distinct portions of a query request. Any selection is applied to every entry in the symbol table. All entries are retrieved directly from the symbol table portion of the index without any access to the vector portion of the index nor any access to the records in the associated table over which the EVI is built.

Table 6. Encoded vector index symbol table scan attributes

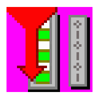
Data access method	**Encoded vector index symbol table scan**
Description	Sequentially scan and process all of the symbol table entries associated with the index. Any selection is applied to every entry in the symbol table. Selected entries are retrieved directly without any access to the vector or the associated table
Advantages	• Pre-summarized results are readily available • Only processes the unique values in the symbol table, avoiding processing table records. • Extract all of the data from the index unique key values, thus eliminating the need for a Table Probe or vector scan.
Considerations	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 a large number of groups involved such that the symbol table is very large, especially if a large portion of symbol table has been put into the overflow area.
Likely to be used	• When asking for GROUP BY, DISTINCT, COUNT or COUNT DISTINCT from a single table and the referenced column(s) are in the key definition • When the number of unique values in the column(s) 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 very much.

Example SQL statement	`CREATE ENCODED VECTOR INDEX EVI1 ON Sales (Region)`
	
	Example 1
	`SELECT Region, count(*)`
	`FROM Sales`
	`GROUP BY Region`
	`OPTIMIZE FOR ALL ROWS`
	
	Example 2
	`SELECT DISTINCT Region`
	`FROM Sales`
	`OPTIMIZE FOR ALL ROWS`
	
	Example 3
	`SELECT COUNT(DISTINCT Region)`
	`FROM Sales`

Table 6. Encoded vector index symbol table scan attributes (continued)

Data access method	Encoded vector index symbol table scan

Messages indicating use	• Optimizer Debug:
	`CPI4328 -- Access path of file EVI1 was used by query.`
	• PRTSQLINF:
	`SQL4008 -- Index EVI1 used for table 1.SQL4010`

SMP parallel enabled	No. Typically not critical as the 'grouping' has already been performed during the index build.
Also referred to as	Encoded Vector Index table scan, Preload
Visual Explain icon	

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 7. Encoded vector index probe attributes

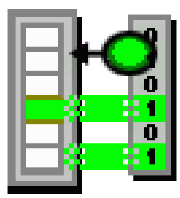
Data access method	Encoded vector index probe
Description	The encoded vector index (EVI) is quickly probed based upon the selection criteria that were rewritten into a series of ranges. It produces either a temporary row number list or bitmap.
Advantages	• Only those index entries that match any selection continue to be processed • Provides very 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 generally built over a single key. The more distinct the column is and the higher the overflow percentage, the less advantageous the encoded vector index becomes. EVIs always require a Table Probe to be performed on the result of the EVI probe operation.
Likely to be used	• 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 of probing the EVI and fully populating the temporary row number list.

Example SQL statement	`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`

Table 7. Encoded vector index probe attributes (continued)

Data access method	Encoded vector index probe

Messages indicating use	• Optimizer Debug:
	`CPI4329 -- Arrival sequence was used for file EMPLOYEE.`
	`CPI4338 –– 3 Access path(s) used for bitmap processing of file EMPLOYEE.`
	• PRTSQLINF:
	`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 that will be used to help schedule the I/O paging against the table for the selected rows.

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

Temporary objects and access methods

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

Table 8. Temporary object's data access methods

Temporary create objects	Scan operations	Probe operations
Temporary hash table	Hash table scan	Hash table probe
Temporary sort list	Sorted list scan	Sorted list probe
Temporary list	List scan	N/A
Temporary row number list	Row number list scan	Row number list probe
Temporary bitmap	Bitmap scan	Bitmap probe

Table 8. Temporary object's data access methods (continued)

Temporary create objects	Scan operations	Probe operations
Temporary index	Temporary index scan	Temporary index probe
Temporary buffer	Buffer scan	N/A
Queue	N/A	N/A

Temporary hash table

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

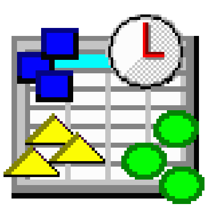
A temporary hash table is an efficient data structure because the rows are organized for quick and easy retrieval after population has occurred. This is primarily due to the hash table remaining resident within main memory so as to avoid any I/Os associated with either the scan or probe against the temporary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

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

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

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

Visual explain icon:



Hash table scan:

During a Hash Table Scan operation, the entire temporary hash table is scanned and all of the entries contained within the hash table will be processed.

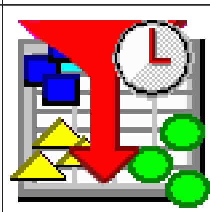
The optimizer considers a hash table scan when the data values need to be collated together, but the sequence of the data is not required. The use of a hash table scan will allow the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary hash table.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

Table 9. Hash table scan attributes

Data access method	Hash table scan
Description	Read all of the entries in a temporary hash table. The hash table may perform distinct processing to eliminate duplicates or takes advantage of the temporary hash table to collate all of the rows with the same value together.
Advantages	• Reduces the random I/O to the table generally associated with longer running queries that may 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	Generally used for distinct or group by processing. Can perform poorly when the entire hash table does not stay resident in memory as it is being processed.
Likely to be used	• 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	`SELECT COUNT(*), FirstNme FROM Employee`
	`WHERE WorkDept BETWEEN 'A01' AND 'E01'`
	`GROUP BY FirstNme`

Messages indicating use	There are multiple ways in which a hash scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a hash scan was used.
	• Optimizer Debug:
	`CPI4329 -- Arrival sequence was used for file EMPLOYEE.`
	• PRTSQLINF:
	`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
Visual Explain icon	

Hash table probe:

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

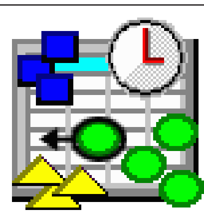
The optimizer initially identifies the keys of the temporary hash table from the join criteria specified in the query. This is done so that when the hash table probe is performed, the values used to probe into the temporary hash table will be extracted from the join-from criteria specified in the selection. Those values will be sent through the same hashing algorithm used to populate the temporary hash table in order to determine if any rows have a matching (equal) value. All of the matching join rows are then returned to be further processed by the query.

Table 10. Hash table probe attributes

Data access method	**Hash table probe**
Description	The temporary hash table is quickly probed based upon the join criteria.
Advantages	• Provides very quick access to the selected rows that match probe criteria • Reduces the random I/O to the table generally 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	Generally used to process equal join criteria. Can perform poorly when the entire hash table does not stay resident in memory as it is being processed.
Likely to be used	• 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	`SELET * FROM Employee XXX, Department YYY`
	`WHERE XXX.WorkDept = YYY.DeptNbr`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	There are multiple ways in which a hash probe can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a hash probe was used.
	• Optimizer Debug:
	`CPI4327 -- File EMPLOYEE processed in join position 1.`
	`CPI4327 -- File DEPARTMENT processed in join position 2.`
	• PRTSQLINF:
	`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	Hash Table Probe, Preload Hash Table Probe Distinct Hash Table Probe Distinct, Preload
Visual Explain icon	

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

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

Related concepts

"Nested loop join implementation" on page 46
DB2 Universal Database for iSeries provides a **nested loop** join method. For this method, the processing of the tables in the join are ordered. This order is called the **join order**. The first table in the final join order is called the **primary table**. The other tables are called **secondary tables**. Each join table position is called a **dial**.

Temporary sorted list

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

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

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

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

Visual explain icon:

Sorted list scan:

During a sorted list scan operation, the entire temporary sorted list is scanned and all of the entries contained within the sorted list will be processed.

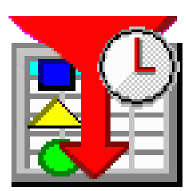
A sorted list scan is generally considered when the optimizer is considering a plan that requires the data values to be sequenced. The use of a sorted list scan will allow the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary sorted list. An additional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

Table 11. Sorted list scan attributes

Data access method	Sorted list scan
Description	Read all of the entries in a temporary sorted list. The sorted list may perform distinct processing to eliminate duplicate values or take advantage of the temporary sorted list to sequence all of the rows.
Advantages	• Reduces the random I/O to the table generally 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	Generally used to process ordering or distinct processing. Can perform poorly when the entire sorted list does not stay resident in memory as it is being populated and processed.
Likely to be used	• 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	`CREATE INDEX X1 ON Employee (LastName, WorkDept)`
	
	`SELECT * FROM Employee`
	`WHERE WorkDept BETWEEN 'A01' AND 'E01'`
	`ORDER BY FirstNme`
	`OPTIMZE FOR ALL ROWS`

Messages indicating use	There are multiple ways in which a sorted list scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a sorted list scan was used.
	• Optimizer Debug:
	`CPI4328 -- Access path of file X1 was used by query.`
	`CPI4325 -- Temporary result file built for query.`
	• PRTSQLINF:
	`SQL4008 -- Index X1 used for table 1.`
	`SQL4002 -- Reusable ODP sort used.`

SMP parallel enabled	No
Also referred to as	Sorted List Scan, Preload Sorted List Scan Distinct Sorted List Scan Distinct, Preload
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 keys of the temporary sorted list from the join criteria specified in the query. This is done so that when the sorted list probe is performed, the values used to probe into the temporary sorted list will be extracted from the join-from criteria specified in the selection. Those values will be used to position within the sorted list in order to determine if any rows have a matching value. All of the matching join rows are then returned to be further processed by the query.

Table 12. Sorted list probe attributes

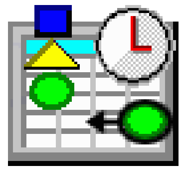
Data access method	**Sorted list probe**
Description	The temporary sorted list is quickly probed based upon the join criteria.
Advantages	• Provides very quick access to the selected rows that match probe criteria • Reduces the random I/O to the table generally associated with longer running queries 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	Generally used to process non-equal join criteria. Can perform poorly when the entire sorted list does not stay resident in memory as it is being populated and processed.
Likely to be used	• 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	`SELECT * FROM Employee XXX, Department YYY`
	`WHERE XXX.WorkDept > YYY.DeptNbr`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	There are multiple ways in which a sorted list probe can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a sorted list probe was used.
	• Optimizer Debug:
	`CPI4327 -- File EMPLOYEE processed in join position 1.`
	`CPI4327 -- File DEPARTMENT processed in join position 2.`
	• PRTSQLINF:
	`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	Sorted List Probe, Preload Sorted List Probe Distinct Sorted List Probe Distinct, Preload

Table 12. Sorted list probe attributes (continued)

Data access method	Sorted list probe
Visual Explain icon	

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

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

Related concepts

"Nested loop join implementation" on page 46
DB2 Universal Database for iSeries provides a **nested loop** join method. For this method, the processing of the tables in the join are ordered. This order is called the **join order**. The first table in the final join order is called the **primary table**. The other tables are called **secondary tables**. Each join table position is called a **dial**.

Temporary list

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

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

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

Visual explain icon:

List scan:

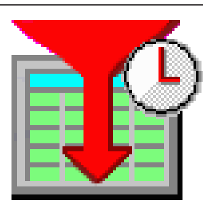
The list scan operation is used when a portion of the query will be processed multiple times, but no key columns can be identified. In these cases, that portion of the query is processed once and its results are stored within the temporary list. The list can then be scanned for only those rows that satisfy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

Table 13. List scan attributes

Data access method	List scan
Description	Sequentially scan and process all of the rows in the temporary list.
Advantages	• The temporary list and list scan can be used by the optimizer to minimize repetition of an operation or to simplify the optimizer's logic flow • Selection can be performed before generating the list to subset the number of rows in the temporary object
Considerations	Generally used to prevent portions of the query from being processed multiple times when no key columns are required to satisfy the request.
Likely to be used	• When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) • When Symmetric Multiprocessing will be used for the query

Example SQL statement	`SELECT * FROM Employee XXX, Department YYY`
	`WHERE XXX.LastName IN ('Smith', 'Jones', 'Peterson')`
	`AND YYY.DeptNo BETWEEN 'A01' AND 'E01'`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	There are multiple ways in which a list scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a list scan was used.
	• Optimizer Debug:
	`CPI4325 -- Temporary result file built for query.`
	`CPI4327 -- File EMPLOYEE processed in join position 1.`
	`CPI4327 -- File DEPARTMENT processed in join position 2.`
	• PRTSQLINF:
	`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.`

SMP parallel enabled	Yes
Also referred to as	List Scan, Preload
Visual Explain icon	

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

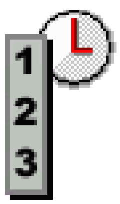
Temporary row number list

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

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

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

Visual explain icon:



Row number list scan:

During a row number list scan operation, the entire temporary row number list is scanned and all of the row addresses contained within the row number list will be processed. A row number list scan is generally considered when the optimizer is considering a plan that involves an encoded vector index or if the cost of the random I/O associated with an index probe or scan operation can be reduced by first preprocessing and sorting the row numbers associated with the Table Probe operation.

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

A row number list scan is identical to a bitmap scan operation. The only difference between the two operations is that a row number list scan is performed over a list of row addresses while the bitmap scan is performed over a bitmap that represents the row addresses.

Table 14. Row number list scan

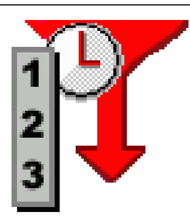
Data access method	**Row number list scan**
Description	Sequentially scan and process all of the row numbers in the temporary row number list. The sorted row numbers can be merged with other temporary row number lists or can be used as input into a Table Probe operation.

Table 14. Row number list scan (continued)

Data access method	Row number list scan
Advantages	• 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 only contains the addresses of the selected row in the table, a separate Table Probe operation must be performed in order to fetch the table rows
Likely to be used	• 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

Example SQL statement	`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`

Messages indicating use	There are multiple ways in which a row number list scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a row number list scan was used.
	• Optimizer Debug:
	`CPI4329 -- Arrival sequence was used for file`
	`           EMPLOYEE.`
	`CPI4338 -- 3 Access path(s) used for bitmap`
	`           processing of file EMPLOYEE.`
	• PRTSQLINF:
	`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	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. This query used a combination of a radix index and two encoded vector indexes to create the row number lists. The temporary row number lists for each index was scanned and merged into a final composite row number list that represents the intersection of the rows represented by all of the temporary row number lists. The final row number list is then used by the Table Probe operation to determine what rows are selected and need to be processed for the query results.

Row number list probe:

A row number list probe operation is used to test row numbers generated by a separate operation against the selected rows of a temporary row number list. The row numbers can be generated by any operation that constructs a row number for a table. That row number is then used to probe into a temporary row number list to determine if that row number matches the selection used to generate the temporary row number list.

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

A row number list probe is identical to a bitmap probe operation. The only difference between the two operations is that a row number list probe is performed over a list of row addresses while the bitmap probe is performed over a bitmap that represents the row addresses.

Table 15. Row number list probe

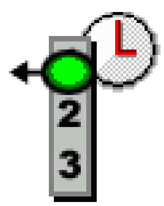
Data access method	Row number list probe
Description	The temporary row number list is quickly probed based upon the row number generated by a separate operation.
Advantages	• The temporary row number list only contains a rows' address, no data, so the temporary can be efficiently probed within memory • 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 only contains the addresses of the selected rows in the table, a separate Table Probe operation must be performed in order to fetch the table rows
Likely to be used	• When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) • 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	`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`

Table 15. Row number list probe (continued)

Data access method	Row number list probe

Messages indicating use	There are multiple ways in which a row number list probe can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a row number list probe was used.
	• Optimizer Debug:
	`CPI4328 -- Access path of file X1 was used by query.`
	`CPI4338 -– 2 Access path(s) used for bitmap processing of file EMPLOYEE.`
	• PRTSQLINF:
	`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.`

SMP parallel enabled	Yes
Also referred to as	Row Number List Probe, Preload
Visual Explain icon	

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

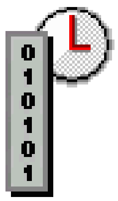
Temporary bitmap

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

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

Visual explain icon:



Bitmap scan:

During a bitmap scan operation, the entire temporary bitmap is scanned and all of the row addresses contained within the bitmap will be processed. A bitmap scan is generally considered when the optimizer is considering a plan that involves an encoded vector index or if the cost of the random I/O associated with an index probe or scan operation can be reduced by first preprocessing and sorting the row numbers associated with the Table Probe operation.

The use of a bitmap scan will allow the optimizer to generate a plan that can take advantage of multiple indexes to match up to different portions of the query.

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

A bitmap scan is identical to a row number list scan operation. The only difference between the two operations is that a row number list scan is performed over a list of row addresses while the bitmap scan is performed over a bitmap that represents the row addresses.

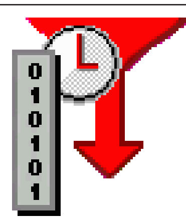
Table 16. Bitmap scan attributes

Data access method	Bitmap scan attributes
Description	Sequentially scan and process all of the row numbers in the temporary bitmap. The sorted row numbers can be merged with other temporary bitmaps or can be used as input into a Table Probe operation.
Advantages	• The temporary bitmap only contains a reference to a rows' 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 only contains the addresses of the selected row in the table, a separate Table Probe operation must be performed in order to fetch the table rows
Likely to be used	• 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

Table 16. Bitmap scan attributes (continued)

Data access method	Bitmap scan attributes

Example SQL statement	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

Messages indicating use	There are multiple ways in which a bitmap scan can be indicated through the messages. The messages in this example illustrate how the Classic Query Engine will indicate a bitmap scan was used. • Optimizer Debug: CPI4329 -- Arrival sequence was used for file EMPLOYEE. CPI4338 -– 3 Access path(s) used for bitmap processing of file EMPLOYEE. • PRTSQLINF: 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 his query. This query used a combination of a radix index and two encoded vector indexes to create the row number lists. The temporary bitmaps for each index were scanned and merged into a final composite bitmap that represents the intersection of the rows represented by all of the temporary bitmaps. The final bitmap is then used by the Table Probe operation to determine what rows are selected and need to be processed for the query results.

Bitmap probe:

A bitmap probe operation is used to test row numbers generated by a separate operation against the selected rows of a temporary bitmap. The row numbers can be generated by any operation that constructs a row number for a table. That row number is then used to probe into a temporary bitmap to determine if that row number matches the selection used to generate the temporary bitmap.

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

A bitmap probe is identical to a row number list probe operation. The only difference between the two operations is that a row number list probe is performed over a list of row addresses while the bitmap probe is performed over a bitmap that represents the row addresses.

Table 17. Bitmap probe attributes

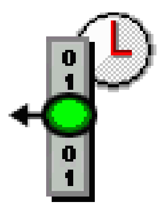
Data access method	Bitmap probe attributes
Description	The temporary bitmap is quickly probed based upon the row number generated by a separate operation.
Advantages	• The temporary bitmap only contains a reference to a rows' address, no data, so the temporary can be efficiently probed within memory • 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 only contains the addresses of the selected rows in the table, a separate Table Probe operation must be performed in order to fetch the table rows
Likely to be used	• 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	`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`

Table 17. Bitmap probe attributes (continued)

Data access method	Bitmap probe attributes

Messages indicating use	There are multiple ways in which a bitmap probe can be indicated through the messages. The messages in this example illustrate how the Classic Query Engine will indicate a bitmap probe was used.
	• Optimizer Debug:
	`CPI4328 -- Access path of file X1 was used by query.`
	`CPI4338 -– 2 Access path(s) used for bitmap`
	`processing of file EMPLOYEE.`
	• PRTSQLINF:
	`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.`

SMP parallel enabled	Yes
Also referred to as	Bitmap Probe, Preload Row Number Bitmap Probe Row Number Bitmap Probe, Preload
Visual Explain icon	

Using the example above, the optimizer created a temporary bitmap for each of the encoded vector indexes. Additionally, an index probe operation was performed against the radix index X1 to satisfy the ordering requirement. Since the ORDER BY clause requires that the resulting rows be sequenced by the WorkDept column, the temporary bitmap can no longer be scanned to process the selected rows. However, the temporary bitmap can be probed using a row address extracted from the index X1 used to satisfy the ordering. By probing the temporary bitmap with the row address extracted from index probe operation, the sequencing of the keys in the index X1 are preserved and 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 of the same attributes and benefits as a radix index that is created by a user 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 includes setting the logical page size and applying any selection to the creation to speed up the use of the temporary index after it has been created.

The temporary index can be used to satisfy a variety of query requests:
- Ordering
- Grouping/Distinct

- Joins
- Record selection

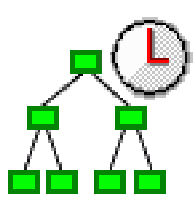
Generally a temporary index is a more expensive temporary object to create than other temporary objects. It can be populated by either performing a table scan to fetch the rows to be used for the index or by performing an index scan or probe against one or more indexes to produce the rows. The optimizer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

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

SQE usage of temporary indexes is different than 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, a 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 SQE optimizer use of temporary indexes to behave more like CQE optimizer; it becomes shorter lived, but 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 of the rows in the index will be sequentially processed, but the resulting row numbers will be sequenced based upon the key columns.

The sequenced rows can be used by the optimizer to satisfy a portion of the query request (such as ordering or grouping).

Table 18. Temporary index scan attributes

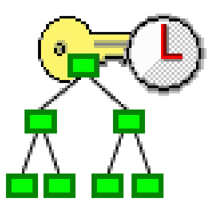
Data access method	**Temporary index scan**
Description	Sequentially scan and process all of the keys associated with the temporary index.

Table 18. Temporary index scan attributes (continued)

Data access method	Temporary index scan
Advantages	• Potential to extract all of the data from the index keys' 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 a large number of rows are selected because of the random I/O associated with the Table Probe.
Likely to be used	• 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	`SELECT * FROM Employee`
	`WHERE WorkDept BETWEEN 'A01' AND 'E01'`
	`ORDER BY LastName`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	• 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 will determine where the selection against the WorkDept column best belongs. It can be performed as the temporary index itself is being created or it can be performed as a part of the temporary index scan. Adding the selection to the temporary index creation has the possibility of making the open data path (ODP) for this query non-reusable. This ODP reuse is taken into consideration when determining how selection will be performed.

Temporary index probe:

A temporary index probe operation is identical to the index probe operation that is performed upon the permanent radix index. Its main function is to provide a form of quick access against the index keys of the temporary index; however it can still 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 19. Temporary index probe attributes

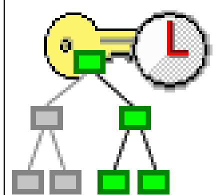
Data access method	Temporary index probe
Description	The index is quickly probed based upon the selection criteria that were rewritten into a series of ranges. Only those keys that satisfy the selection will be used to generate a table row number.
Advantages	• Only those index entries that match any selection continue to be processed. Provides very quick access to the selected rows • Potential to extract all of the data from the index keys' 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 a large number of rows are selected because of the random I/O associated with the Table Probe.
Likely to be used	• 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	`SELET * FROM Employee XXX, Department YYY`
	`WHERE XXX.WorkDept = YYY.DeptNo`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	There are multiple ways in which a temporary index probe can be indicated through the messages. The messages in this example illustrate one example of how the Classic Query Engine will indicate a temporary index probe was used.
	• Optimizer Debug:
	`CPI4321 -- Access path built for file DEPARTMENT.`
	`CPI4327 -- File EMPLOYEE processed in join position 1.`
	`CPI4326 -- File DEPARTMENT processed in join position 2.`
	• PRTSQLINF:
	`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.`

SMP parallel enabled	Yes

Table 19. Temporary index probe attributes (continued)

Data access method	Temporary index probe
Also referred to as	Index Probe Index Probe, Preload Index Probe, Distinct Index Probe Distinct, Preload Index Probe, Key Selection
Visual Explain icon	

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

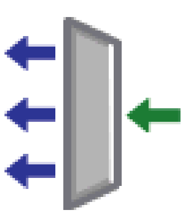
Temporary buffer

The temporary buffer is a temporary object that is used to help facilitate operations such as parallelism. It is an unsorted data structure that is used to store intermediate rows of a query. The main difference between a temporary buffer and a temporary list is that the buffer does not need to be fully populated in order to allow its results to be processed.

The temporary buffer acts as a serialization point between parallel and non-parallel portions of a query. The operations used to populate the buffer cannot be performed in parallel, whereas the operations that fetch rows from the buffer can be performed in parallel. The temporary buffer is required for the SQL Query Engine because the index scan and index probe operations are not considered to be SMP parallel enabled for this engine. Unlike the Classic Query Engine, which will perform these index operations in parallel, the SQL Query Engine will not subdivide the work necessary within the index operation to take full advantage of parallel processing. The buffer is used to allow a query to be processed under parallelism by serializing access to the index operations, while allowing any remaining work within the query to be processed in parallel.

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

Visual explain icon:



Buffer scan:

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

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

A buffer scan operation is identical to the list scan operation that is performed upon the temporary list object. The main difference is that a buffer does not 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 20. Buffer scan attributes

Data access method	Buffer scan
Description	Sequentially scan and process all of the rows in the temporary buffer. Enables SMP parallelism to be performed over a non-parallel portion of the query.
Advantages	• 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	Generally used to prevent portions of the query from being processed multiple times when no key columns are required to satisfy the request.
Likely to be used	• When the query is attempting to take advantage of DB2 UDB Symmetric Multiprocessing • When a portion of the query cannot be performed in parallel (for example, index scan or index probe)

Example SQL statement	`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`

Messages indicating use	• Optimizer Debug:
	`CPI4328 -- Access path of file X1 was used by query.`
	`CPI4330 -- 8 tasks used for parallel index scan`
	`           of file EMPLOYEE.`
	• PRTSQLINF:
	`SQL4027 -- Access plan was saved with DB2 UDB`
	`           SMP installed on the system.`
	`SQL4008 -- Index X1 used for table 1.`
	`SQL4011 -- Index scan-key row positioning`
	`           used on table 1.`
	`SQL4030 -- 8 tasks specified for parallel scan`
	`           on table 1.`

SMP parallel enabled	Yes
Also referred to as	Not applicable

Table 20. Buffer scan attributes (continued)

Data access method	Buffer scan
Visual Explain icon	

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

Queue

The Queue is a temporary object that allows the optimizer to feed the recursion of a recursive query by putting on the queue those data values needed for the recursion. This data typically includes those values used on the recursive join predicate and other recursive data being 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 that data needed to feed the recursion or directly modified by the recursion process and its size is managed by the optimizer.

Unlike other temporary objects created by the optimizer, the queue is not populated in all at once by the underlying query node tree but is really a real time temporary holding area for values feeding the recursion. In this regard, a queue is not considered temporary as it will not prevent the query from running if ALWCPYDTA(*NO) was specified, because the data can still being flowing up and out of the query at the same time the recursive values are inserted into the queue to be used to retrieve additional join rows.

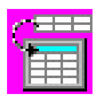
A queue is an internal data structure and can only be created by the database manager.

Visual explain icon:

Enqueue:

During a enqueue operation, an entry it put on the queue that 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 21. Enqueue Attributes

<table>
<tr><th>Data Access Method</th><th>Enqueue</th></tr>
<tr><td>Description</td><td>Places an entry on the queue needed to cause further recursion</td></tr>
<tr><td>Advantages</td><td>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/view.</td></tr>
<tr><td>Likely to be used</td><td>A required access method for recursive queries</td></tr>
<tr><td>Example SQL statement</td><td><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></td></tr>
<tr><td>Messages indicating use</td><td>There are no explicit message that indicate the use of an enqueue</td></tr>
<tr><td>SMP parallel enabled</td><td>Yes</td></tr>
<tr><td>Also referred to as</td><td>Not applicable</td></tr>
<tr><td>Visual Explain icon</td><td></td></tr>
</table>

Use the CYCLE option in the definition of the recursive query if there is the possibility that the data reflecting the parent, child relationship may be cyclic, causing an infinite recursion loop. CYCLE will prevent 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, which columns make up the parent child relationship and the search type of Depth or Breadth but 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:

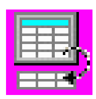
During a dequeue operation, an entry is taken off the queue and those values specified by recursive reference are fed back in to the recursive join process.

The optimizer always supplies a corresponding enqueue, dequeue pair of operations for each reference of a 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 22. Dequeue Attributes

Data Access Method	Dequeue
Description	Removes an entry off the queue, provides minimally 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 is always the left side of inner join with constraint where the right side of the join being the target child rows.
Advantages	• Provides very quick access to recursive values • Allows for post selection of local predicate on recursive data values
Likely to be used	• A required access method for recursive queries • A single dequeued values can feed the recursion of multiple iterative fullselects that reference the same rcte/view

Example SQL statement	`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`

Messages indicating use	There are no explicit messages that indicate the use of a dequeue
SMP parallel enabled	Yes
Also referred to as	Not applicable
Visual Explain icon	

Objects processed in parallel

The DB2 UDB Symmetric Multiprocessing feature provides the optimizer with additional methods for retrieving data that include parallel processing. Symmetrical multiprocessing (SMP) is a form of parallelism achieved on a single server where multiple (CPU and I/O) processors that share memory and disk resource work simultaneously toward achieving a single end result.

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

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

Processing requirements

Parallelism requires that SMP parallel processing must be enabled 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 server startup for use by the database manager. The database manager uses the tasks to process and retrieve data from different disk devices. Since these tasks can be run on multiple processors simultaneously, the elapsed time of a query can be reduced. Even though much of the I/O and CPU processing of a parallel query is done by the tasks, the accounting of the I/O and CPU resources used are transferred to the application job. The summarized I/O and CPU resources for this type of application continue to be accurately displayed by the Work with Active Jobs (WRKACTJOB) command.

| The job should be run in a shared storage pool with the *CALC paging option, as this will cause more efficient use of active memory.

Related concepts

"Nested loop join implementation" on page 46
DB2 Universal Database for iSeries provides a **nested loop** join method. For this method, the processing of the tables in the join are ordered. This order is called the **join order**. The first table in the final join order is called the **primary table**. The other tables are called **secondary tables**. Each join table position is called a **dial**.

Related reference

"Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command" on page 115
You can modify different types of attributes of the queries that you will execute during a certain job with the Change Query Attributes (CHGQRYA) CL command, or by using the iSeries Navigator Change Query Attributes interface.

Related information

SET CURRENT DEGREE statement

Parallel processing for queries and indexes system value

Automatically tune performance

Work with Active Jobs (WRKACTJOB) command

Change Query Attributes (CHGQRYA) command

Spreading data automatically

DB2 Universal Database for iSeries automatically spreads the data across the disk devices available in the auxiliary storage pool (ASP) where the data is allocated. This ensures that the data is spread without user intervention.

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

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

Processing queries: Overview

This overview of the query optimizer provides guidelines for designing queries that will perform and will use server resources more efficiently.

This overview covers queries that are optimized by the query optimizer and includes interfaces such as SQL, OPNQRYF, APIs (QQQQRY), ODBC, and Query/400 queries. Whether you apply the guidelines, the query results will still be correct.

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

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

- How the server accesses data.

  These methods are the algorithms that are used to retrieve data from the disk. The methods include index usage and row selection techniques. In addition, parallel access methods are available with the DB2 UDB Symmetric Multiprocessing operating system feature.
- Query optimizer.

  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. You should understand the techniques employed by the query optimizer for performing this task.

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

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

General query optimization tips

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

- Create indexes whose leftmost key columns match your selection predicates to help supply the optimizer with selectivity values (key range estimates).
- For join queries, create indexes that match your join columns to help the optimizer determine the average number of matching rows.
- Minimize extraneous mapping by specifying only columns of interest on the query. For example, specify only the columns you need to query on the SQL SELECT statement instead of specifying SELECT *. Also, you should specify FOR FETCH ONLY if the columns do not need to be updated.
- If your queries often use table scan access method, use the 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 iSeries Access ODBC driver and Query Management driver always uses this mode. If ALWCPYDTA(*YES) is specified, the query optimizer will attempt to implement the query without copies of the data, but may create copies if required. If ALWCPYDTA(*NO) is specified, copies of the data 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 (see exceptions below) once it is built so as to be available for potentially future runs of the query. However, the optimizer has dynamic replan capability. This means that even if previously built (and saved) plan is found, the optimizer may rebuild it if it determines that a more optimal plan is possible. This 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 may be rebuilt the first time the query is opened (when the host variable values are present).
- For an iSeries 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 may 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 cases above where a plan is saved above, 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 the query by calculating an *implementation cost* based on the current state of the database. The optimizer uses 2 costs when making decisions: an I/O cost and a CPU cost. The goal of the optimizer is to minimize both I/O and CPU cost.

Optimizing Access to each table

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

- Determines the default filter factor for each predicate in the selection clause.
- Determines the true filter factor of the predicates by doing a key range estimate when the selection predicates match the left most keys of an index or by using columns statistic when available.
- Determines the cost of table scan processing if an index is not required.
- Determines the cost of creating an index over a table if an index is required. This index is created by performing either a table scan or creating an index-from-index.
- Determines the cost of using a sort routine or hashing method if appropriate.
- Determines the cost of using existing indexes using Index Probe or Index Scan

– Orders the indexes. For SQE, the indexes are ordered in general such that the indexes that access the smallest number of entries are examined first. For CQE, the indexes are generally ordered from mostly recently created to oldest.
– For each index available, the optimizer does the following:
  - Determines if the index meets the selection criteria.
  - Determines the cost of using the index by estimating the number of I/Os and the CPU cost that will be needed to perform the Index Probe or the Index Scan and the possible Table Probes.
  - Compares the cost of using this index with the previous cost (current best).
  - Picks the cheaper one.
  - Continues to search for best index until the optimizer decides to look at no more indexes.

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

  For CQE, the *time limit* controls how much time the optimizer spends choosing an implementation. It is based on how much time was spent so far and the current best implementation cost found. The idea is to prevent the optimizer from spending more time optimizing the query than it takes to actually execute the query. Dynamic SQL queries are subject to the optimizer time restrictions. Static SQL queries optimization time is not limited. For OPNQRYF, if you specify OPTALLAP(*YES), the optimization time is not limited. For small tables, the query optimizer spends little time in query optimization. For large tables, the query optimizer considers more indexes. Generally, the optimizer considers five or six indexes (for each table of a join) before running out of optimization time. Because of this, it is normal for the optimizer to spend longer lengths of time analyzing queries against larger tables.

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

Join optimization

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

Nested loop join implementation

DB2 Universal Database for iSeries provides a **nested loop** join method. For this method, the processing of the tables in the join are ordered. This order is called the **join order**. The first table in the final join order is called the **primary table**. The other tables are called **secondary tables**. Each join table position is called a **dial**.

The nested loop will be implemented either using an index on secondary tables, a hash table, or a table scan (arrival sequence) on the secondary tables. In general, the join will be implemented using either an index or a hash table.

Index nested loop join implementation

During the join, DB2 Universal Database for iSeries:

1. Accesses the first primary table row selected by the predicates local to the primary table.

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

     All rows that satisfy the join condition from each secondary dial are located using an index. Rows are retrieved from secondary tables in random sequence. This random disk I/O time often accounts for a large percentage of the processing time of the query. Since a given secondary dial is searched once for each row selected from the primary and the preceding secondary dials that satisfy the join condition for each of the preceding secondary dials, a large number of searches may be performed against the later dials. Any inefficiencies in the processing of the later dials can significantly inflate the query processing time. This is the reason why attention to performance considerations for join queries can reduce the run-time of a join query from hours to minutes.

     If an efficient index cannot be found, a temporary index may be created. Some join queries build temporary indexes over secondary dials even when an index exists for all of the join keys. Because efficiency is very important for secondary dials of longer running queries, the query optimizer may choose to build a temporary index which contains only entries which pass the local row selection for that dial. This preprocessing of row selection allows the database manager to process row selection in one pass instead of each time rows are matched for a dial.
   - If using a Hash Table Probe to access the secondary table, a hash temporary result table is created that contains all of the rows selected by local selection against the table on the first probe. The structure of the hash table is such that rows with the same join value are loaded into the same hash table partition (clustered). The location of the rows for any given join value can be found by applying a hashing function to the join value.

     A nested loop join using a Hash Table Probe has several advantages over a nested loop join using an Index Probe:
     - The structure of a hash temporary result table is simpler than that of an index, so less CPU processing is required to build and probe a hash table.
     - The rows in the hash result table contain all of the data required by the query so there is no need to access the dataspace of the table with random I/O when probing the hash table.
     - Like join values are clustered, so all matching rows for a given join value can typically be accessed with a single I/O request.
     - The hash temporary result table can be built using SMP parallelism.
     - Unlike indexes, entries in hash tables are not updated to reflect changes of column values in the underlying table. The existence of a hash table does not affect the processing cost of other updating jobs in the server.
   - If using a Sorted List Probe to access the secondary table, a sorted list result is created that contains all of the rows selected by local selection against the table on the first probe. The structure of the sorted list table is such that rows with the same join value are sorted together in the list. The location of the rows for any given join value can be found by probing using the join value.
   - If using a table scan to access the secondary table, scan the secondary to locate the first row that satisfies the join condition for the first secondary table using the table scan to match the join condition or local row selection columns of the secondary table. The join may be implemented with a table scan when the secondary table is a user-defined table function.
4. Determines if the row is selected by applying any remaining selection local to the first secondary dial.

   If the secondary dial row is not selected then the next row that satisfies the join condition is located. Steps 1 through 4 are repeated until a row that satisfies both the join condition and any remaining selection is selected from all secondary tables
5. Returns the result join row.

6. Processes the last secondary table again to find the next row that satisfies the join condition in that dial.

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

Note the following characteristics of a nested loop join:
- If ordering or grouping is specified and all the columns are over a single table and that table is eligible to be the primary, then the optimizer costs the join with that table as the primary and performing the grouping and ordering with an index.
- If ordering and grouping is specified on two or more tables or if temporaries are allowed, DB2 Universal Database for iSeries breaks the processing of the query into two parts:
  1. Perform the join selection omitting the ordering or grouping processing and write the result rows to a temporary work table. This allows the optimizer to consider any table of the join query as a candidate for the primary table.
  2. The ordering or grouping processing is then performed on the data in the temporary work table.

Queries that cannot use hash join

Hash join cannot be used for queries that:
- Hash join cannot be used for queries involving physical files or tables that have read triggers.
- Require that the cursor position be restored as the result of the SQL ROLLBACK HOLD statement or the ROLLBACK CL command. For SQL applications using commitment control level other than *NONE, this requires that *ALLREAD be specified as the value for the ALWBLK precompiler parameter.
- Hash join cannot be used for a table in a join query where the join condition something other than an equals operator.
- CQE does not support hash join if the query contains any of the following:
  - Subqueries unless all subqueries in the query can be transformed to inner joins.
  - UNION or UNION ALL
  - Perform left outer or exception join.
  - Use a DDS created join logical file.

Related concepts

"Objects processed in parallel" on page 42
The DB2 UDB Symmetric Multiprocessing feature provides the optimizer with additional methods for retrieving data that include parallel processing. Symmetrical multiprocessing (SMP) is a form of parallelism achieved on a single server where multiple (CPU and I/O) processors that share memory and disk resource work simultaneously toward achieving a single end result.

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 of the rows in the table to determine if they satisfy the selection criteria specified in the query. It does this in a way to maximize the I/O throughput for the table.

"Sorted list probe" on page 23
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 20
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 Radix Index Scan is that the rows being returned must first be identified by a probe operation to subset the rows being retrieved.

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

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

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

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

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

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

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

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

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

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

Since the query optimizer attempts a combination of join and local row selection when determining access path usage, it is possible to achieve almost all of the same advantages of a temporary index by use of an existing index. In the above example, using either implementation, an existing index may be used or a temporary index may be created. A temporary index is built with the local row selection on PROJNO applied during the index's 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 or EMSTDATE, PROJNO, EMP_ACT or ...) the local row selection can be applied **at the same time** as the join selection (rather than before the join selection, as happens when the temporary index is created, or after the join selection, as happens when only the first key column of the index matches the join column).

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

Join order optimization

The join order is fixed if any join logical files are referenced. The join order is also fixed if the OPNQRYF JORDER(*FILE) parameter is specified or the query options file (QAQQINI) FORCE_JOIN_ORDER parameter is *YES.

Otherwise, the following join ordering algorithm is used to determine the order of the tables:

1. Determine an access method for each individual table as candidates for the primary dial.
2. Estimate the number of rows returned for each table based on local row selection.

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

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

   ```
   1-2    2-1    1-3    3-1    1-4    4-1   2-3    3-2    2-4    4-2    3-4    4-3
   ```
4. Choose the combination with the lowest join cost and number of selected rows or both.
5. Determine the cost, access method, and expected number of rows for each remaining table joined to the previous secondary table.
6. Select an access method for each table that has the lowest cost for that table.
7. Choose the secondary table with the lowest join cost and number of selected rows or both.
8. Repeat steps 4 through 7 until the lowest cost join order is determined.

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

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

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

The allowable join order combinations for this query would be:

1–2–3, 2–1–3, or 2–3–1

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

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

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

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

Related information

Open Query File (OPNQRYF) command

Change Query Attributes (CHGQRYA) command

Cost estimation and index selection for join secondary dials

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

In step 3 and in step 5 in "Join order optimization" on page 50, the query optimizer has to estimate a cost and choose an access method for a given dial combination. The choices made are similar to those for row selection except that a plan using a probe must be chosen.

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 may be loaded into active memory as the result of a Set Object Access (SETOBJACC) CL command. Usage of this command may significantly improve the performance of a query, but the query optimizer does not change the query implementation to take advantage of the memory resident state of the table.
- The query is the only process running on the server. No allowance is given for server CPU utilization or I/O waits which occur because of other processes using the same resources. CPU related costs are scaled to the relative processing speed of the server running the query.
- The values in a column are uniformly distributed across the table. For example, if 10% of the rows in a table have the same value, then it is assumed that every tenth row in the table contains that value.
- The values in a column are independent from the values in any other columns in a row, unless there is an index available whose key definition is (A,B). Multi key field indexes allows the optimizer to detect when the values between columns are correlated. For example, if a column named `A` has a value of `1` in 50% of the rows in a table and a column named `B` has a value of `2` in 50% of the rows, then it is expected that a query which selects rows where `A = 1`, and `B = 2` selects 25% of the rows in the table.

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

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

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

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

Predicates generated through transitive closure

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

See the following examples:

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

The optimizer will modify the query to be:

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

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

- The dials affected must have join operators of equal.
- The predicate is **isolatable**, which means that a false condition from this predicate 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 already exists in the WHERE clause (OPNQRYF QRYSLT parameter).

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

Look ahead predicate generation (LPG)

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

Look at the following query:

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

The optimizer may 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, those subset of EMP_ACT records can then be accessed in a way much more efficient 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 will join at all to EMPLOYEE. This 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. Classic Query Engine 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.

Related reference

"Control queries dynamically with the query options file QAQQINI" on page 116
The query options file QAQQINI support provides the ability to dynamically modify or override the environment in which queries are executed 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.

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 2 columns. If you give the optimizer extra information to work with when requesting a join, it can determine the best way to do the join. The additional information might seem redundant, but is helpful to the optimizer.

If the select-statement you are considering accesses two or more tables, all the recommendations suggested in "Creating an index strategy" on page 141 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

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

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

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

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

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

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

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

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

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

Sources of join query performance problems

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

This occurs when:

- An index is not available which provides average number of duplicate values statistics for the potential join columns.
- The query optimizer uses default filter factors to estimate the number of rows being selected when applying local selection to the table because indexes or column statistics do not exist over the selection columns.

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

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

Tips for improving the performance of join queries

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

Table 23. Checklist for Creating an Application that Uses Join Queries

What to Do	How It Helps
Check the database design. Make sure that there are indexes available over all of the join columns and row selection columns or both. The optimizer provides index advise in several places to aid in this process. Use either the index advisor under iSeries navigator - Database, the advised information under visual explain, or the advised information in the 3020 record in the database monitor.	This gives the query optimizer a better opportunity to select an efficient access method because it can determine the average number of duplicate values. Many queries may be able to use the existing index to implement the query and avoid the cost of creating a temporary index or hash table.
Check the query to see whether some complex predicates should be added to other dials to allow the optimizer to get a better idea of the selectivity of each dial.	Since the query optimizer does not add predicates for predicates connected by OR or non-isolatable predicates, or predicate operators of LIKE, modifying the query by adding these predicates may help.
Specify ALWCPYDTA(*OPTIMIZE) or ALWCPYDTA(*YES)	If the query is creating a temporary index or hash table, and you feel that the processing time may be better if the optimizer only used the existing index or hash table, specify ALWCPYDTA(*YES). If the query is not creating a temporary index or hash table, and you feel that the processing time may be better if a temporary index was created, specify ALWCPYDTA(*OPTIMIZE). Alternatively, specify the OPTIMIZE FOR n ROWS to inform the optimizer of the application has intention to read every resulting row. To do this set n to a large number. You can also set n to a small number before ending the query.
For OPNQRYF, specify OPTIMIZE(*FIRSTIO) or OPTIMIZE(*ALLIO)	Specify the OPTIMIZE(*FIRSTIO) or OPTIMIZE(*ALLIO) option to accurately reflect your application. Use *FIRSTIO, if you want the optimizer to optimize the query to retrieve the first block of rows most efficiently. This biases the optimizer toward using existing objects. If you want to optimize the retrieval time for the entire answer set, use *ALLIO. This may cause the optimizer to create temporary objects such as temporary indexes or hash tables in order to minimize I/O.

Table 23. Checklist for Creating an Application that Uses Join Queries (continued)

<table>
<tr><th>What to Do</th><th>How It Helps</th></tr>
<tr><td>Star join queries</td><td>A join in which one table is joined with all secondary tables consecutively is sometimes called a star join. In the case of a star join where all secondary join predicates contain a column reference to a particular table, there may be performance advantages if that table is placed in join position one. In Example A, all tables are joined to table EMPLOYEE. The query optimizer can freely determine the join order. For SQE, the optimizer uses Look Ahead Predicate generation to determine the optimal join order. For CQE, the query should be changed to force EMPLOYEE into join position one by using the query options file (QAQQINI) FORCE_JOIN_ORDER parameter of *YES. Note that in these examples the join type is a join with no default values returned (this is an inner join.). The reason for forcing the table into the first position is to avoid random I/O processing. If EMPLOYEE is not in join position one, every row in EMPLOYEE can be examined repeatedly during the join process. If EMPLOYEE is fairly large, considerable random I/O processing occurs resulting in poor performance. By forcing EMPLOYEE to the first position, random I/O processing is minimized.

Example A: Star join query<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>Example B: Star join query with order forced via FORCE_JOIN_ORDER<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></td></tr>
<tr><td>Specify ALWCPYDTA(*OPTIMIZE) to allow the query optimizer to use a sort routine.</td><td>In the cases where ordering is specified and all key columns are from a single dial, this allows the query optimizer to consider all possible join orders.</td></tr>
<tr><td>Specify join predicates to prevent all of the rows from one table from being joined to every row in the other table.</td><td>This improves performance by reducing the join fan-out. Every secondary table should have at least one join predicate that references on of its columns as a ’join-to’ column.</td></tr>
</table>

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, will be 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

Below is an example of a query with a DISTINCT:

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

The optimizer will rewrite 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 DISTINCTs can be removed.

Grouping optimization

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

Grouping hash implementation

This technique uses the base hash access method to perform grouping or summarization of the selected table rows. For each selected row, the specified grouping value is run through the hash function. The computed hash value and grouping value are used to quickly find the entry in the hash table corresponding to the grouping value.

If the current grouping value already has a row in the hash table, the hash table entry is retrieved and summarized (updated) with the current table row values based on the requested grouping column operations (such as SUM or COUNT). If a hash table entry is not found for the current grouping value, a new entry is inserted into the hash table and initialized with the current grouping value.

The time required to receive the first group result for this implementation will most likely be longer than other grouping implementations because the hash table must be built and populated first. Once the hash table is completely populated, the database manager uses the table to start returning the grouping results. Before returning any results, the database manager must apply any specified grouping selection criteria or ordering to the summary entries in the hash table.

Where the grouping hash method is most effective

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

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

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

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

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

Index grouping implementation

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

Ordered grouping

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

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

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

Pre-summarized processing

This SQE only implementation utilizes an Encoded Vector Index to extract the summary information already in the index's symbol table. The symbol table portion of an EVI contains the unique values of the key along with a count of the number of table records that have that unique value, basically the grouping for the columns of the index key are already performed. If the query references a single table and performs simple aggregation, the EVI may 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 can result in dramatic improvements in queries when the number of records in the table 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 make use of 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
```

The reason that this query cannot use the EVI is because it references col2 of the table, which is not in the key definition of the EVI. Note also that if multiple columns are defined in the EVI key, for example, col1 and col2, that it is important that the left most columns of the key be utilized. For example, if an EVI existed with a key definition of (col1, col2), but the query referenced just col2, it is very unlikely the EVI will be used.

Optimizing grouping by eliminating grouping columns

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

This processing is done to allow the optimizer to consider more indexes to implement the query and to reduce the number of columns that will be added as key columns to a temporary index or hash table.

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

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

OPNQRYF example:

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

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

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

Optimizing grouping by adding additional grouping columns

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

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

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

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

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

Optimizing grouping by using index skip key processing

Index Skip Key processing can be used when grouping with the keyed sequence implementation algorithm which uses an existing index. It is a specialized version of ordered grouping that processes very 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 will improve performance by potentially not processing all index key values for a group.

Index skip key processing can be used:

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

    Example 1, using SQL:

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

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

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

    Example 2, using SQL:

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

    DECLARE C1 CURSOR FOR
    ```

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

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

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

  Example 1, using SQL:

```
CREATE INDEX IX1 ON DEPARTMENT(DEPTNAME)

CREATE INDEX IX2 ON EMPLOYEE(WORKDEPT, SALARY)

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

Optimizing grouping by removing read triggers

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

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

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

Related information

Add Physical File Trigger (ADDPFTRG) command

Ordering optimization

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

Sort Ordering implementation

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

Index Ordering implementation

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

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

Optimizing ordering by eliminating ordering columns

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

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

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

Optimizing ordering by adding additional ordering columns

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

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

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

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

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

View implementation

Views, 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, FIRSTNME, SALARY
  FROM D21EMPL
  WHERE JOB='CLERK'
```

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

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

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

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

View materialization implementation

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

View materialization is done whenever it is not possible to create a view composite. Note that for SQE, view materialization is optional. The following types of queries require view materialization:

- The outermost select of the view contains grouping, the query contains grouping, and refers to a column derived from a column function in the view in the HAVING or select-list.
- The query is a join and the outermost select of the view contains grouping or DISTINCT.
- The outermost select of the view contains DISTINCT, and the query has UNION, grouping, or DISTINCT and one of the following:
  - Only the query has a shared weight NLSS table
  - Only the view has a shared weight NLSS table
  - Both the query and the view have a shared weight NLSS table, but the tables are different.
- The query contains a column function and the outermost select of the view contains a DISTINCT
- The view does not contain an access plan. This can occur when a view references a view and a view composite cannot be created because of one of the reasons listed above. This does not apply to nested table expressions and common table expressions.
- The Common table expression (CTE) is reference more than once in the query's FROM clause(s) and the CTE's SELECT clause references a MODIFIES or EXTERNAL ACTION UDF.

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

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 will be seen by users and database performance tools.

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

```
SELECT D.DEPTNAME,A.AVGSAL
  FROM CORPDATA.DEPARTMENT D, AVGSALVW A
  WHERE D.DEPTNO=A.WORKDEPT 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 is done by precomputing and storing results of a query in the materialized query table. The database engine can use these results instead of recomputing them for a user specified query. The query optimizer will look for any applicable MQTs and can choose to implement the query using a given MQT provided this is a faster implementation choice.

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

Related information

Create Table statement

MQT supported function

Although a MQT can contain almost any query, the optimizer only supports a limited set of query functions when matching MQTs to user specified queries. The user specified query and the MQT query must both be supported by the SQE optimizer.

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_RC2
  - GENERATE_UNIQUE
  - GETHINT
  - INSERT
  - MONTHNAME
  - NEXT_DAY
  - RADIANS
  - REPEAT
  - REPLACE
  - SOUNDEX
  - VARCHAR_FORMAT

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

Related reference

"Details on the MQT matching algorithm" on page 68
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(*OPTMIZE) or INSENSITIVE cursor.
- The query must not be a SENSITIVE cursor.
- The table to be replaced with a MQT must not be update or delete capable for this query.
- The MQT currently has the ENABLE QUERY OPTIMIZATION attribute active
- The MATERIALIZED_QUERY_TABLE_USAGE QAQQINI option must be set to *ALL or *USER to enable use of MQTs. The default setting of MATERIALIZED_QUERY_TABLE_USAGE does not allow usage of MQTs.
- The timestamp of the last REFRESH TABLE for a MQT is within the duration specified by the MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option or *ANY is specified which allows MQTs to be considered regardless of the last REFRESH TABLE. The default setting of MATERIALIZED_QUERY_TABLE_REFRESH_AGE does not allow usage of MQTs.
- The query must be capable of being run through SQE.
- The following QAQQINI options must match: IGNORE_LIKE_REDUNDANT_SHIFTS, NORMALIZE_DATA, and VARIABLE_LENGTH_OPTIMIZATION. These options are stored at CREATE materialized query table time and must match the options specified at query run time.
- The commit level of the MQT must be greater than or equal to the query commit level. The commit level of the MQT is either specified in the MQT query using the WITH clause or it is defaulted to the commit level that the MQT was run under when it was created.

MQT examples

The following are examples of using MQTs.

Example 1

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

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

Create a table, MQT1, that uses this query:

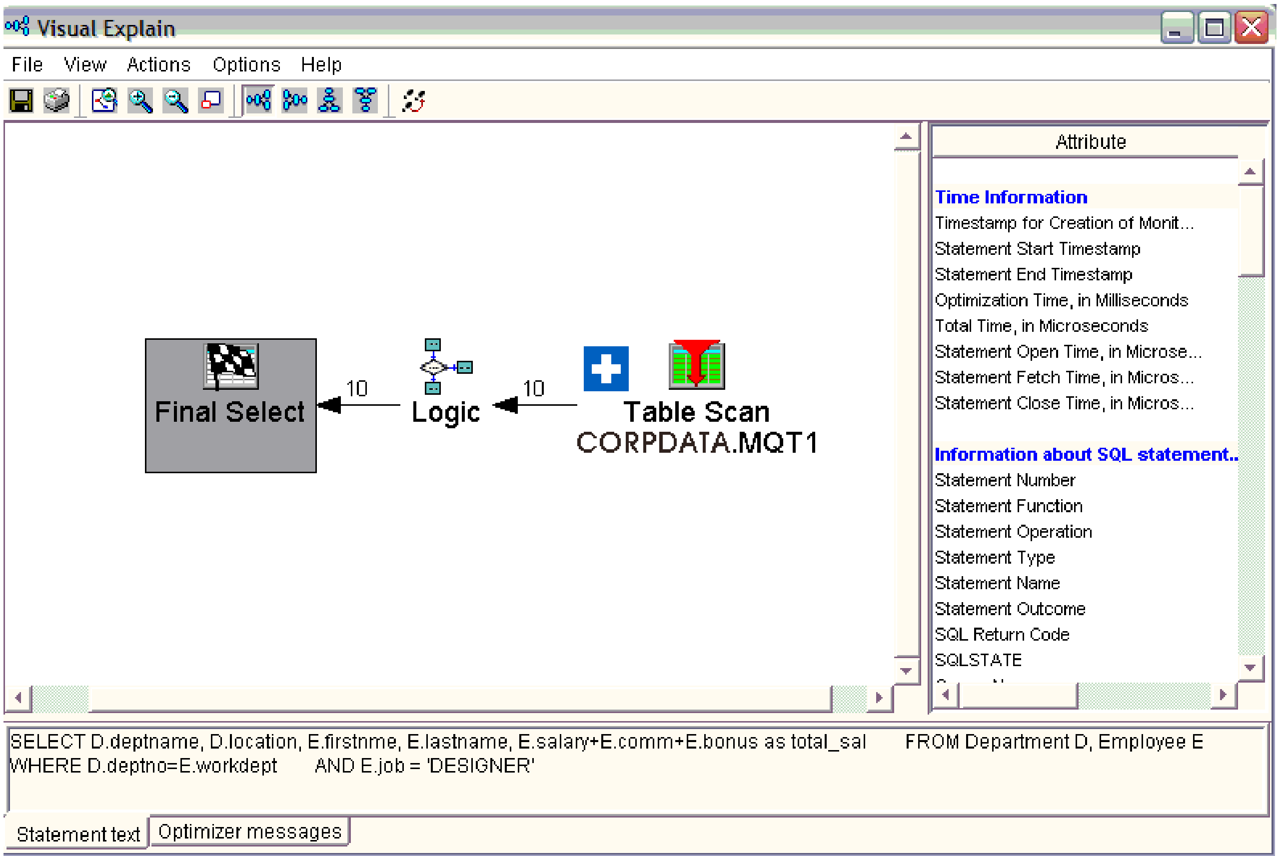
```
CREATE TABLE MQT1
         AS (SELECT D.deptname, D.location, E.firstnme, E.lastname, E.salary, E.comm, E.bonus, E.job
       FROM Department D, Employee E
       WHERE D.deptno=E.workdept)
 DATA INITIALLY IMMEDIATE  REFRESH DEFERRED
  ENABLE QUERY OPTIMIZATION
  MAINTAINED BY USER
```

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

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

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

Visual Explain diagram of the query when using the MQT:



Example 2

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

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

Create a table, MQT2, that uses this query:

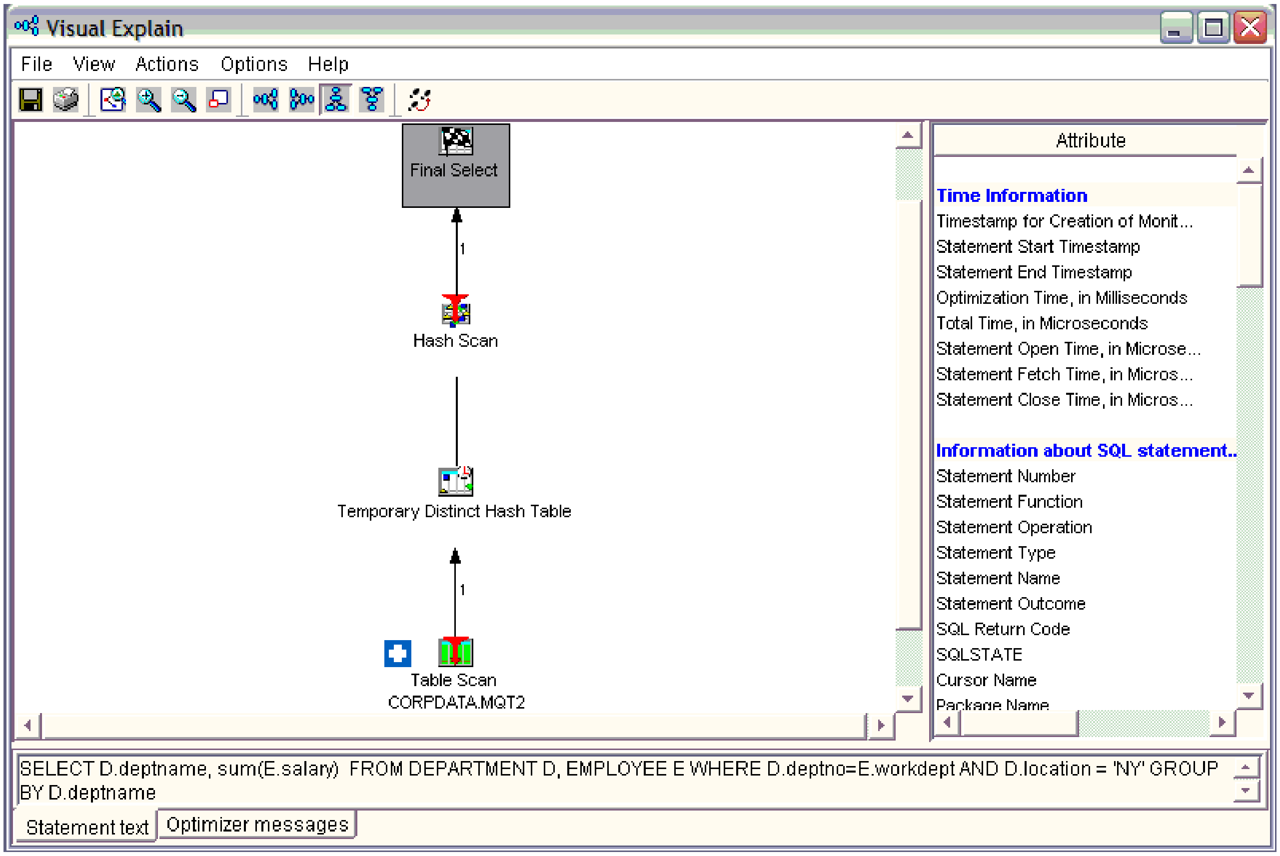
```
CREATE TABLE MQT2
        AS (SELECT D.deptname, D.location, sum(E.salary) as sum_sal
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.deptno=E.workdept
GROUP BY D.Deptname, D.location)
DATA INITIALLY IMMEDIATE  REFRESH DEFERRED
 ENABLE QUERY OPTIMIZATION
 MAINTAINED BY USER
```

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

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

Since the MQT may 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 less tables than the query:

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

Create an MQT based on the query above:

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

The rewritten query looks like this:

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

All predicates specified in the MQT, must also be specified in the query. The query may contain additional predicates. Predicates specified in the MQT must match exactly the predicates in the query. Any additional predicates specified in the query, but not in the MQT must be able to be derived from columns projected from the MQT. See previous example 1.

Example 4

Set the total salary for all departments that are located in ’NY’.

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

Create an MQT based on the query above:

```
CREATE TABLE MQT4
        AS (SELECT D.deptname, D.location, sum(E.salary) as sum_sal
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.deptno=E.workdept AND D.location = 'NY'
GROUP BY D.deptnamet, 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 is the same as the constant in the MQT; therefore the MQT can be used.

The MQT matching algorithm will also attempt to match where the predicates between the MQT and the query are not exactly 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 will be 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 query above:

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

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

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

The regroup new expression/aggregation rules are:

Table 24. 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)
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)

Table 24. Expression/aggregation rules for MQTs (continued)

Query	MQT	Final query
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)

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

If the FETCH FIRST N ROWS clause is specified in the MQT, then a FETCH FIRST N ROWS clause must also be specified in the query and the number of rows specified for the MQT must be greater than or equal to the number of rows specified in the query. It is not recommended that a MQT contain the FETCH FIRST N ROWS clause.

The ORDER BY clause on the MQT can be used to order the data in the MQT if a REFRESH TABLE is run. It is ignored during MQT matching and if the query contains an ORDER BY clause, it will be part of the rewritten query.

Related reference

"MQT supported function" on page 64
Although a MQT can contain almost any query, the optimizer only supports a limited set of query functions when matching MQTs to user specified queries. The user specified query and the MQT query must both be supported by the SQE optimizer.

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 iSeries Navigator and i5/OS functionality.

To assist you in tuning your performance, this function now produces statistics on MQT usage in a query. To access this through the iSeries Navigator, navigate to: **Database** → **Schemas** → **Tables**. Right-click your table and select **Show Materialized Query Tables**.

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

In addition to all existing attributes of an MQT, two new fields have been added to the iSeries Navigator.

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

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

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

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 where data in one results row has a natural relationship (call it a parent, child relationship) with data in another row or rows. Although the kinds of recursion implemented in these systems can be performed by using SQL Stored Procedures and temporary results tables, 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 and an iterative fullselect that 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 and those restrictions can be found in the SQL Programming. A key restriction is that query functions like grouping, aggregation or distinct that require a materialization of all the qualifying records before performing the function cannot be allowed within the iterative fullselect itself and must be requested 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) and the number of connections and total cost to arrive at that final destination.

Because this example uses the recursion process to also accumulate information like the running cost and number of connections, four values are actually put on 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 then 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 above query. It seeds the rows that will 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 above query. It contains a single reference in the FROM clause to the destinations recursive common table expression and will source 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, if you want a limit of at most 3 connecting flights, a local predicate using the accumulating connection count, r.connects<=3, can be specified.

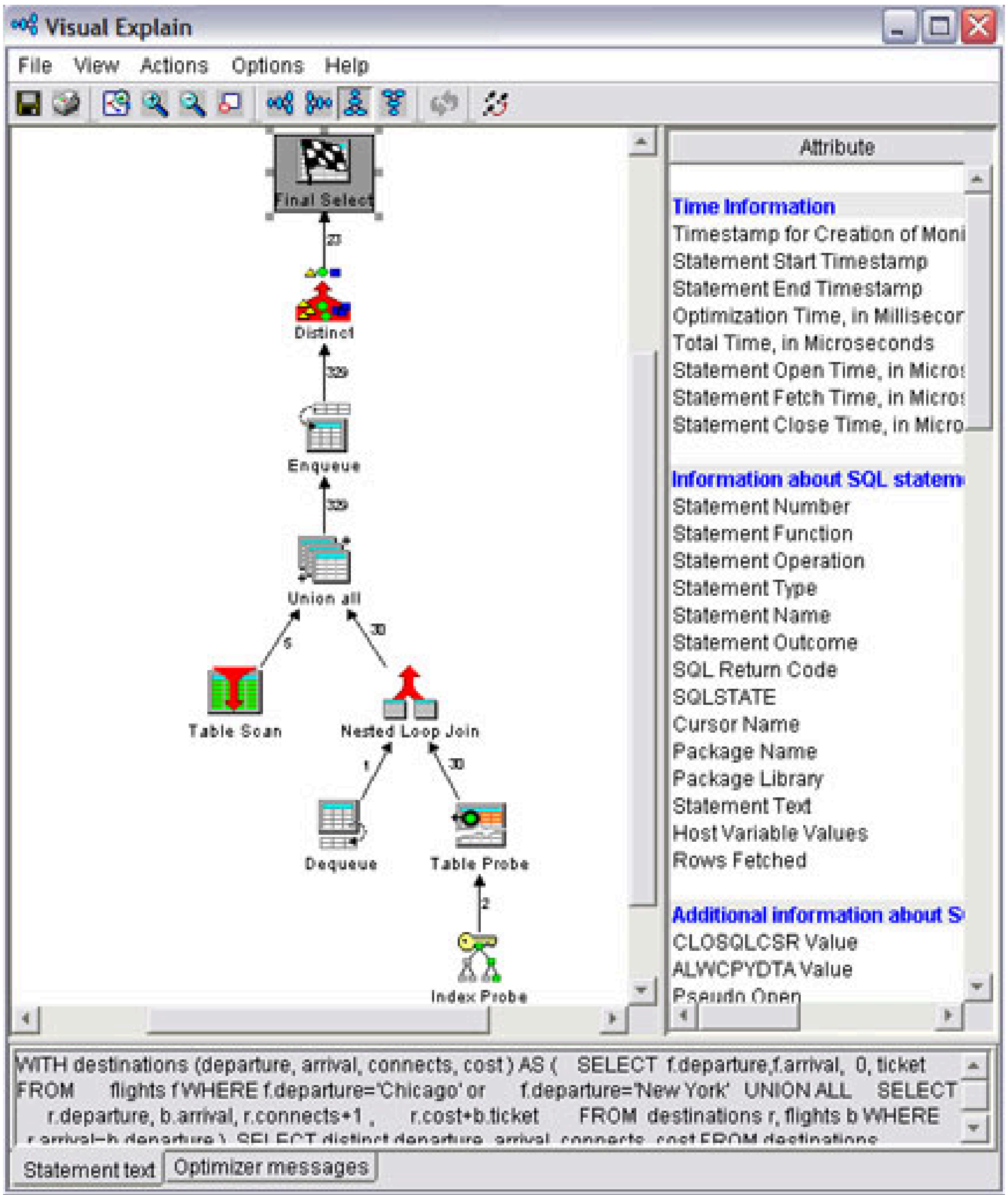
```
SELECT
  r.departure, b.arrival, r.connects + 1 ,
  r.cost + b.ticket
  FROM  destinations r, flights b
 WHERE r.arrival=b.departure
```

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 will be 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 and 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. Because the recursive queue feeds the recursion process and holds transient data, the join between the dequeue of these queue entries and the rest of the fullselect tables will always be a constrained join, with the queue on the left.



Multiple initialization and iterative fullselects

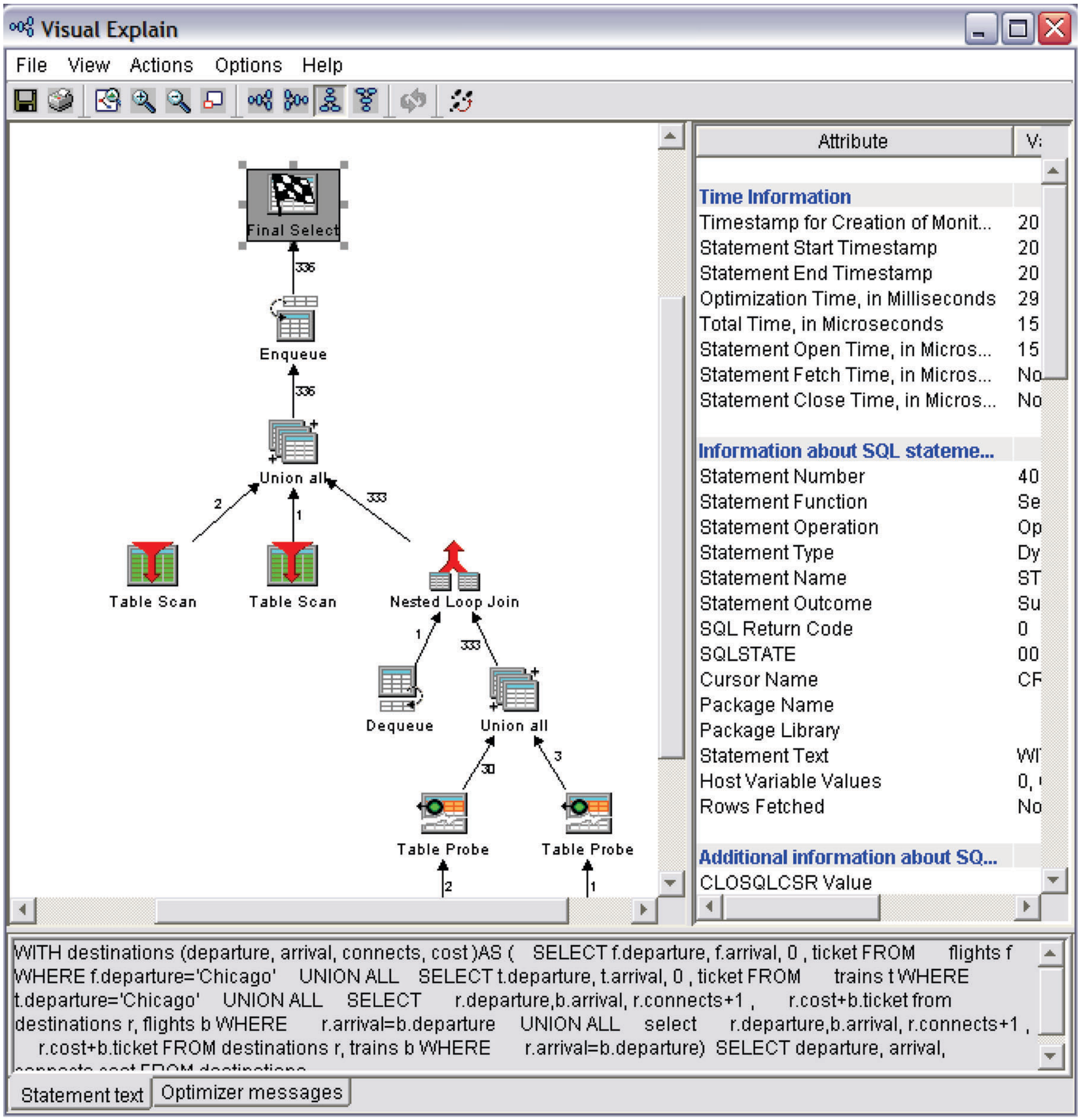
The use of multiple initialization and iterative fullselects specified in the recursive query definition allows for a multitude of data sources and separate selection requirements to feed the recursion process.

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;
```

As 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 fullselect first and then using a single queue feed those same rows and all other row results 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 will run first and the iterative fullselects will share equal access to the contents of the queue.



Predicate Pushing

When processing most queries with a 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 in to the defined recursive part of the query (through the Union ALL), however, may considerably alter the process of recursion itself. So as a general rule, the Union All specified in a recursive query is currently a predicate fence and 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.

If the intent of the query is to find all destinations accessible from ’Chicago’ but do not include the final destinations of ’Dallas’, pushing the ″arrival<>’Dallas’″ predicate in to the recursive query alters the output of the intended results, preventing the output of final destinations that are not ’Dallas’ but 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 may greatly decrease the amount 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 as they can incorrectly alter the recursive results if pushed in to a recursive definition or can cause unnecessary rows to be materialized and then rejected when a local predicate may legitimately help limit the recursion.

Specifying SEARCH consideration

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

Using a queuing (First In First Out) mechanism to keep track of 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 is an implementation distinction, however, and not a guarantee. Applications may want to guarantee how the data is retrieved. Some applications may 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 along with naming the recursive join value, identifying a set sequence column and providing the sequence column in an outer ORDER BY clause, the results will be output in depth or breadth first order. Note this is ultimately a relationship sort and not a value based sort.

Here is the example above output in depth 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. In the case of 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. In the case of DEPTH FIRST, the query engine needs to represent the entire ancestry of join values leading up to the current row and puts that information in a queue entry. Also, because these sort values are not coming from a external data source, the implementation for the sort will always be a temporary sorted list (no indexes possible).

Do not use the SEARCH option if you do not have a requirement that your data be materialized in a depth or breadth first manner as 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.

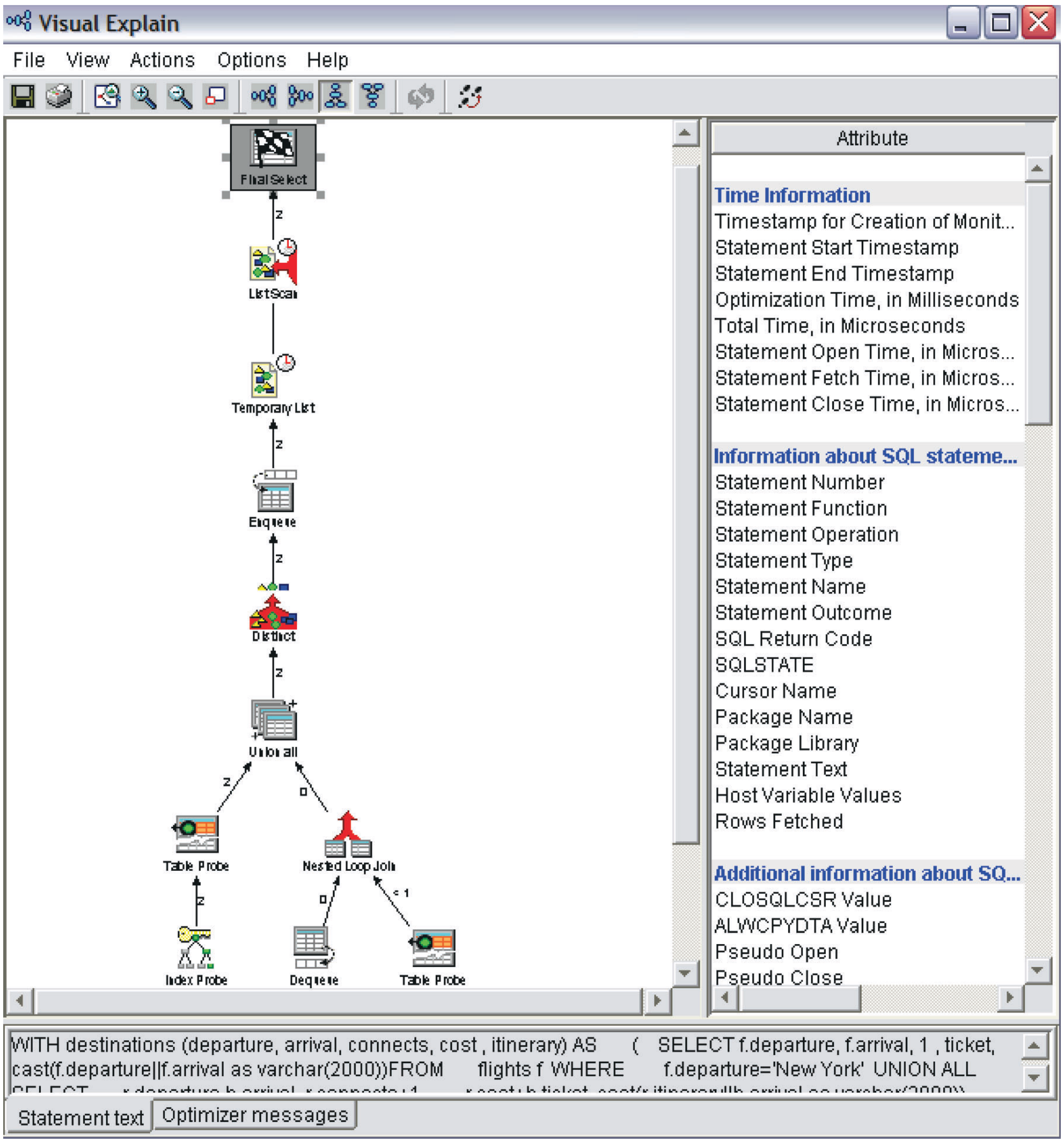
The SQL architecture allow for the optional checking for cyclic data and will discontinue the repeating cycles at that point. This additional checking is done by the use of the CYCLE option. The correct join recursion value must be specified on the CYCLE request and a cyclic indicator must be specified. Note that the cyclic indicator may 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 to 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, the query engine uses a compressed representation of the recursion values on the ancestry chain so that the query engine can 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.



SMP and recursive queries

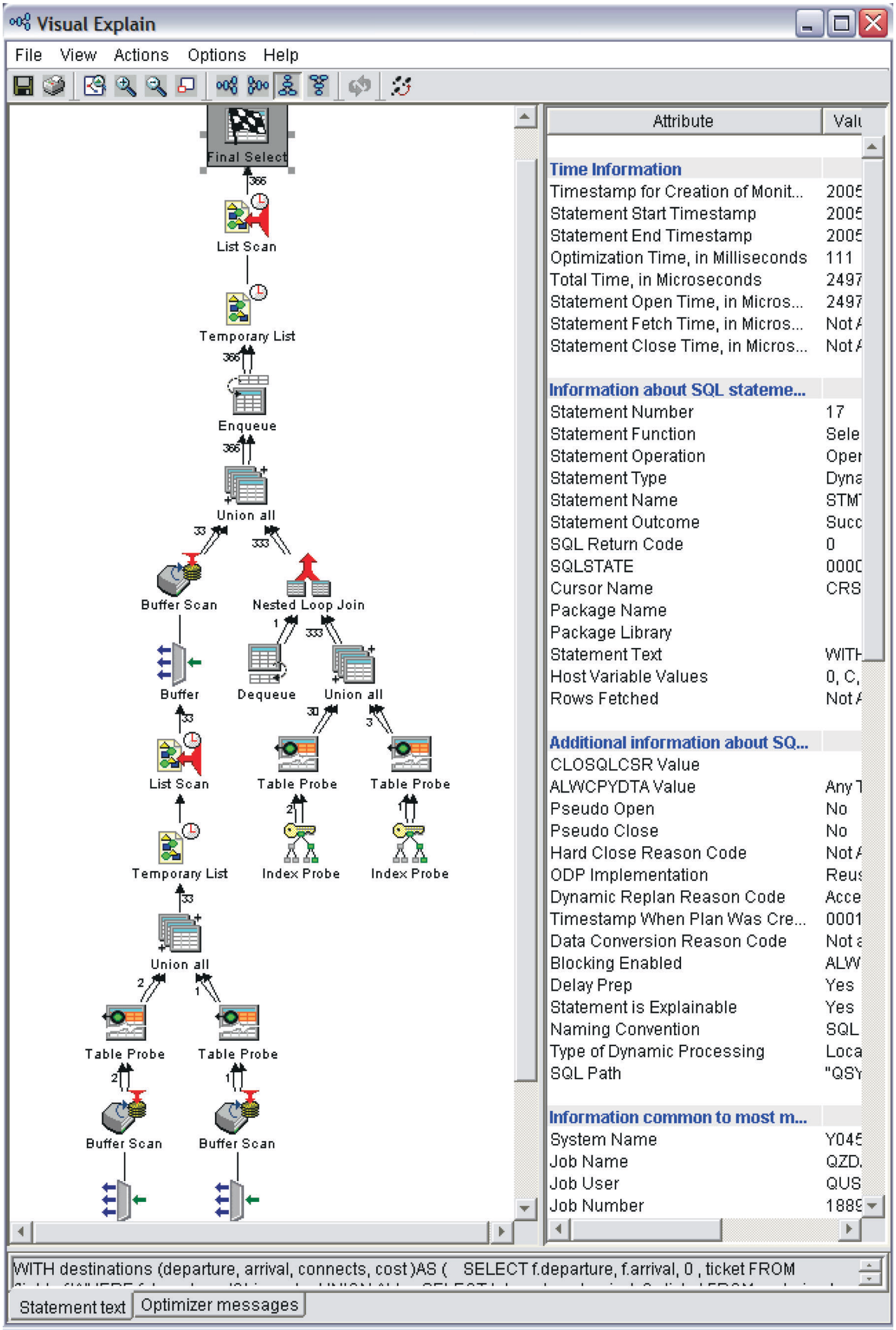
Recursive queries can benefit as much from symmetric multiprocessing (SMP) as do other queries on the system.

Recursive queries and parallelism however present some unique requirements. Because the initialization fullselect of a recursive query (the fullselect that seeds the initial values of the recursion), 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 do this is to have all the threads share the same queue allowing a thread to enqueue a new recursive key value just as a waiting thread is there to dequeue that request. A shared queue allow 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. Illustrated in the query below, where there are two fullselects that seed the recursion, a buffer is provide 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 illustrates the plan for the following query run with CHGQRYA DEGREE(*NBRTASKS 4). It illustrates that the results of the multiple initialization fullselects are buffered up and that 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) are putting their results in to a Temporary List object which become 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
Final Select
List Scan
Temporary List
Enqueue
Union all
Buffer Scan
Nested Loop Join
Buffer
Dequeue
Union all
List Scan
Table Probe
Table Probe
Temporary List
Index Probe
Index Probe
Union all
Table Probe
Table Probe
Buffer Scan
Buffer Scan
Attribute
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...
CLOSQLCSR Value
ALWCPYDTA Value
Pseudo Open
Pseudo Close
Hard Close Reason Code
ODP Implementation
Dynamic Replan Reason Code
Timestamp When Plan Was Cre...
Data Conversion Reason Code
Blocking Enabled
Delay Prep
Statement is Explainable
Naming Convention
Type of Dynamic Processing
SQL Path
Information common to most m...
System Name
Job Name
Job User
Job Number
WITH destinations (departure, arrival, connects, cost)AS (SELECT f.departure, f.arrival, 0 , ticket FROM
Statement text
Optimizer messages

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.

Verify the performance of SQL applications

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

The commands that can help you to verify performance are:

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

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

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

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, the access plans used during the running of the statement, and a list of the command parameters used to precompile the source member for the object.

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 and 20 class files and 20 service programs to debug mode. It also specifies certain attributes of the debugging session. For example, it can specify whether database files in production libraries can be updated while in debug mode.

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

Examine query optimizer debug messages in the job log

Query optimizer debug messages issue informational messages to the job log about the implementation of a query. These messages explain what happened during the query optimization process.

For example, you can learn:

- Why an index was or was not used

- Why a temporary result was required
- Whether joins and blocking are used
- What type of index was advised by the optimizer
- Status of the job's queries
- Indexes used
- Status of the cursor

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

Viewing debug messages using STRDBG command:

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

```
STRDBG PGM(Schema/program) UPDPROD(*YES)
```

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

Viewing debug messages using QAQQINI table:

You can also set the QRYOPTLIB parameter on the Change Query Attributes (CHGQRYA) command to a user schema where the QAQQINI table exists. Set the parameter on the QAQQINI table to MESSAGES_DEBUG, and set the value to *YES. This option places query optimization information in the job log. Changes made to the QAQQINI table are effective immediately and will affect all users and queries that use this table. Once you change the MESSAGES_DEBUG parameter, all queries that use this QAQQINI table will write debug messages to their respective joblogs. Pressing F10 from the command Entry panel displays the message text. To see the second-level text, press F1 (Help). The second-level text sometimes offers hints for improving query performance.

Viewing debug messages in Run SQL Scripts:

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

Viewing debug messages in Visual Explain:

In Visual Explain, debug messages are always available. You do not need to turn them on or off. Debug messages appear in the lower portion of the window. You can view detailed messages by double-clicking on a message.

Gather information about embedded SQL statements with the PRTSQLINF command

The Print SQL Information (PRTSQLINF) command returns information about the embedded SQL statements in a program, SQL package (the object normally used to store the access plan for a remote query), or service program. This information is then stored in a spooled file.

PRTSQLINF provides information about:

- The SQL statements being executed
- The type of access plan used during execution. This includes information about how the query will be implemented, the indexes used, the join order, whether a sort is done, whether a database scan is sued, 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 similar to the information that you can get from debug messages. However, while query debug messages work at runtime, PRTSQLINF works retroactively. You can also see this information in the second level text of the query governor inquiry message CPA4259.

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

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

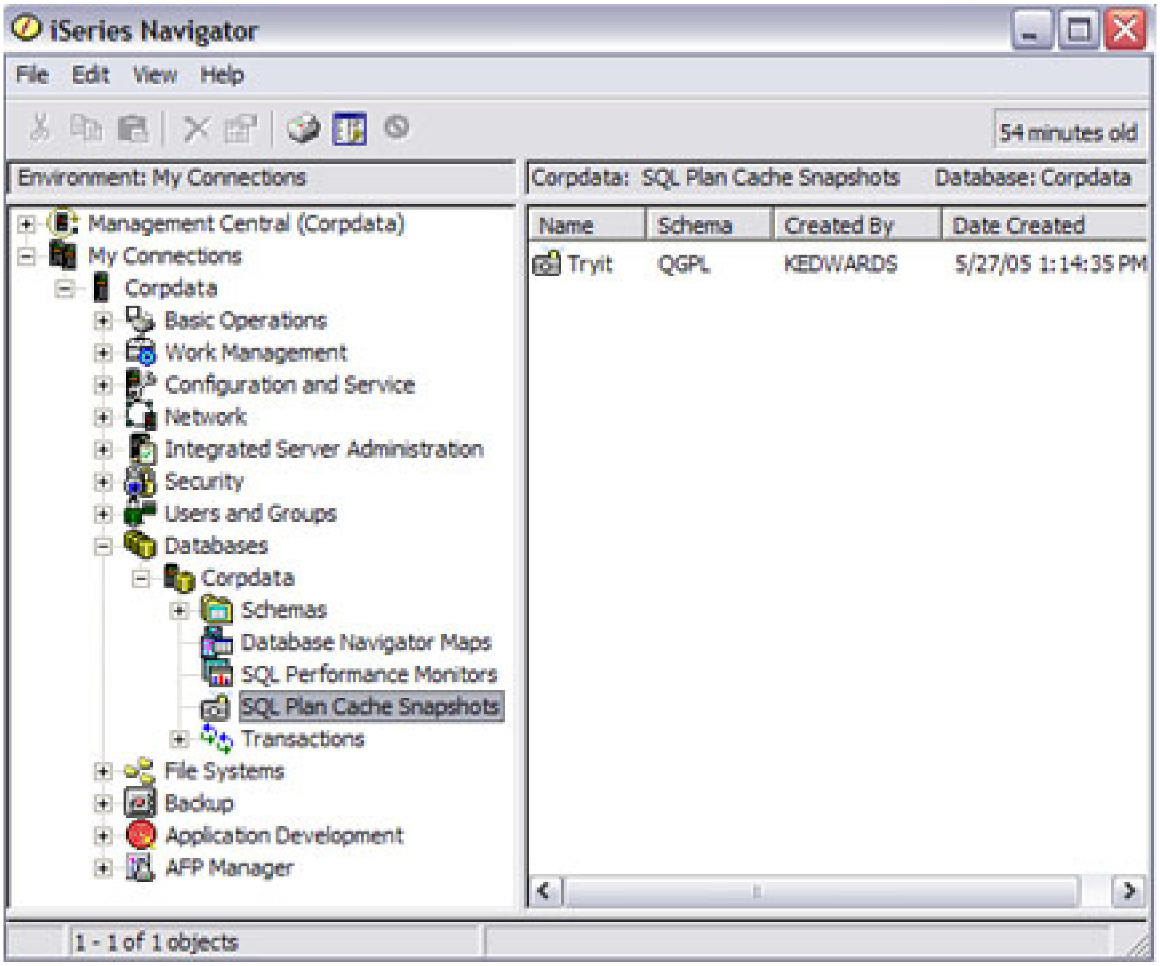
Related information

Print SQL Information (PRTSQLINF) command

Viewing the plan cache with iSeries Navigator

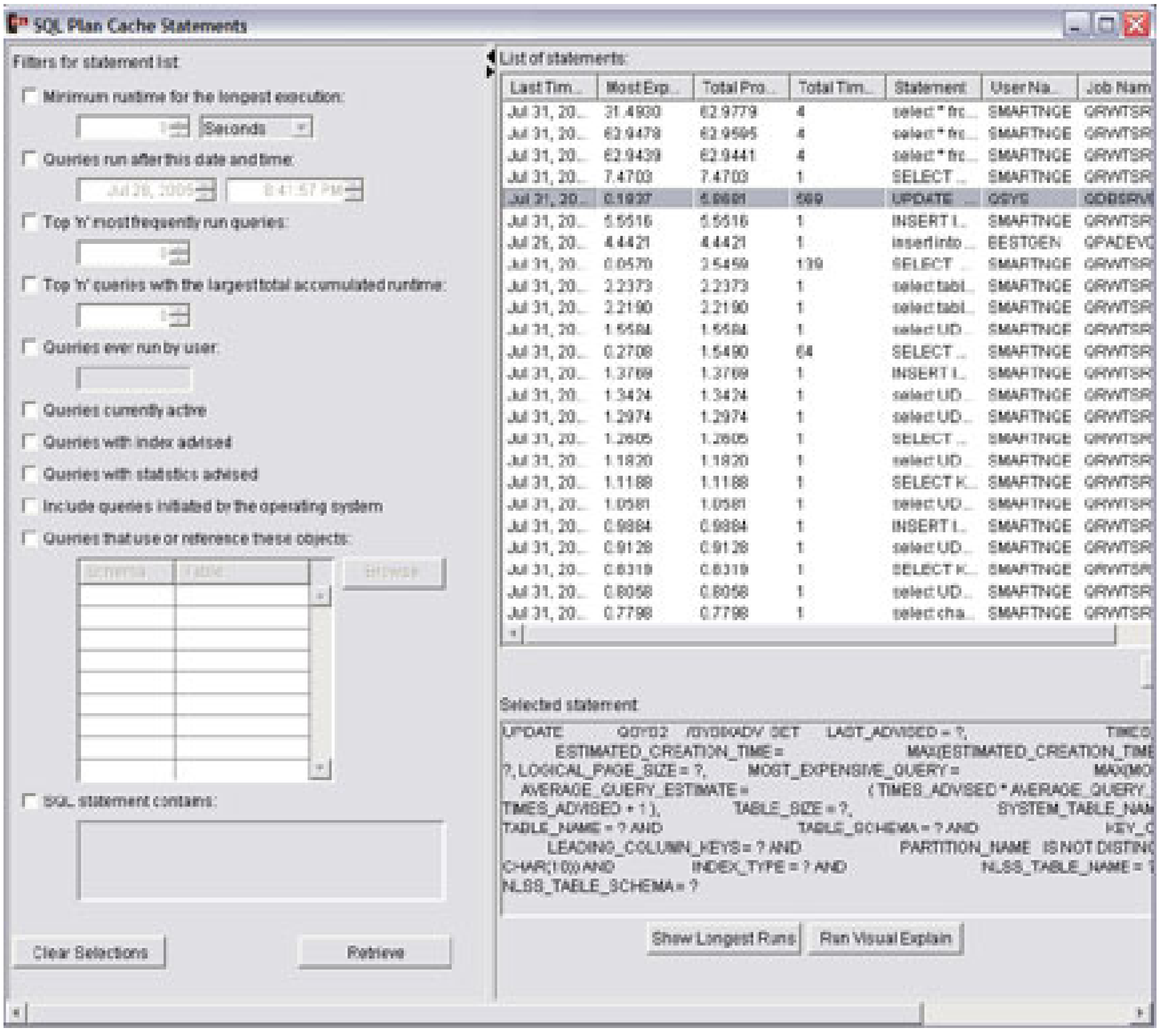
The Plan Cache contains a wealth of information about the SQE queries being run through the database. Its contents are viewable through the iSeries Navigator GUI interface.

This Plan Cache interface provides a window into the database query operations on the system. The interface to the Plan Cache resides under the **iSeries Navigator** → **system name** → **Database**.



Clicking the SQL Plan Cache folder shows a list of any snapshots gathered so far. A snapshot is a database monitor file generated from the plan cache and can be treated very much the same as the SQL Performance Monitors list. The same analysis capability exists for snapshots as exists for traditional SQL performance monitors.

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 brings up a screen with filtering capability. This screen provides a direct view of the current plan cache on the system.



Note that the retrieve action needs to be performed (pushed) to fill the display. The information shown shows the SQL query text, the last time the query was run, the most expensive single instance run of the query, total processing time consumed by the query, total number of times the query has been run and information about the user and job that first created the plan entry. It also shows how many times (if any) that the database engine was able to reuse the results of a prior run of the query to avoid rerunning the entire query.

The screen also 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 will shorten the time it takes to show the results. The list of queries that is returned is ordered by default so that those consuming the most processing time are shown at the top. You can reorder the results by clicking on the column heading for which you want the list ordered. Repeated clicking toggles the order from ascending to descending. When an individual entry is chosen, more detailed information about that entry can be seen. **Show Longest Runs** shows details of up to ten of the longest running instances of that query. **Run Visual Explain** can also be performed against the chosen query to show the details of the query plan. Finally, if one or more entries are highlighted, a snapshot (database performance monitor file) for those selected entries can be generated.

The information presented can be used in multiple ways to help with performance tuning. For example, Visual Explain of key queries can be utilized to show advice for creating an index to improve those queries. Alternatively, the longest running information can be used to determine if the query is being run during a heavy utilization period and can potentially be rescheduled to a more opportune time.

One item to note is that the user and job name information given for each entry is the user and job that initially caused the creation of the cached entry (the user where full optimization took place). This is not necessarily the same as the last user to run that query.

The filtering options provide a way to focus in on a particular area of interest:

Minimum runtime for the longest execution
Filter to those queries with at least one long individual query instance runtime

Queries run after this date and time
Filters to those queries that have been run recently

Top ’n’ most frequently run queries
Finds those queries run most often.

Top ’n’ queries with the largest total accumulated runtime
Shows the top resource consumers. This equates to the first n entries shown by default when no filtering is given. Specifying a value for n improves the performance of getting the first screen of entries, though the total entries displayed is limited to n.

Queries ever run by user
Provides a way to see the list of queries a particular user has run. Note that if this filter is specified, the user and job name information shown in the resulting entries still reflect the originator of the cached entry, which is not necessarily the same as the user specified on the filter.

Queries currently active
Shows the list of cached entries associated with queries that are still running or are in pseudo close mode. As with the user filtering, the user and job name information shown in the resulting entries still reflects the originator of the cached entry, which is not necessarily the same as the user currently running the query (there may be multiple users running the query).

Note: Current SQL for a job (right-click the Database icon) is an alternative for the viewing a particular job’s active query.

Queries with index advised
Limits the list to those queries where an index was advised by the optimizer to improve performance.

Queries with statistics advised
Limits the list to those queries where a statistic not yet gathered might have been useful to the optimizer if it was collected. The optimizer automatically gathers these statistics in the background, so this option is normally not that interesting unless, for whatever reason, you want to control the statistics gathering yourself.

Include queries initiated by the operating system
includes into the list the ’hidden’ queries initiated by the database itself behind the scenes to process a request. By default the list only includes user initiated queries.

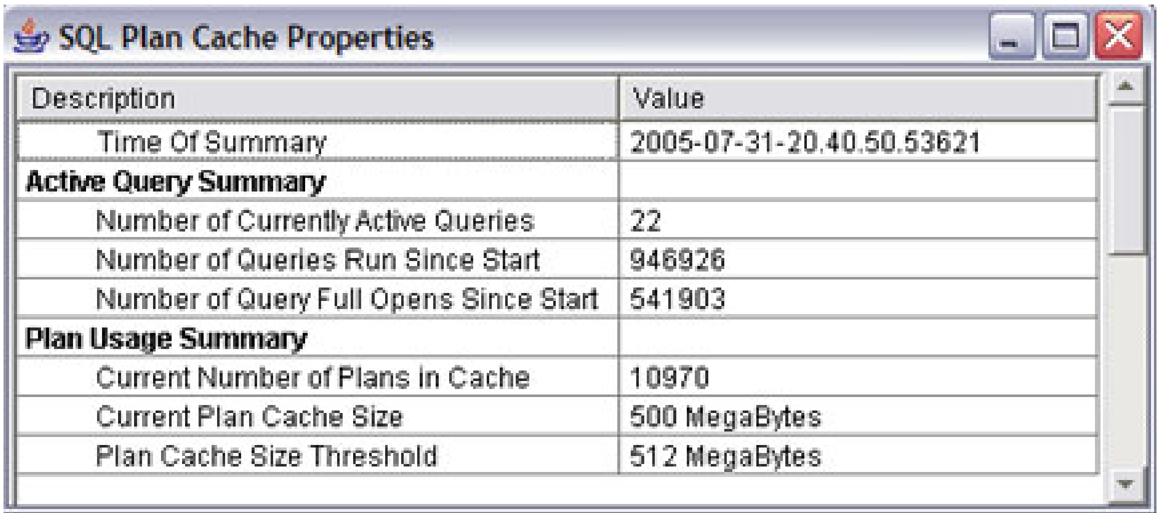
Queries that use or reference these objects
Provides a way to limit the entries to those that referenced or use the table(s) or index(s) specified.

SQL statement contains
Provides a wildcard search capability on the SQL text itself. It is useful for finding particular types of queries. For example, queries with a FETCH FIRST clause can be found by specifying ‘fetch’. The search is case insensitive for ease of use. For example, the string ’FETCH’ will find the same entries as the search string ’fetch’.

Multiple filter options can be specified. Note that in a multi-filter case, the candidate entries for each filter are computed independently and only those entries that are present in all the candidate lists are shown. So, for example, if you specified options **Top ’n’ most frequently run queries** and **Queries ever**

run by user, you will be shown those most-run entries in the cache that happen to have been run at some point by the specified user. You will not necessarily be shown the most frequently run queries run by the user (unless those queries also happen to be the most frequently run queries in the entire cache).

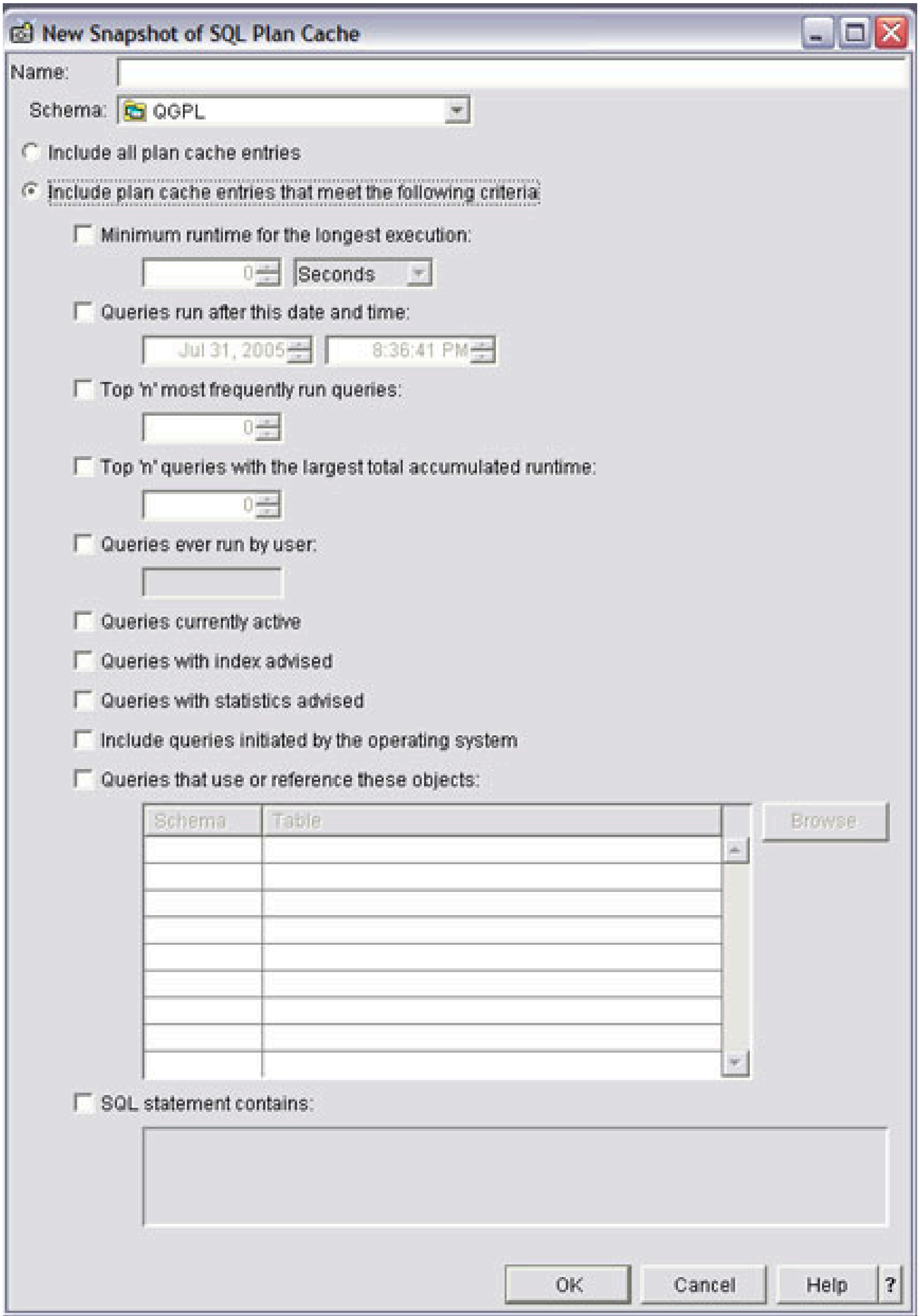
The **SQL Plan Cache → Properties** option shows high level information about the cache, including for example, cache size, number of plans, number of full open and pseudo opens that have occurred.


SQL Plan Cache Properties

Description	Value
Time Of Summary	2005-07-31-20.40.50.53621
Active Query Summary	
Number of Currently Active Queries	22
Number of Queries Run Since Start	946926
Number of Query Full Opens Since Start	541903
Plan Usage Summary	
Current Number of Plans in Cache	10970
Current Plan Cache Size	500 MegaBytes
Plan Cache Size Threshold	512 MegaBytes

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.

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.



The same filtering options are provided here as on the Show Statements screen.

The stored procedure, qsys2.dump_plan_cache, provides the simplest 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');
```

Note that 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, consider implementing a method of performing periodic snapshots of the plan cache to capture trends and heavy usage periods. The APIs described above, used in conjunction with job scheduling (for example), can be used to programmatically perform periodic snapshots.

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.

Monitoring your queries using memory-resident database monitor

The Memory-Resident Database Monitor is a tool that provides another method for monitoring database performance. This tool is only intended for SQL performance monitoring and is useful for programmers and performance analysts. The monitor, with the help of a set of APIs, takes database monitoring information and manages them for the user in memory. This memory-based monitor reduces CPU overhead as well as resulting table sizes.

The Start Database Monitor (STRDBMON) can constrain server resources when collecting performance information. This overhead is mainly attributed to the fact that performance information is written directly to a database table as the information is collected. The memory-based collection mode reduces the server resources consumed by collecting and managing performance results in memory. This allows the monitor to gather database performance statistics with a minimal impact to the performance of the server as whole (or to the performance of individual SQL statements).

The monitor collects much of the same information as the STRDBMON monitor, but the performance statistics are kept in memory. At the expense of some detail, information is summarized for identical SQL statements to reduce the amount of information collected. The objective is to get the statistics to memory as fast as possible while deferring any manipulation or conversion of the data until the performance data is dumped to a result table for analysis.

The memory-based monitor is not meant to replace the STRDBMON monitor. There are circumstances where the loss of detail in the monitor will not be sufficient to fully analyze an SQL statement. In these cases, the STRDBMON monitor should still be used.

The memory-based monitor manages the data in memory, combining and accumulating the information into a series of row formats. This means that for each unique SQL statement, information is accumulated from each run of the statement and the detail information is only collected for the most expensive statement execution.

Each SQL statement is identified by the monitor according to the following:

- statement name
- package (or program)
- schema that contains the prepared statement
- cursor name that is used

For pure dynamic statements, the statement text is kept in a separate space and the statement identification will be handled internally via a pointer.

While this system avoids the significant overhead of writing each SQL operation to a table, keeping statistics in memory comes at the expense of some detail. Your objective should be to get the statistics to memory as fast as possible, then reserve time for data manipulation or data conversion later when you dump data to a table.

The memory-based monitor manages the data that is in memory by combining and accumulating the information into the new row formats. Therefore, for each unique SQL statement, information accumulates from each running of the statement, and the server only collects detail information for the most expensive statement execution.

Each SQL statement is identified by the monitor by the statement name, the package (or program) and schema that contains the prepared statement and the cursor name that is used. For pure dynamic statements:
- Statement text is kept in a separate space and
- Statement identification is handled internally via a pointer.

API support for the memory-based monitor

A set of APIs enable support for the memory-based monitor. An API supports each of the following activities:
- Start the new monitor
- Dump statistics to tables
- Clear the monitor data from memory
- Query the monitor status
- End the new monitor

When you start the new monitor, information is stored in the local address space of each job that the system monitors. As each statement completes, the system moves information from the local job space to a common system space. If more statements are executed than can fit in this amount of common system space, the system drops the statements that have not been executed recently.

Related information

Start SQL Database Monitor (QQQSSDBM) API

Dump SQL Database Monitor (QQQDSDBM) API

Clear SQL Database Monitor Statistics (QQQCSDBM) API

Query SQL Database Monitor (QQQQSDBM) API

End SQL Database Monitor (QQQESDBM) API

Memory-resident database monitor external API description

The memory-resident database monitor is controlled by a set of APIs.

Table 25. External API Description

API	Description
Start SQL Database Monitor (QQQSSDBM)	API to start the SQL monitor
Clear SQL Database Monitor Statistics (QQQCSDBM)	API to clear SQL monitor memory
Dump SQL Database Monitor (QQQDSDBM)	API to dump the contents of the SQL monitor to table
End SQL Database Monitor (QQQESDBM) API	API to end the SQL monitor
Query SQL Database Monitor (QQQQSDBM)	API to query status of the database monitor

Memory-resident database monitor external table description

The memory resident database monitor uses its own set of tables instead of using the single table with logical files that the STRDBMON monitor uses. The memory resident database monitor tables closely match the suggested logical files of the STRDBMON monitor.

Table 26. External table Description

Monitor table	Description
QAQQQRYI	Query (SQL) information
QAQQTEXT	SQL statement text
QAQQ3000	Table scan
QAQQ3001	Index used
QAQQ3002	Index created
QAQQ3003	Sort
QAQQ3004	Temporary table
QAQQ3007	Optimizer time out/ all indexes considered
QAQQ3008	Subquery
QAQQ3010	Host variable values
QAQQ3030	Materialized Query Tables considered

Sample SQL queries

As with the STRDBMON monitor, it is up to the user to extract the information from the tables in which all of the monitored data is stored. This can be done through any query interface that the user chooses.

If you are using iSeries Navigator with the support for the SQL Monitor, you have the ability to analyze the results direct through the graphical user interface. There are a number of shipped queries that can be used or modified to extract the information from any of the tables. For a list of these queries, go to Common queries on analysis of DB Performance Monitor data the DB2 UDB for iSeries website .

Memory-resident database monitor row identification

The join key column QQKEY simplifies the joining of multiple tables together. This column replaces the join field (QQJFLD) and unique query counters (QQCNT) that the database monitor used. The join key column contains a unique identifier that allows all of the information for this query to be received from each of the tables.

This join key column does not replace all of the detail columns that are still required to identify the specific information about the individual steps of a query. The Query Definition Template (QDT) Number or the Subselect Number identifies information about each detailed step. Use these columns to identify which rows belong to each step of the query process:

- QQQDTN - Query Definition Template Number
- QQQDTL - Query Definition Template Subselect Number (Subquery)
- QQMATN - Materialized Query Definition Template Number (View)
- QQMATL - Materialized Query Definition Template Subselect Number (View w/ Subquery)
- QQMATULVL - Materialized Query Definition Template Union Number (View w/Union)

Use these columns when the monitored query contains a subquery, union, or a view operation. All query types can generate multiple QDT's to satisfy the original query request. The server uses these columns to separate the information for each QDT while still allowing each QDT to be identified as belonging to this original query (QQKEY).

Using iSeries Navigator with summary monitors

You can work with summary monitors from the iSeries Navigator interface. A summary monitor creates a Memory-Resident Database monitor (DBMon), found on the native interface.

As the name implies, this monitor resides in memory and only retains a summary of the data collected. When the monitor is paused or ended, this data is written to a hard disk and can be analyzed. Because the monitor stores its information in memory, the performance impact to your system is minimized. However, you do lose some of the detail.

Starting a summary monitor

You can start a summary monitor from the iSeries Navigator interface.

You can start this monitor by right-clicking SQL Performance Monitors under the Database portion of the iSeries Navigator tree and selecting **New → SQL Performance Monitor**. In the monitor wizard, select **Summary**.

When you create a summary monitor, certain kinds of information are always collected. This information includes summary information, SQL statement information, and host variable information. You can also choose to collect the following types of information:

Table scans and arrival sequences
: Select to include information about table scan data for the monitored jobs. Table scans of large tables can be time-consuming. If the SQL statement is long running, it may indicate that an index might be necessary to improve performance.

Indexes used
: Select to include information about how indexes are used by monitored jobs. This information can be used to quickly tell if any of the permanent indexes were used to improve the performance of a query. Permanent indexes are typically necessary to achieve optimal query performance. This information can be used to determine how often a permanent index was used by in the statements that were monitored. Indexes that are never (or very rarely) used should probably be dropped to improve the performance of inserts updates and deletes to a table. Before dropping the index, you may want to determine if the index is being used by the query optimizer as a source of statistics.

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

Data sorts
: Select to include information about data sorts that monitored jobs perform. Sorts of large result sets in an SQL statement may be a time consuming operation. In some cases, an index can be created that will eliminate the need for a sort.

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

Indexes considered
: Select to include information about which indexes were considered for the monitored jobs. This information can help to determine if an index is used in the query. If an index was considered, but not used, you might need to rewrite the index or drop it. Before dropping the index, you may want to determine if the index is being used by the query optimizer as a source of statistics.

Subselect processing
: Select to include information about subselect processing. This information can indicate which subquery in a complex SQL statement is the most expensive.

You can choose which jobs you want to monitor or choose to monitor all jobs. You can have multiple instances of monitors running on you system at one time. For summary monitors, only one monitor instance can be monitoring all jobs. Additionally, you cannot have two monitors monitoring the same job. When collecting information for all jobs, the monitor will collect on previously started jobs or new jobs started after the monitor is created. You can edit this list by selecting and removing jobs from the **Selected jobs** list.

Related reference

"Determining unnecessary indexes" on page 148
You can easily determine which indexes are being used for query optimization.

Analyzing summary monitor information

Once data has been collected in the monitor, it can be analyzed.

You can analyze information in a summary monitory by right-clicking the summary monitor in the right pane and selecting **Analyze**. A summary monitor must be ended or paused in order to analyze the data.

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

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

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

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

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

Additionally, you can select more Detailed Results:

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

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

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

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

Index used information
Contains a row of information for each permanent index that an SQL statement used. This can be used to quickly tell if any of the permanent indexes were used to improve the performance of a query. Permanent indexes are typically necessary to achieve optimal query performance. This information can be used to determine how often a permanent index was used by in the statements that were monitored. Indexes that are never (or very rarely) used should probably be dropped to improve the performance of inserts updates and deletes to a table. Before dropping the index you may also want to look at the last used date in the Description information for the index.

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

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

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

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

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

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

Finally, you can select the Composite view.

Summary data
: Contains resource and other general information about monitored jobs.

Statement text
: Contains the SQL text that monitored jobs call.

Table scan
: Contains the table scan data for the monitored jobs.

Data sorts
: Contains details of data sorts that monitored jobs perform.

Host variable use
: Contains the values of host variables that monitored jobs use.

Optimizer time out/access paths considered
: Contains details of any occurrences of time outs of monitored jobs.

Indexes used
: Contains details of how indexes are used by monitored jobs.

Index creation
: Contains details of the creation of indexes by monitored jobs.

Subselect processing
: Contains information about each subselect in an SQL statement.

Temporary file use
: Contains details of temporary files that monitored jobs created.

Importing a monitor

You can import monitor data that has been collected using Start Database Monitor (STRDBMON) command or some other interface by using iSeries Navigator.

To import monitor data, right-click **SQL Performance monitors** and select **Import**. Once you have imported a monitor, you can analyze the data.

Monitoring your queries using Start Database Monitor (STRDBMON)

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 as they should, or whether they need fine tuning. Database monitors can generate significant CPU and disk storage overhead when in use.

You can gather performance information for a specific query, for every query on the server, or for a group of queries on the server. 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 each row's unique identification number.

What kinds of statistics you can gather

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

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

How you can use performance statistics

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

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

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

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 ENDDBMON 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 ten (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 may 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 will be written to the specified output 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, you will not get two copies of the database statistic records, one copy for the private monitor and one copy for the public monitor. You will get only one copy of the database statistic records.

If the monitor is started on all jobs (a public monitor), any jobs waiting on job queues or any jobs 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 server when the command is issued. Each job in the server 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, the user profile, the name of the table(s) being queried, the estimated run time of the query, the TCP/IP internet address, or any combination of those filters. Specifying a STRDBMON filter should help minimize the number of statistic records captured for any monitor.

Example 1: Starting Database Monitoring For All Jobs

```
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. Ten records will be held before being written to the file.

Example 2: Starting Database Monitoring For a Specific Job

```
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. Twenty records will be held before being written to the file.

Example 3: Starting Database Monitoring For a Specific Job 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 may exist in more than one independent auxiliary storage pool (ASP); the library in the name space of the originator's job will always be used.

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

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

This command starts database monitoring for all jobs 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 may exist in more than one independent auxiliary storage pool (ASP); the library in the name space of the originator's job will always be used. Note that because this is a public type monitor, so any QZDA jobs that are started will also have statistics records collected.

Example 5: Starting Database Monitoring For All Jobs 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 Database Monitoring For the Current® Job 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 TABLE1 in Library LIB41.

Example 7: Starting Database Monitoring For the Current Job and 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 8: Starting Database 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 of the following conditions:

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

Related information

Start Database Monitor (STRDBMON) command

End Database Monitor (ENDDBMON) command

The End Database Monitor (ENDDBMON) 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 joblog 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 ENDDBMON command.
- If JOB(*ALL) was specified on the Start Database Monitor (STRDBMON) command, the monitor can only be ended by specifying ENDDBMON JOB(*ALL). The monitor cannot be ended by specifying ENDDBMON JOB(*).

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

Database monitor examples

The iSeries navigator interface provides a powerful tool for gathering and analyzing performance monitor data using database monitor. However, you may 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 (ENDDBMON) 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 already exist, the command will create one from the skeleton table in QSYS/QAQQDBMN.

2. Run your application
3. ENDDBMON
4. Create views over LIB/PERFDATA using the SQL DDL. Creating the views is not mandatory. All of 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 should closely study the physical and logical view formats to understand all the data being collected so you can create queries that give the best information for your applications.

Related information

Start Database Monitor (STRDBMON) command

End Database Monitor (ENDDBMON) command

Database monitor performance analysis example 1:

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

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

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

A lot of data about many queries is contained in multiple rows in table LIB/PERFDATA. It is not uncommon for data about a single query to be contained in 10 or more rows within the table. The combination of defining the logical 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 27. Output for SQL Queries that Performed Table Scans

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

If the query does not use SQL, the SQL information row (QQQ1000) is not created. This makes it more difficult to determine which rows in LIB/PERFDATA pertain to which query. When using SQL, row QQQ1000 contains the actual SQL statement text that matches the 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.

Database monitor performance analysis example 2:

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

```
SELECT A.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.End_Timestamp - D.Start_Timestamp) AS TOT_TIME,
     D.Statement_Text_Long
   FROM LIB/QQQ3000 A INNER JOIN LIB/QQQ3014 B
     ON (A.Join_Column = B.Join_Column AND
     A.Unique_Count = B.Unique_Count)
     LEFT OUTER JOIN LIB/QQQ3019 C
     ON (A.Join_Column = C.Join_Column AND A.Unique_Count = C.Unique_Count)
     LEFT OUTER JOIN LIB/QQQ1000 D
     ON (A.Join_Column = D.Join_Column AND A.Unique_Count = D.Unique_Count)
```

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

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

Table 28. Output for All Queries that Performed Table Scans

Lib Name	Table Name	Total Rows	Index Advised	Query OPNID	ODP Open Time	Clock Time	Recs Rtned	Rows Rtned	TOT_ TIME	Statement Text

LIB1	**TBL1**	**20000**	**Y**		**1.1**	**4.7**	**10**	**10**	**6.2**	`SELECT *`
										`FROM LIB1/TBL1`
										`WHERE FLD1 = 'A'`

LIB1	**TBL2**	**100**	**N**		**0.1**	**0.7**	**100**	**100**	**0.9**	`SELECT *` `FROM LIB1/TBL2`

LIB1	**TBL1**	**20000**	**Y**		**2.6**	**4.4**	**32**	**32**	**7.1**	`SELECT *`
										`FROM LIB1/TBL1`
										`WHERE FLD1 = 'A'`
										`AND FLD2 > 9000`

LIB1	**TBL4**	**4000**	**N**	**QRY04**	**1.2**	**4.2**	**724**	*	*	*

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

Database monitor performance analysis example 3:

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

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

```
SELECT A.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 AND
     A.Unique_Count = B.Unique_Count)
     LEFT OUTER JOIN LIB/QQQ1000 C
     ON (A.Join_Column = C.Join_Column AND
     A.Unique_Count = C.Unique_Count)
   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 column Index_Advised_Columns that contains a list of possible key columns to use when creating the index suggested by the query optimizer. Second, the query selection limits the output to those table scan queries where the optimizer advises that an index be created (A.Index_Advised = 'Y'). The table below shows what the results might look like.

Table 29. Output with Recommended Key Columns

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

At this point you should determine whether it makes sense to create a permanent index as advised by the optimizer. In this example, creating one index over LIB1/TBL1 satisfies all three queries since each use a primary or left-most key column of FLD1. By creating one index over LIB1/TBL1 with key columns FLD1, FLD2, there is potential to improve the performance of the second query even more. The frequency these queries are run and the overhead of maintaining an additional index over the table should be considered when deciding whether to create the suggested index.

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

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

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

Additional database monitor examples:

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

1. How many queries are performing dynamic replans?

   ```
   SELECT COUNT(*)
     FROM   LIB/QQQ1000
     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/QQQ1000
  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/QQQ3002
  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/QQQ3002 A, LIB/QQQ1000 B
   WHERE A.Join_Column = B.Join_Column
     AND A.Unique_Count = B.Unique_Count
     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/QQQ3002 A INNER JOIN LIB/QQQ3014 B
     ON (A.Join_Column = B.Join_Column AND
     A.Unique_Count = B.Unique_Count)
   LEFT OUTER JOIN LIB/QQQ1000 C
     ON (A.Join_Column = C.Join_Column AND
     A.Unique_Count = C.Unique_Count)
   WHERE A.System_Table_Schema LIKE '%'
     AND A.System_Table_Name = '%'
```

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

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

```
SELECT 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/QQQ1000
   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/QQQ1000
   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/QQQ3014 A LEFT OUTER JOIN LIB/QQQ3019 B
     ON (A.Join_Column = B.Join_Column AND
     A.Unique_Count = B.Unique_Count)
```

```
    LEFT OUTER JOIN LIB/QQQ1000 C
      ON (A.Join_Column = C.Join_Column AND
      A.Unique_Count = C.Unique_Count)
    ORDER BY 1 DESC
```

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

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

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

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

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

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

11. Which SQL queries were implemented with nonreusable ODPs?

```
SELECT B.Statement_Text_Long
   FROM LIB/QQQ3010 A, LIB/QQQ1000 B
   WHERE A.Join_Column = B.Join_Column
     AND A.Unique_Count = B.Unique_Count
     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/QQQ3014
   WHERE Stopped_By_Query_Governor = 'Y'
```

Note: This example assumes 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/QQQ3014 A LEFT OUTER JOIN LIB/QQQ3019 B
     ON (A.Join_Column = B.Join_Column AND
     A.Unique_Count = B.Unique_Count)
   LEFT OUTER JOIN LIB/QQQ1000 C
     ON (A.Join_Column = C.Join_Column AND
     A.Unique_Count = C.Unique_Count)
   WHERE A.Estimated_Processing_Time/1000 >
     (A.Open_Time + B.Clock_Time_to_Return_All_Rows)
```

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

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

```
 SELECT COUNT(*)
   FROM   QQQ3014
   WHERE  Has_Union = 'Y'
```

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

15. You are a system administrator and an upgrade to the next release is planned. 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 will need to compare the statement text between the rows in the two tables to correlate the data.

Using iSeries Navigator with detailed monitors

You can work with detailed monitors from the iSeries Navigator interface. The detailed SQL performance monitor is the iSeries 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 iSeries Navigator tree and selecting **New** → **Monitor**. This monitor save detailed data in real time to a hard disk and does not need to be paused or ended in order to analyze the results. You can also choose to run a Visual Explain based on the data gather by the monitor. Since this monitor does save data in real time, it may have a performance impact on your system.

Starting a detailed monitor

You can start a detailed monitor from the iSeries Navigator interface.

You can start this monitor by right-clicking SQL Performance Monitors under the Database portion of the iSeries Navigator tree and selecting **New** → **SQL Performance Monitor**. On the monitor wizard, select **Detailed**.

When you create a detailed monitor, you can filter the information that you want to capture.

Minimum estimated query runtime
Select this to include queries that exceed a specified amount of time. Select a number and then a unit of time.

Job name
Select this 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*’ will find all jobs where the name starts with ’QZDAS.’

Job user
Select this 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*’ will find all user IDs where the name starts with ’QUSER.’

Current user
Select this 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*’ will find all users where the name starts with ’QSYS.’

Internet address
Select this to filter by Internet access. The format must be xxx.xxx.xxx.xxx. For example: 5.5.199.199.

Only queries that access these tables
Select this 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 ten table names can be specified.

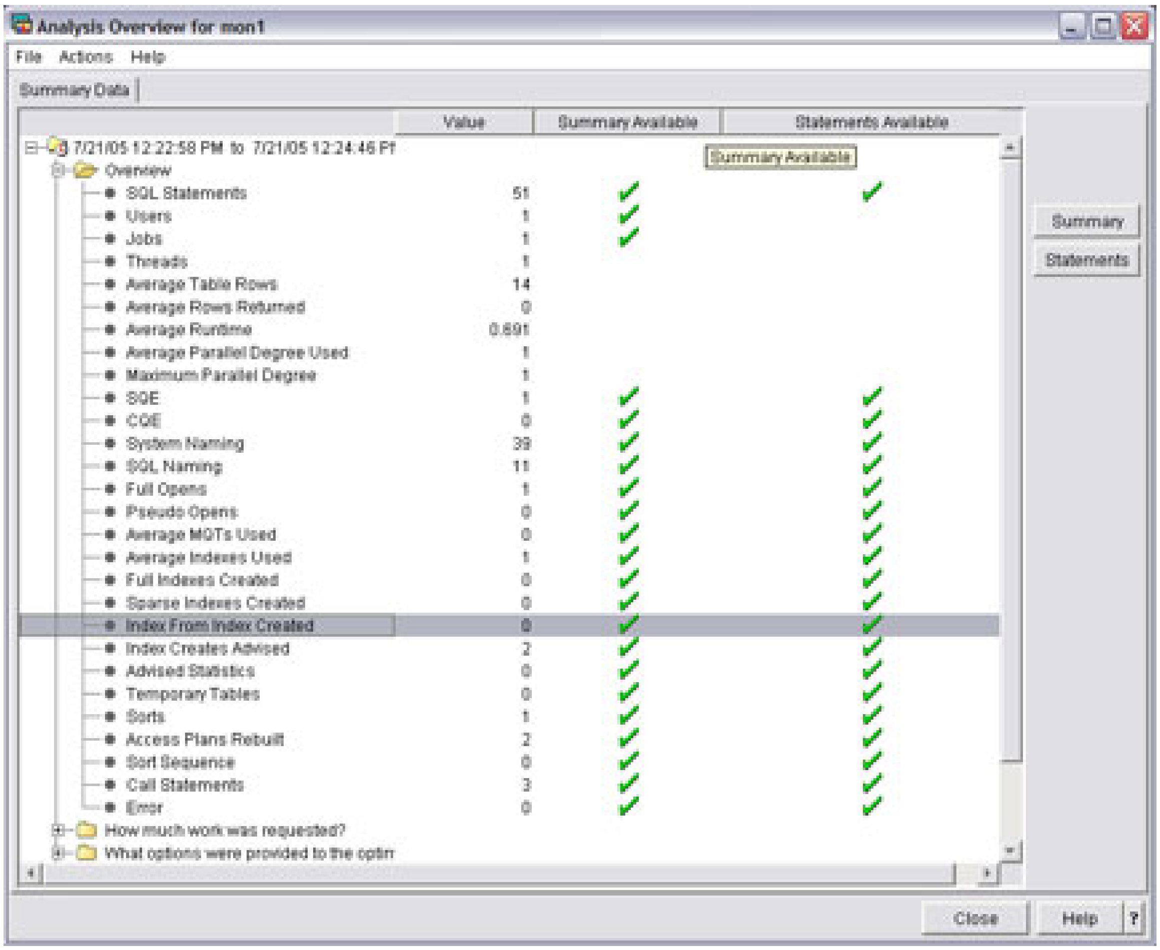
Activity to monitor
Select to collect monitor output for user-generated queries or for both user-generated and system-generated queries.

You can choose which jobs you want to monitor or choose to monitor all jobs. You can have multiple instances of monitors running on you 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.

Analyze 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.



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, and hence which ones are perhaps candidates for performance tuning. The user may then want to start a separate detailed performance monitor on an individual job to get more detailed information without having to monitor the entire system.

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

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

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 iSeries Navigator to compare data sets in two different monitors.

To compare data sets in different monitors, go to **iSeries Navigator** → *system name* → **SQL performance monitors**. Right-click a monitor in the right pane and select **Compare**.

On the Compare 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 compare to ignore. For example, consider the following scenario: You have an application running in a test schema and have it optimized. Now you move it to the production schema and you want to compare how it executes there. The statements in the compare 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 to ignore ″PROD″ in the second monitor so that the statements in the two monitors appear identical.

Compare 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 will be 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**.

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
: Filter to those queries with at least one long individual query instance runtime

Queries run after this date and time
: Filters to those queries that have been run recently

Queries that use or reference these objects
: Provides a way to limit the entries to those that referenced or use the table(s) or index(s) specified.

SQL statement contains
: Provides a wildcard search capability on the SQL text itself. It is useful for finding particular types of queries. For example, queries with a FETCH FIRST clause can be found by specifying 'fetch'. The search is case insensitive for ease of use. For example, the string 'FETCH' will find the same entries as the search string 'fetch'.

Multiple filter options can be specified. Note that in a multi-filter case, the candidate entries for each filter are computed independently and only those entries that are present in all the candidate lists are shown. So, for example, if you specified options **Minimum runtime for the longest execution** and **Queries run after this date and time**, you will be shown those entries with the minimum runtime that are run after the specified date and time.

Related reference

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

Importing a monitor

You can import monitor data that has been collected using Start Database Monitor (STRDBMON) command or some other interface by using iSeries Navigator.

To import monitor data, right-click **SQL Performance monitors** and select **Import**. Once you have imported a monitor, you can analyze the data.

Query optimizer index advisor

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

The 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 be the most selective secondary key column. The optimizer will use 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. Hence, secondary key columns that are fairly selective should be included.

It is up to the user to determine the true selectivity of any secondary key columns and to determine whether those key columns should be included when creating the index. When building the index the primary key columns should be the left-most key columns followed by any of the secondary key columns the user chooses and they should be prioritized by selectivity.

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. The CQE optimizer when suggesting indexes only considers the selection criteria and 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 include:

- The index advisor interface in iSeries 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 114
You can use Visual Explain to view many types of information.

"Database monitor view 3018 - STRDBMON/ENDDBMON" on page 236

"Viewing statements in a monitor" on page 110
You can view SQL statements that are included in a detailed monitor.

Display index advisor information

You can display index advisor information from the optimizer using iSeries Navigator.

iSeries navigator displays information found in the QSYS2/SYSIXADV system table.

To display index advisor information, follow these steps:

1. In the iSeries 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 can choose to create an index from the list, remove the index advised from the list, or clear the list entirely.

Database manager indexes advised system table:

This topic describes the indexes advised system table.

Table 30. 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	CHAR(10)	Schema containing the table

Table 30. SYSIXADV system table (continued)

Column name	System column name	Data type	Description
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)	Library name of the NLSS table

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 should significantly reduce the number of keys selected based on the corresponding query selection. Secondary key columns are columns that may or may not significantly reduce the number of keys selected.

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

View the implementation of your queries with Visual Explain

You can use the **Visual Explain** tool with iSeries Navigator to create a query graph that graphically displays the implementation of an SQL statement. You can use this tool to see information about both static and dynamic SQL statements. Visual Explain supports the following types of SQL statements: SELECT, INSERT, UPDATE, and DELETE.

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 iSeries Navigator. The second is through the Visual Explain (QQQVEXPL) API.

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

- Enter an SQL statement in the **Run SQL Scripts** window. Select the statement and choose **Explain** from the context menu, or select **Run and Explain** from the **Visual Explain** 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. Click **Run Visual Explain**. You can also start an SQL Performance Monitor from Run SQL Scripts. Select **Start SQL Performance monitor** from the **Monitor** menu.
- Start the Current SQL for a Job function by right-clicking **Databases** and select **Current SQL for a Job**. Select a job from the list and click **SQL Statement**. When the SQL is displayed in the lower pane, you can start Visual Explain by clicking **Run Visual Explain**.
- Right-click SQL Plan Cache and select **Show Statements**. Select filtering information and select the statement in the List of Statements window. Click **Run 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. Click **Run 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 query optimizer's estimates.

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

Run and Explain

If you select Run and Explain, the query is run by the system before the diagram is displayed. This option may take a significant amount of time, but the information displays 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 iSeries Navigator can be explained through iSeries Navigator. First you must import the database monitor table into iSeries Navigator. To do this, right-click the SQL Performance Monitors and choose the **Import** option. Specify a name for the performance monitor (name it will be known by within iSeries Navigator) and the qualified name of the database monitor table. Be sure to select Detailed as the type of monitor. Detailed represents the file-based (STRDBMON) monitor while Summary represents the memory-resident monitor (which is not supported by Visual Explain). Once the monitor has been imported, follow the steps to start Visual Explain from within iSeries Navigator.

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

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 implementation of a query, 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, thus excluding some preliminary or intermediate operations that are not essential for understanding the main flow of query implementation. The full view may show more icons that further depict the flow of the execution tree. You can change the graph detail by select **Graph Detail** from the **Options** menu and selecting either **Basic** or **Full**. The default view is **Basic**. Note that in order to see all of the detail for a **Full** view, you will need to change the Graph Detail to **Full**, close out Visual Explain, and run the query again. The setting for Graph Detail will persist.

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

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

"Query optimizer index advisor" on page 110
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.

Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command

You can modify different types of attributes of the queries that you will execute during a certain job with the Change Query Attributes (CHGQRYA) CL command, or by using the iSeries Navigator 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 42
The DB2 UDB Symmetric Multiprocessing feature provides the optimizer with additional methods for retrieving data that include parallel processing. Symmetrical multiprocessing (SMP) is a form of parallelism achieved on a single server where multiple (CPU and I/O) processors that share memory and disk resource work simultaneously toward achieving a single end result.

Related information

Change Query Attributes (CHGQRYA) command

Control queries dynamically with the query options file QAQQINI

The query options file QAQQINI support provides the ability to dynamically modify or override the environment in which queries are executed through the 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:

- APPLY_REMOTE
- ASYNC_JOB_USAGE
- COMMITMENT_CONTROL_LOCK_LIMIT
- FORCE_JOIN_ORDER
- IGNORE_DERIVED_INDEX
- IGNORE_LIKE_REDUNDANT_SHIFTS
- LOB_LOCATOR_THRESHOLD
- MATERIALIZED_QUERY_TABLE_REFRESH_AGE
- MATERIALIZED_QUERY_TABLE _USAGE
- MESSAGES_DEBUG
- NORMALIZE_DATA
- OPEN_CURSOR_CLOSE_COUNT
- OPEN_CURSOR_THRESHOLD
- OPTIMIZE_STATISTIC_LIMITATION
- OPTIMIZATION_GOAL
- PARALLEL_DEGREE
- PARAMETER_MARKER_CONVERSION
- QUERY_TIME_LIMIT
- REOPTIMIZE_ACCESS_PLAN
- SQLSTANDARDS_MIXED_CONSTANT
- SQL_FAST_DELETE_ROW_COUNT
- SQL_STMT_COMPRESS_MAX
- SQL_SUPPRESS_WARNINGS
- SQL_TRANSLATE_ASCII_TO_JOB
- STAR_JOIN
- STORAGE_LIMIT
- SYSTEM_SQL_STATEMENT_CACHE
- UDF_TIME_OUT
- VARIABLE_LENGTH_OPTIMIZATION

Related reference

"Look ahead predicate generation (LPG)" on page 52
A special type of transitive closure called look ahead predicate generation (LPG) may be costed for joins. In this case, the optimizer attempts to minimize the random I/O costs of a join by pre-applying the results of the query to a large fact table. LPG will typically be used with a class of queries referred to as star join queries, however it can possibly be used with any join query.

Specifying the QAQQINI file:

Use the Change Query Attributes (CHGQRYA) command with the QRYOPTLIB (query options library) parameter to specify which schema currently contains or will contain the query options file QAQQINI.

The query options file will be retrieved from the schema specified on the QRYOPTLIB parameter for each query and remains in effect for the duration of the job or user session, or until the QRYOPTLIB parameter is changed by the Change Query Attributes (CHGQRYA) command.

If the Change Query Attributes (CHGQRYA) command is not issued or is issued but the QRYOPTLIB parameter is not specified, the schema QUSRSYS is searched for the existence of the QAQQINI file. If a query options file is not found for a query, no attributes will be modified. Since the server is shipped with no INI file in QUSRSYS, you may receive a message indicating that there is no INI file. This message is not an error but an indication that a QAQQINI file that contains all default values is being used. The initial value of the QRYOPTLIB parameter for a job is QUSRSYS.

Related information

Change Query Attributes (CHGQRYA) command

Creating the QAQQINI query options file:

Each server is shipped with a QAQQINI template file in schema QSYS. The QAQQINI file in QSYS is to be used as a template when creating all user specified QAQQINI files.

To create your own QAQQINI file, use the Create Duplicate Object (CRTDUPOBJ) command to create a copy of the QAQQINI file in the schema that will be 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 may be corrupted and an error will be signaled that the options file cannot be retrieved or that the options file cannot be updated.

Because of the trigger programs attached to the QAQQINI file, the following CPI321A informational message will be displayed six times in the job log when the CRTDUPOBJ CL is used to create the file. This is not an error. It is only an informational message.

CPI321A Information Message: Trigger QSYS_TRIG_&1___QAQQINI___00000&N in library &1 was added to file QAQQINI in library &1. The ampersand variables (&1, &N) are replacement variables that contain either the library name or a numeric value.

Note: It is highly recommended that the file QAQQINI, in QSYS, not be modified. This is the original template that is to be duplicated into QUSRSYS or a user specified library for use.

Related information

Change Query Attributes (CHGQRYA) command

Create Duplicate Object (CRTDUPOBJ) command

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            QQPARM   256A        VARLEN(10) +
                                  TEXT('Query+
                                        option parameter') +
                                  COLHDG('Parameter')
A            QQVAL    256A        VARLEN(10) +
                                  TEXT('Query option +
                                        parameter value') +
                                  COLHDG('Parameter Value')
A            QQTEXT   1000G       VARLEN(100) +
                                  TEXT('Query +
                                        option text') +
                                  ALWNULL +
                                  COLHDG('Query Option' +
                                         'Text') +
                                  CCSID(13488) +
                                  DFT(*NULL)
A          K QQPARM
```

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 statment. This example sets MESSAGES_DEBUG = *YES so that the query optimizer will print out the optimizer debug messages:

```
UPDATE MyLib/QAQQINI SET QQVAL='*YES'
WHERE QQPARM='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 QQPARM='QUERY_TIME_LIMIT'
```

To insert a new row into MyLib/QAQQINI use the INSERT SQL statement. This example adds the QUERY_TIME_LIMIT row with a value of *NOMAX to the QAQQINI file:

```
 INSERT INTO MyLib/QAQQINI
VALUES('QUERY_TIME_LIMIT','*NOMAX','New time limit set by DBAdmin')
```

QAQQINI query options file authority requirements:

QAQQINI is shipped with a *PUBLIC *USE authority. This allows users to view the query options file, but not change it. Because changing the values of the QAQQINI file affect all queries that are run on the system, only the system or database administrator should 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. This is true even if the user has no authority to the query options library and file. This provides the system administrator with an additional security mechanism.

When the QAQQINI file resides in the library QUSRSYS the query options will effect all of the query users on the server. To prevent anyone from inserting, deleting, or updating the query options, the system administrator should remove update authority from *PUBLIC to the file. This will prevent users from changing the data in the file.

When the QAQQINI file resides in a user library and that library is specified on the QRYOPTLIB parameter of the Change Query Attributes (CHGQRYA) command, the query options will effect all of the queries run for that user's job. To prevent the query options from being retrieved from a particular library the system administrator can revoke authority to the 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 will be 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 31. Query Options Specified on QAQQINI Command

Parameter	Value	Description
ALLOW_TEMPORARY_INDEXES	*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, will a temporary index be allowed.
APPLY_REMOTE	*DEFAULT	The default value is set to *YES.
	*NO	The CHGQRYA attributes for the job are not applied to the remote jobs. The remote jobs will use the attributes associated to them on their servers.
	*YES	The query attributes for the job are applied to the remote jobs used in processing database queries involving distributed tables. For attributes where *SYSVAL is specified, the system value on the remote server is used for the remote job. This option requires that, if CHGQRYA was used for this job, the remote jobs must have authority to use the CHGQRYA command.

Table 31. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
ASYNC_JOB_USAGE	*DEFAULT	The default value is set to *LOCAL.
	*LOCAL	Asynchronous jobs may be used for database queries that involve only tables local to the server where the database queries are being run. In addition, for queries involving distributed tables, this option allows the communications required to be asynchronous. This allows each server involved in the query of the distributed tables to run its portion of the query at the same time (in parallel) as the other servers.
	*DIST	Asynchronous jobs may be used for database queries that involve distributed tables.
	*ANY	Asynchronous jobs may be used for any database query.
	*NONE	No asynchronous jobs are allowed to be used for database query processing. In addition, all processing for queries involving distributed tables occurs synchronously. Therefore, no inter-system parallel processing will occur.
CACHE_RESULTS	*DEFAULT	The default value is the same as *SYSTEM.
	*SYSTEM	The database manager may cache a query result set. A subsequent run of the query by that job or, if the ODP for the query has been deleted, by any job, will consider reusing the cached result set.
	*JOB	The database manager may cache a query result set from one run to the next for a job, 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.
COMMITMENT_CONTROL_ LOCK_LIMIT	*DEFAULT	*DEFAULT is equivalent to 500,000,000.
	Integer Value	The maximum number of records that can be locked to a commit transaction initiated after setting the new value. The valid integer value is 1–500,000,000.
FORCE_JOIN_ORDER	*DEFAULT	The default is set to *NO.
	*NO	Allow the optimizer to reorder join tables.
	*SQL	Only force the join order for those queries that use the SQL JOIN syntax. This mimics the behavior for the optimizer before V4R4M0.
	*PRIMARY nnn	Only force the join position for the file listed by the numeric value nnn (nnn is optional and will default to 1) into the primary position (or dial) for the join. The optimizer will then determine the join order for all of the remaining files based upon cost.
	*YES	Do not allow the query optimizer to reorder join tables as part of its optimization process. The join will occur in the order in which the tables were specified in the query.

Table 31. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
IGNORE_DERIVED_INDEX	*DEFAULT	The default value is the same as *NO.
	*YES	Allow the SQE optimizer to ignore the derived index and process the query. The resulting query plan will be created without any regard to the existence of the derived index(s). The index types that are ignored include: • Keyed logical files defined with select or omit criteria and with the DYNSLT keyword omitted • Keyed logical files built over multiple physical file members (V5R2 restriction, not a restriction for V5R3) • Keyed logical files where one or more keys reference an intermediate derivation in the DDS. Exceptions to this are: 1. when the intermediate definition is defining the field in the DDS so that shows up in the logical's format and 2. RENAME of a field (these two exceptions do not make the key derived) • Keyed logical files with K *NONE specified. • Keyed logical files with Alternate Collating Sequence (ACS) specified • SQL indexes created when the sort sequence active at the time of creation requires a weighting (translation) of the key to occur. This is true when any of several non-US language IDs are specified. It also occurs if language ID shared weight is specified, even for language US.
	*NO	Do not ignore the derived index. If a derived index exists, have CQE process the query.
IGNORE_LIKE_ REDUNDANT_SHIFTS	*DEFAULT	The default value is set to *OPTIMIZE.
	*ALWAYS	When processing the SQL LIKE predicate or OPNQRYF command %WLDCRD built-in function, redundant shift characters are ignored for DBCS-Open operands. Note that this option restricts the query optimizer from using an index to perform key row positioning for SQL LIKE or OPNQRYF %WLDCRD predicates involving DBCS-Open, DBCS-Either, or DBCS-Only operands.
	*OPTIMIZE	When processing the SQL LIKE predicate or the OPNQRYF command %WLDCRD built-in function, redundant shift characters may or may not be ignored for DBCS-Open operands depending on whether an index is used to perform key row positioning for these predicates. Note that this option will enable the query optimizer to consider key row positioning for SQL LIKE or OPNQRYF %WLDCRD predicates involving DBCS-Open, DBCS-Either, or DBCS-Only operands.
LIMIT_PREDICATE_ 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.

Table 31. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
LOB_LOCATOR_THRESHOLD	*DEFAULT	The default value is set to 0. This indicates that the database will take no action to free locators.
	Integer Value	If the value is 0, then the database will take no action to free locators. For values 1 through 250,000, on a FETCH request, the database will compare the active LOB locator count for the job against the threshold value. If the locator count is greater than or equal to the threshold, the database will free host server created locators that have been retrieved. This option applies to all host server jobs (QZDASOINIT) and has no impact to other jobs.
MATERIALIZED_QUERY_ TABLE_REFRESH_AGE	*DEFAULT	The default value is set to 0.
	0	No materialized query tables may be used.
	*ANY	Any tables indicated by the MATERIALIZED_ QUERY_TABLE_USAGE INI parameter may be used.
	timestamp_ duration	Only tables indicated by MATERIALIZED_ QUERY_TABLE_USAGE INI option which have a REFRESH TABLE performed within the specified timestamp duration may be used.
MATERIALIZED_QUERY_ TABLE_USAGE	*DEFAULT	The default value is set to *NONE.
	*NONE	Materialized query tables may not be used in query optimization and implementation.
	*ALL	User-maintained materialized query tables may be used.
	*USER	User-maintained materialized query tables may be used.
MESSAGES_DEBUG	*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	*DEFAULT	The default is set to *NO.
	*NO	Unicode constants, host variables, parameter markers, and expressions that combine strings will not be normalized.
	*YES	Unicode constants, host variables, parameter markers, and expressions that combine strings will be normalized
OPEN_CURSOR_CLOSE_ COUNT	*DEFAULT	*DEFAULT is equivalent to 0. See Integer Value for details.
	Integer Value	OPEN_CURSOR_CLOSE_COUNT is used in conjunction with OPEN_CURSOR_THRESHOLD to manage the number of open cursors within a job. If the number of open cursors, which includes open cursors and pseudo-closed cursors, reaches the value specified by the OPEN_CURSOR_THRESHOLD, pseudo-closed cursors are hard (fully) closed with the least recently used cursors being closed first. This value determines the number of cursors to be closed. The valid values for this parameter are 1 to 65536. The value for this parameter should be less than or equal to the number in the OPEN_CURSOR_THREHOLD parameter. This value is ignored if OPEN_CURSOR_THRESHOLD is *DEFAULT. If OPEN_CURSOR_THRESHOLD is specified and this value is *DEFAULT, the number of cursors closed is equal to OPEN_CURSOR_THRESHOLD multiplied by 10 percent and rounded up to the next integer value.

Table 31. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
OPEN_CURSOR_ THRESHOLD	*DEFAULT	*DEFAULT is equivalent to 0. See Integer Value for details.
	Integer Value	OPEN_CURSOR_THRESHOLD is used in conjunction with OPEN_CURSOR_CLOSE_COUNT to manage the number of open cursors within a job. If the number of open cursors, which includes open cursors and pseudo-closed cursors, reaches this threshold value, pseudo-closed cursors are hard (fully) closed with the least recently used cursors being closed first. The number of cursors to be closed is determined by OPEN_CURSOR_CLOSE_COUNT. The valid user-entered values for this parameter are 1 - 65536. Having a value of 0 (default value) indicates that there is no threshold and hard closes will not be forced on the basis of the number of open cursors within a job.
OPTIMIZATION_GOAL	*DEFAULT	Optimization goal is determined by the interface (ODBC, SQL precompiler options, OPTIMIZE FOR nnn ROWS clause).
	*FIRSTIO	All queries will be optimized with the goal of returning the first page of output as fast as possible. This goal works well when the control of the output is controlled by a user who is most likely to cancel the query after viewing the first page of output data. Queries coded with an OPTIMIZE FOR nnn ROWS clause will honor the goal specified by the clause.
	*ALLIO	All queries will be optimized with the goal of running the entire query to completion in the shortest amount of elapsed time. This is a good option for when the output of a query is being written to a file or report, or the interface is queuing the output data. Queries coded with an OPTIMIZE FOR nnn ROWS clause will honor the goal specified by the clause.
OPTIMIZE_STATISTIC_ LIMITATION	*DEFAULT	The amount of time spent in gathering index statistics is determined by the query optimizer.
	*NO	No index statistics will be gathered by the query optimizer. Default statistics will be used for optimization. (Use this option sparingly.)
	*PERCENTAGE integer value	Specifies the maximum percentage of the index that will be searched while gathering statistics. Valid values for are 1 to 99.
	*MAX_ NUMBER_ OF_RECORDS_ ALLOWED integer value	Specifies the largest table size, in number of rows, for which gathering statistics is allowed. For tables with more rows than the specified value, the optimizer will not gather statistics and will use default values.

Table 31. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
PARALLEL_DEGREE	*DEFAULT	The default value is set to *SYSVAL.
	*SYSVAL	The processing option used is set to the current value of the system value, QQRYDEGREE.
	*IO	Any number of tasks can be used when the database query optimizer chooses to use I/O parallel processing for queries. SMP parallel processing is not allowed.
	*OPTIMIZE	The query optimizer can choose to use any number of tasks for either I/O or SMP parallel processing to process the query or database file keyed access path build, rebuild, or maintenance. SMP parallel processing is used only if the system feature, DB2 Symmetric Multiprocessing for i5/OS, is installed. Use of parallel processing and the number of tasks used is determined with respect to the number of processors available in the server, this job has a share of the amount of active memory available in the pool in which the job is run, and whether the expected elapsed time for the query or database file keyed access path build or rebuild is limited by CPU processing or I/O resources. The query optimizer chooses an implementation that minimizes elapsed time based on the job has a share of the memory in the pool.
	*OPTIMIZE xxx	This option is very similar to *OPTIMIZE. The value xxx indicates the ability to specify an integer percentage value from 1-200. The query optimizer determines the parallel degree for the query using the same processing as is done for *OPTIMIZE, Once determined, the optimizer will adjust the actual parallel degree used for the query by the percentage given. This 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.
	*MAX	The query optimizer chooses to use either I/O or SMP parallel processing to process the query. SMP parallel processing will only be used if the system feature, DB2 Symmetric Multiprocessing for i5/OS, is installed. The choices made by the query optimizer are similar to those made for parameter value *OPTIMIZE except the optimizer assumes that all active memory in the pool can be used to process the query or database file keyed access path build, rebuild, or maintenance.
	*NONE	No parallel processing is allowed for database query processing or database table index build, rebuild, or maintenance.
	*NUMBER_OF _TASKS nn	Indicates the maximum number of tasks that can be used for a single query. The number of tasks will be capped off at either this value or the number of disk arms associated with the table.
PARAMETER_MARKER_ CONVERSION	*DEFAULT	The default value is set to *YES.
	*NO	Constants cannot be implemented as parameter markers.
	*YES	Constants can be implemented as parameter markers.

Table 31. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
QUERY_TIME_LIMIT	*DEFAULT	The default value is set to *SYSVAL.
	*SYSVAL	The query time limit for this job will be obtained from the system value, QQRYTIMLMT.
	*NOMAX	There is no maximum number of estimated elapsed seconds.
	integer value	Specifies the maximum value that is checked against the estimated number of elapsed seconds required to run a query. If the estimated elapsed seconds is greater than this value, the query is not started. Valid values range from 0 to 2,147,352,578.
REOPTIMIZE_ACCESS_PLAN	*DEFAULT	The default value is set to *NO.
	*NO	Do not force the existing query to be reoptimized. However, if the optimizer determines that optimization is necessary, the query will be reoptimized.
	*YES	Force the existing query to be reoptimized.
	*FORCE	Force the existing query to be reoptimized.
	*ONLY_ REQUIRED	Do not allow the plan to be reoptimized for any subjective reasons. For these cases, continue to use the existing plan since it is still a valid workable plan. This may mean that you may not get all of the performance benefits that a reoptimization plan may derive. Subjective reasons include, file size changes, new indexes, and so on. Non-subjective reasons include, deletion of an index used by existing access plan, query file being deleted and recreated, and so on.
SQLSTANDARDS_MIXED_ CONSTANT	*DEFAULT	The default value is set to *YES.
	*YES	SQL IGC constants will be treated as IGC-OPEN constants.
	*NO	If the data in the IGC constant only contains shift-out DBCS-data shift-in, then the constant will be treated as IGC-ONLY, otherwise it will be treated as IGC-OPEN.
SQL_FAST_DELETE_ROW_COUNT	*DEFAULT	The default value is set to 0. Having a value of 0 indicates that the database manager will choose how many rows to consider when determining whether fast delete should be used instead of a 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 result in no operational difference than using 0 for the option.
	*NONE	This value will force 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 will not write individual rows into a journal. The valid values range from 1 to 999,999,999,999,999.

Table 31. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
SQL_STMT_COMPRESS_MAX	*DEFAULT	The default value is set to 2, which indicates that the access plan associated with any statement will be removed after a statement has been compressed twice without being executed.
	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. Note that executing the SQL statement resets the count for that statement to 0. The valid Integer values are 1 to 255.
SQL_SUPPRESS_WARNINGS	*DEFAULT	The default value is set to *NO.
	*YES	Examine the SQLCODE in the SQLCA after execution of a statement. If the SQLCODE > 0, then alter the SQLCA so that no warning is returned to the caller. Set the SQLCODE to 0, the SQLSTATE to ’00000’ and SQLWARN to ’ ’.
	*NO	Specifies that SQL warnings will be returned to the caller.
SQL_TRANSLATE_ASCII_ TO_JOB	*DEFAULT	The default value is set to *NO.
	*YES	Translate ASCII SQL statement text to the CCSID of the iSeries job.
	*NO	Translate ASCII SQL statement text to the EBCIDIC CCSID associated with the ASCII CCSID.
STAR_JOIN (see note)	*DEFAULT	The default value is set to *NO
	*NO	The EVI Star Join optimization support is not enabled.
	*COST	Allow query optimization to consider (cost) the usage of EVI Star Join support. The determination of whether the Distinct List selection is used will be determined by the optimizer based on how much benefit can be derived from using that selection.
STORAGE_LIMIT	*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 may 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	*DEFAULT	The default value is set to *YES.
	*YES	Examine the SQL system-wide statement cache when an SQL prepare request is processed. If a matching statement already exists in the cache, use the results of that prepare. This allows the application to potentially have better performing prepares.
	*NO	Specifies that the SQL system-wide statement cache should not be examined when processing an SQL prepare request.

Table 31. Query Options Specified on QAQQINI Command (continued)

Parameter	Value	Description
UDF_TIME_OUT (see note)	*DEFAULT	The amount of time to wait is determined by the database. The default is 30 seconds.
	*MAX	The maximum amount of time that the database will wait for the UDF to finish.
	integer value	Specify the number of seconds that the database should wait for a UDF to finish. If the value given exceeds the database maximum wait time, the maximum wait time will be used by the database. Minimum value is 1 and maximum value is system defined.
VARIABLE_LENGTH_ OPTIMIZATION	*DEFAULT	The default value is set to *YES.
	*YES	Allow aggressive optimization of variable length columns. Allows index only access for the column(s). It also allows constant value substitution when an equal predicate is present against the column(s). As a consequence, the length of the data returned for the variable length column may not include any trailing blanks that existed in the original data.
	*NO	Do not allow aggressive optimization of variable length columns.

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

Set resource limits with the Predictive Query Governor

The DB2 Universal Database for iSeries 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 iSeries. It can be used with all DB2 Universal Database for iSeries query interfaces and is not limited to use with SQL queries.

The ability of the governor to predict and stop queries before they are started is important because:

- Operating a long-running query and abnormally ending the query before obtaining any results wastes server resources.
- Some 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 will take longer than the user wants to wait.

The governor in DB2 Universal Database for iSeries is based on two measurements:

- The estimated runtime for a query.
- The estimated temporary storage consumption for a query.

If the query's 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. This is the first place where the query optimizer attempts to find the time limit.
- Set the Query Time Limit option in the query options file. This is the second place where the query optimizer attempts to find the time limit.

- Set the QQRYTIMLMT system value. Allow each job to use the value *SYSVAL on the Change Query Attributes (CHGQRYA) CL command, and set the query options file to *DEFAULT. This is the third place where the query optimizer attempts to find the time limit.

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. This 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. This is the second place where the query optimizer attempts to find the time limit.

It is important to remember that 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 may vary considerably from the actual resource used. In those case, you may need to artificially adjust your limits to correspond to an inaccurate estimate.

When setting the time limit for the entire server, it is typically best to set the limit to the maximum allowable time that any query should be allowed to run. By setting the limit too low you will run the risk of preventing some queries from completing and thus preventing the application from successfully finishing. There are many functions that use the query component to internally perform query requests. These requests will also be compared to the user-defined time limit.

You can check the inquiry message CPA4259 for the predicted runtime and storage. If the query is canceled, debug messages will still be 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 in conjunction with the query optimizer.

When a user issues a request to the server to run a query, the following occurs:

1. The query access plan is created by the optimizer.

   As part of the evaluation, the optimizer predicts or estimates the runtime for the query. This helps determine the best way to access and retrieve the data for the query. 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 is 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 and 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 will see that only one limit was exceeded. Also, if no limit was explicitly specified by the user, a large integer value will be 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 to the message, and the query request is *always* ended.
5. If a default message reply is not used, the user chooses to do one of the following:

- End the query request before it is actually run.
- Continue and run the query even though the 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 do this by using the JOB parameter on the Change Query Attributes (CHGQRYA) command to specify either a query options file library to search (QRYOPTLIB) or a specific QRYTIMLMT, 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 is 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 being performed, the time of day, or the amount of system resources available. This provides a significant amount of flexibility when trying to balance system resources with temporary query requirements.

Canceling a query with the Query Governor:

When a query is expected to 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.

Controlling the default reply to the query governor inquiry message:

The system administrator can control whether the interactive user has the option of ignoring the database query inquiry message by using the 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 and 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 and must reply to the inquiry.
- If a value of *SYSRPYL 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 will allow the keyword CMPDTA to be used to select the system reply list entry that applies to the process or user profile. The user profile name is 10 characters long and starts at position 51. The process name is 10 character long and starts at position 27.
- The following example will add a reply list element that will cause the default reply of `C` to cancel any requests for jobs whose user profile is 'QPGMR'.

  ```
  ADDRPYLE  SEQNBR(56) MSGID(CPA4259) CMPDTA(QPGMR      51) RPY(C)
  ```

  The following example will add a reply list element that will cause the default reply of `C` to cancel any requests for jobs whose process name is 'QPADEV0011'.

```
ADDRPYLE  SEQNBR(57) MSGID(CPA4259) CMPDTA(QPADEV0011 27) RPY(C)
```

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 occurs even if the job is *not* in debug mode. You can then review the optimizer tuning messages in the job log to see if additional tuning is needed to obtain optimal query performance. This allows you to try several permutations of the query with different attributes, indexes, and syntax or both to determine what performs better through the optimizer without actually running the query to completion. This saves on system resources because the actual query of the data is never actually done. If the tables to be queried contain a large number of rows, this represents a significant savings in system resources.

Be careful when you use this technique for performance testing, because all query requests will be stopped before they are run. This is especially important for a 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 that relates to 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:

To set the query time limit for the current job or user session using query options file QAQQINI, specify QRYOPTLIB parameter on the Change Query Attributes (CHGQRYA) command to a user library where the QAQQINI file exists with the parameter set to QUERY_TIME_LIMIT, and the value set to a valid query time limit.

To set the query time limit for 45 seconds you can use the following Change Query Attributes (CHGQRYA) command:

```
CHGQRYA   JOB(*) QRYTIMLMT(45)
```

This sets the query time limit at 45 seconds. If the user runs a query with an estimated runtime equal to or less than 45 seconds, the query runs without interruption. The time limit remains in effect for the duration of the job or user session, or until the time limit is changed by the 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 sets the query time limit at 45 seconds for job 123456/USERNAME/JOBNAME. If job 123456/USERNAME/JOBNAME tries to run a query with an estimated runtime equal to or less than 45 seconds the query runs without interruption. If the estimated runtime for the query is greater than 45 seconds, for example 50 seconds, a message is sent to the user stating that the estimated runtime of 50 seconds exceeds the query time limit of 45 seconds. The time limit remains in effect for the duration of job 123456/USERNAME/JOBNAME, or until the time limit for job 123456/USERNAME/JOBNAME is changed by the 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 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

Testing 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 to run the query, such as a hash table, sort or temporary index.

To test for a temporary object’s usage, 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 will be used for the query, regardless of the temporary object’s estimated size.
- 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 allows you to see what temporary object(s) was 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.

To set the query temporary storage limit for a job using query options file QAQQINI, specify QRYOPTLIB parameter on the Change Query Attributes (CHGQRYA) command to 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.

In the case where 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's value is used.

To set the temporary storage limit for 100 megabytes 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 megabytes, the query runs without interruption. If the estimate is more than 100 megabytes, 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 megabytes 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

Control parallel processing for queries

There are two types of parallel processing available. The first is a parallel I/O that is available at no charge. The second is DB2 UDB Symmetric Multiprocessing, a feature that you can purchase. You can turn parallel processing on and off.

Even though parallelism has been enabled for a server or given job, the individual queries that run in a job might not actually use a parallel method. This might be because of functional restrictions, or the optimizer might choose a non-parallel method because it runs faster.

Because queries being processed with parallel access methods aggressively use main storage, CPU, and disk resources, the number of queries that use parallel processing should be limited and controlled.

Controlling system wide parallel processing for queries:

You can use the QQRYDEGREE system value to control parallel processing for a server.

The current value of the system value can be displayed or modified using the following CL commands:

- WRKSYSVAL - Work with System Value
- CHGSYSVAL - Change System Value
- DSPSYSVAL - Display System Value
- RTVSYSVAL - Retrieve System Value

The special values for QQRYDEGREE control whether parallel processing is allowed by default for all jobs on the server. The possible values are:

***NONE**
No parallel processing is allowed for database query processing.

***IO**
I/O parallel processing is allowed for queries.

***OPTIMIZE**
The query optimizer can choose to use any number of tasks for either I/O or SMP parallel processing to process the queries. SMP parallel processing is used only if the DB2 UDB Symmetric Multiprocessing feature is installed. The query optimizer chooses to use parallel processing to minimize elapsed time based on the job's share of the memory in the pool.

***MAX**
The query optimizer can choose to use either I/O or SMP parallel processing to process the query. SMP parallel processing can be used only if the DB2 UDB Symmetric Multiprocessing feature is installed. The choices made by the query optimizer are similar to those made for parameter value *OPTIMIZE, except the optimizer assumes that all active memory in the pool can be used to process the query.

The default value of the QQRYDEGREE system value is *NONE, so you must change the value if you want parallel query processing as the default for jobs run on the server.

Changing this system value affects all jobs that will be run or are currently running on the server whose DEGREE query attribute is *SYSVAL. However, queries that have already been started or queries using reusable ODPs are not affected.

Controlling job level parallel processing for queries:

You can also control query parallel processing at the job level using the DEGREE parameter of the Change Query Attributes (CHGQRYA) command or in the QAQQINI file, or using 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 UDB Symmetric Multiprocessing feature is installed. Use of parallel processing and the number of tasks used is

determined with respect to the number of processors available in the server, the job's share of the amount of active memory available in the pool in which the job is run, and whether the expected elapsed time for the query is limited by CPU processing or I/O resources. The query optimizer chooses an implementation that minimizes elapsed time based on the job's share of the memory in the pool.

***MAX**
: The query optimizer can choose to use either I/O or SMP parallel processing to process the query. SMP parallel processing can be used only if the DB2 UDB Symmetric Multiprocessing feature is installed. The choices made by the query optimizer are similar to those made for parameter value *OPTIMIZE except the optimizer assumes that all active memory in the pool can be used to process the query.

***NBRTASKS** *number-of-tasks*
: Specifies the number of tasks to be used when the query optimizer chooses to use SMP parallel processing to process a query. I/O parallelism is also allowed. SMP parallel processing can be used only if the DB2 UDB Symmetric Multiprocessing feature is installed.

  Using a number of tasks less than the number of processors available on the server restricts the number of processors used simultaneously for running a given query. A larger number of tasks ensures that the query is allowed to use all of the processors available on the server to run the query. Too many tasks can degrade performance because of the over commitment of active memory and the overhead cost of managing all of the tasks.

***SYSVAL**
: Specifies that the processing option used should be set to the current value of the QQRYDEGREE system value.

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

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 will be 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 based on the number of processors available in the system, this job's share of the amount of active memory available in the pool in which the job is run, and whether the expected elapsed time for the operation is limited by CPU processing or I/O resources. The database manager chooses an implementation that minimizes elapsed time based on the job's 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 similar to 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 is determined by the current degree in effect 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

Collecting statistics with the Statistics Manager

As stated earlier, the collection of statistics is handled by a separate component called the Statistics Manager. Statistical information can be used by the query optimizer to determine the best access plan for a query. Since the query optimizer bases its choice of access plan on the statistical information found in the table, it is important that this information be current.

On many platforms, statistics collection is a manual process that is the responsibility of the database administrator. With iSeries servers, the database statistics collection process is handled automatically, and only rarely is it necessary to update statistics manually.

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

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

Related concepts

"Statistics Manager" on page 5

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

Automatic statistics collection

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

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

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

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

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

Automatic statistics refresh

Column statistics are not maintained when the underlying table data changes. 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 completely new query was optimized or because an existing plan was re-optimized.

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

- Number of rows in the table has changed by more than 15% of the total table row count
- Number of rows changed in the table is more than 15% of the total table row count

If the statistics is determined to be stale, the Statistics Manager still uses the stale column statistics to answer the questions from the optimizer, but it also marks the column statistics as stale in the Plan Cache and generates a request to refresh the statistics.

Viewing statistics requests

You can view the current statistics requests by using iSeries Navigator or by using Statistics APIs.

To view requests in iSeries Navigator, right-click **Database** and select **Statistic Requests**. This window shows all user requested statistics collections that are pending or active, as well as all system requested statistics collections that are being considered (are candidates), 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 140
The following APIs are used to implement the statistics function of iSeries Navigator.

Indexes versus column statistics

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

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

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

When trying to determine the selectivity of predicates, the Statistics Manager considers column statistics and indexes as resources for its answers in the following order:
1. Try to use a multi-column keyed index when ANDed or ORed predicates reference multiple columns
2. If there is no perfect index that contains all of the columns in the predicates, it will try to find a combination of indexes that can be used.
3. For single column questions, it will use available column statistics
4. If the answer derived from the column statistics shows a selectivity of less than 2%, indexes are used to verify this answer

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

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

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

Monitoring background statistics collection

The system value QDBFSTCCOL controls who is allowed to create statistics in the background.

The following list provides the possible values:

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

***NONE**
Restricts everyone from creating statistics in the background. This does not prevent immediate user-requested statistics from being collected, however.

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

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

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

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

There are four types of request queues based on who submitted the request and how long the collection is estimated to take. The default priority assigned to each thread can determine to which queue the thread belongs:
- Priority 90 — short user requests
- Priority 93 — long user requests
- Priority 96 — short system requests
- Priority 99 — long system requests

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

Related information

Allow background database statistics collection (QDBFSTCCOL) system value

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 iSeries Navigator or the statistics APIs. If statistics are not needed immediately, then the creation of column statistics may be performed automatically by the system after the first touch of a column by a query.

Statistics are copied when using Create Duplicate Object (CRTDUPOBJ) command with DATA(*YES). You can use this 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 iSeries Navigator. Right-click a table or alias and select **Statistic Data**. Another way is to create a user-defined table function and call that function from an SQL statement or stored procedure.

Manually collecting and refreshing statistics

You can manually collect and refresh statistics through iSeries Navigator or by using Statistics APIs.

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

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

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

High Availability (HA) solutions

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

ISV (Independent Solution Provider) preparation

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

Business Intelligence environments

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

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

You should be aware that this procedure works only if you do not delete (drop) the tables and 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 that significantly change the cardinality, add new ranges of values, or change the distribution of data values can affect the performance for queries when they are first run against the new data. This may happen because, on the first run of such a query, the optimizer uses stale column statistics to make decisions on the access plan. At that point, it starts a request to refresh the column statistics.

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

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

Backup and recovery

When thinking about backup and recovery strategies, keep in mind that creation of column statistics is not journaled. Column statistics that exist at the time a save operation occurs are saved as part of the table and restored with the table. Any column statistics created after the save took place are lost and cannot be recreated by using techniques such as applying journal entries.

If you have a rather long interval between save operations and rely heavily on journaling for restoring your environment to a current state, consider keeping track of column statistics that are generated after the latest save operation.

Related information

Allow background database statistics collection (QDBFSTCCOL) system value

Statistics Manager APIs

The following APIs are used to implement the statistics function of iSeries Navigator.

- Cancel Requested Statistics Collections(QDBSTCRS, QdbstCancelRequestedStatistics) immediately cancels statistics collections that have been requested, but are not yet completed or not successfully completed.
- Delete Statistics Collections (QDBSTDS, QdbstDeleteStatistics) immediately deletes existing completed statistics collections.
- List Requested Statistics Collections (QDBSTLRS, QdbstListRequestedStatistics) lists all of the columns and combination of columns and file members that have background statistic collections requested, but not yet completed.
- List Statistics Collection Details(QDBSTLDS, QdbstListDetailStatistics) lists additional statistics data for a single statistics collection.
- List Statistics Collections(QDBSTLS, QdbstListStatistics) lists all of the columns and combination of columns for a given file member that have statistics available.
- Request Statistics Collections(QDBSTRS, QdbstRequestStatistics) allows you to request one or more statistics collections for a given set of columns of a specific file member.
- Update Statistics Collection(QDBSTUS, QdbstUpdateStatistics) allows you to update the attributes and to refresh the data of an existing single statistics collection

Related reference

“Viewing statistics requests” on page 136
You can view the current statistics requests by using iSeries Navigator or by using Statistics APIs.

Display information with Database Health Center

Use the Database Health Center to capture information about your database. You can view the total number of objects, the size limits of selected objects in your database, and the design limits of selected objects.

To start the Health Center, follow these steps:

1. In the iSeries 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 iSeries 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. 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 will create the table for you.

Query optimization tools: Comparison table

Use this table to learn what information each tool can yield about your query, when in the process a specific tool can analyze your query, and the tasks that each tool can perform to improve your query.

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 iSeries Navigator Database and API interface
Messages are printed to spool file	Messages is displayed in job log	Performance rows are written to database table	Performance information is collected in memory and then written to database table	Information is displayed visually through iSeries Navigator
Easier to tie messages to query with subqueries or unions	Difficult to tie messages to query with subqueries or unions	Uniquely identifies every query, subquery and materialized view	Repeated query requests are summarized	Easy to view implementation of the query and associated information

Creating an index strategy

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

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

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

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

Related information

SQL Create Index statement

Specifying PAGESIZE on CRTPF or CRTLF commands

When creating keyed files or indexes using the Create Physical File (CRTPF) or Create Logical File (CRTLF) commands, or the SQL CREATE INDEX statement, you can use the PAGESIZE parameter to specify the access path logical page size that is used by the system when the access path is created.

This logical page size is the amount of bytes of the access path that can be moved into the job's storage pool from the auxiliary storage for a page fault.

You should consider using the default of *KEYLEN for this parameter except in rare circumstances so that the page size can be determined by the system based on the total length of the key, or keys. When the access path is used by very selective queries (for example, individual key look up), a smaller page size is typically more efficient. Also, when the keys being selected by queries are grouped together in the access path and many records are being selected, or the access path is being scanned, a larger page size is typically 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, you should consider the maintenance cost of creating and using additional indexes. For radix indexes with MAINT(*IMMED) maintenance occurs when inserting, updating or deleting rows.

To reduce the maintenance of your indexes consider:

- Minimizing the number of indexes over a given table by creating composite (multiple column) key indexes such that an index 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, while maintaining a reasonable balance on the number of indexes so as to limit maintenance overhead

Encoded vector indexes

An encoded vector index (EVI) is an index object that is used by the query optimizer and database engine to provide fast data access in decision support and query reporting environments.

EVIs are a complementary alternative to existing index objects (binary radix tree structure - logical file or SQL index) and are a variation on bitmap indexing. Because of their compact size and relative simplicity, EVIs provide for faster scans of a table that can also be processed in parallel.

An EVI is a data structure that is stored as two components:
- The symbol table contains statistical and descriptive information about each distinct key value represented in the table. Each distinct key is assigned a unique code, either 1, 2 or 4 bytes in size.
- The vector is an array of codes listed in the same ordinal position as the rows in the table. The vector does not contain any pointers to the actual rows in the table.

Advantages of EVIs:
- Require less storage
- May have better build times than radix, especially if the number of unique values in the column(s) 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.

Disadvantages of EVIs:
- Cannot be used in ordering
- Use for grouping is specialized
- Use with joins always done in cooperation with hash table processing
- Some additional maintenance idiosyncrasies

Related information

SQL Create Index statement

How the EVI works

EVIs 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 may use the EVI in one of the following ways:
- **Selection (WHERE clause)**

  If the optimizer decides to use an EVI to process the query, the database engine uses the vector to build the dynamic bitmap (or a list of selected row ids) that contains one bit for each row in the table, the bit being turned on for each selected row. Like a bitmap index, these intermediate dynamic bitmaps (or lists) can be AND'ed and OR'ed together to satisfy an ad hoc query.

  For example, if a user wants to see sales data for a certain region during a certain time period, you can define an EVI over the region column and the Quarter column of the database. When the query runs, the database engine builds dynamic bitmaps using the two EVIs and then ANDs the bitmaps together to produce a bitmap that contains only the relevant rows for both selection criteria. This AND'ing capability drastically reduces the number of rows that the server must read and test. The dynamic bitmap(s) exists only as long as the query is executing. Once the query is completed, the dynamic bitmap(s) are eliminated.
- **Grouping or Distinct**

  The symbol table within the EVI contains the distinct values for the specified columns in the key definition, along with a count of the number of records in the base table that have each distinct value. The symbol table in effect contains the grouping results of the columns in that key. Therefore, queries involving grouping or distinct on the columns in that key are potential candidates for a technique that uses the symbol table directly to determine the query result. Note that the symbol table contains only the key values and their associated counts. Therefore, queries involving column function COUNT are eligible for this technique, but queries involving column functions MIN or MAX on other columns are not eligible (since the min and max values are not stored in the symbol table).

When to create EVIs

There are several instances when you should consider creating EVIs.

Encoded vector indexes should be considered 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 will 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 has a small number of distinct values and (if a column function is specified at all) only the COUNT column function is used.

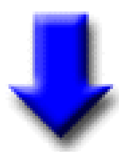
Encoded vector indexes should be created with:

- Single key columns with a low number of distinct values expected
- Keys columns with a low volatililty (they don't change often)
- Maximum number of distinct values expected using the WITH n DISTINCT VALUES clause
- Single key over foreign key columns for a star schema model

EVI maintenance

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

Table 32. EVI Maintenance Considerations

	Condition	Characteristics
Most Effective	When inserting an existing distinct key value	• Minimum overhead • Symbol table key value looked up and statistics updated • Vector element added for new row, with existing byte code
	When inserting a *new* distinct key value - in order, within byte code range	• Minimum overhead • Symbol table key value added, byte code assigned, statistics assigned • Vector element added for new row, with new byte code
	When inserting a new distinct key value - out of order, within byte code range	• Minimum overhead if contained within overflow area threshold • Symbol table key value added to overflow area, byte code assigned, statistics assigned • Vector element added for new row, with new byte code • Considerable overhead if overflow area threshold reached • Access path validated - not available • EVI refreshed, overflow area keys incorporated, new byte codes assigned (symbol table and vector elements updated)
Least Effective	When inserting a new distinct key value - out of byte code range	• Considerable overhead • Access plan invalidated - not available • EVI refreshed, next byte code size used, new byte codes assigned (symbol table and vector elements updated

Recommendations for EVI use

Encoded vector indexes are a powerful tool for providing fast data access in decision support and query reporting environments, but to ensure the effective use of EVIs, you should implement EVIs with the following guidelines:

Create EVIs on

- Read-only tables or tables with a minimum of INSERT, UPDATE, DELETE activity.
- Key columns that are used in the WHERE clause - local selection predicates of SQL requests.
- Single key columns that have a relatively small set of distinct values.
- Multiple key columns that result in a relatively small set of distinct values.
- Key columns that have a static or relatively static set of distinct values.

- Non-unique key columns, with many duplicates.

Create EVIs with the maximum byte code size expected

- Use the ″WITH n DISTINCT VALUES″ clause on the CREATE ENCODED VECTOR INDEX statement.
- If unsure, use a number greater than 65,535 to create a 4 byte code, thus avoiding the EVI maintenance overhead of switching byte code sizes.

When loading data

- Drop EVIs, load data, create EVIs.
- EVI byte code size will be assigned automatically based on the number of actual distinct key values found in the table.
- Symbol table will contain all key values, in order, no keys in overflow area.

Consider SMP and parallel index creation and maintenance

Symmetrical Multiprocessing (SMP) is a valuable tool for building and maintaining indexes in parallel. The results of using the optional SMP feature of i5/OS are faster index build times, and faster I/O velocities while maintaining indexes in parallel. Using an SMP degree value of either *OPTIMIZE or *MAX, additional multiple tasks and additional server resources are used to build or maintain the indexes. With a degree value of *MAX, expect linear scalability on index creation. For example, creating indexes on a 4 processor server can be 4 times as fast as a 1 processor server.

Checking values in the overflow area

You can also use the Display File Description (DSPFD) command (or iSeries Navigator - Database) to check how many values are in the overflow area. Once the DSPFD command is issued, check the overflow area parameter for details on the initial and actual number of distinct key values in the overflow area.

Using CHGLF to rebuild an index’s access path

Use the Change Logical File (CHGLF) command with the attribute Force Rebuild Access Path set to *YES* (FRCRBDAP(*YES)). This command accomplishes the same thing as dropping and recreating the index, but it does not require that you know about how the index was built. This command is especially effective for applications where the original index definitions are not available, or for refreshing the access path.

Related information

SQL Create Index statement

Change Logical File (CHGLF) command

Display File Description (DSPFD) command

Comparing Binary radix indexes and Encoded vector indexes

DB2 UDB for iSeries makes indexes a powerful tool.

The following table summarizes some of the differences between binary radix indexes and encoded vector indexes:

Table 33. Comparison of radix and evi indexes

	Binary Radix Indexes	Encoded Vector Indexes
Basic data structure	A wide, flat tree	A Symbol Table and a vector
Interface for creating	Command, SQL, iSeries Navigator	SQL, iSeries Navigator
Can be created in parallel	Yes	Yes

Table 33. Comparison of radix and evi indexes (continued)

	Binary Radix Indexes	Encoded Vector Indexes
Can be maintained in parallel	Yes	Yes
Used for statistics	Yes	Yes
Used for selection	Yes	Yes, via dynamic bitmaps or RRN list
Used for joining	Yes	Yes (in conjunction with a hash table)
Used for grouping	Yes	Yes
Used for ordering	Yes	No
Used to enforce unique Referential Integrity constraints	Yes	No

Indexes and the optimizer

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

The CQE optimizer attempts to examine most, if not all, indexes built over 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 quickly and efficiently eliminates the rows that are not interesting or required to satisfy the request. Normally, query optimization is thought of as trying to find the rows of interest. A proper indexing strategy will assist the optimizer and database engine with this task.

Instances where an index is not used

DB2 Universal Database for iSeries does not use indexes in the following instances:

- For a column that is expected to be updated; for example, when using SQL, your program might include the following:

  ```
  EXEC SQL
   DECLARE DEPTEMP CURSOR FOR
      SELECT EMPNO, LASTNAME, WORKDEPT
        FROM CORPDATA.EMPLOYEE
        WHERE (WORKDEPT = 'D11' OR
                 WORKDEPT = 'D21') AND
                EMPNO = '000190'
        FOR UPDATE OF EMPNO, WORKDEPT
  END-EXEC.
  ```

  When using the OPNQRYF command, for example:

  ```
  OPNQRYF FILE((CORPDATA/EMPLOYEE)) OPTION(*ALL)
     QRYSLT('(WORKDEPT *EQ ''D11'' *OR WORKDEPT *EQ ''D21'')
     *AND EMPNO *EQ ''000190''')
  ```

  Even if you do not intend to update the employee's department, the system cannot use an index with a key of WORKDEPT.

  The system can use an index if all of the updateable columns used within the index are also used within the query as an isolatable selection predicate with an equal operator. In the previous example, the system uses an index with a key of EMPNO.

  The system can operate more efficiently if the FOR UPDATE OF column list only names the column you intend to update: *WORKDEPT*. Therefore, do not specify a column in the FOR UPDATE OF column list unless you intend to update the column.

If you have an updateable cursor because of dynamic SQL or the FOR UPDATE clause was not specified and the program contains an UPDATE statement then all columns can be updated.

- For a column being compared with another column from the same row. For example, when using SQL, your program might include the following:

```
EXEC SQL
 DECLARE DEPTDATA CURSOR FOR
    SELECT WORKDEPT, DEPTNAME
      FROM CORPDATA.EMPLOYEE
      WHERE WORKDEPT = ADMRDEPT
END-EXEC.
```

When using the OPNQRYF command, for example:

```
OPNQRYF FILE (EMPLOYEE) FORMAT(FORMAT1)
   QRYSLT('WORKDEPT *EQ ADMRDEPT')
```

Even though there is an index for *WORKDEPT* and another index for *ADMRDEPT*, DB2 Universal Database for iSeries will not use either index. The index has no added benefit because every row of the table needs to be looked at.

Determining unnecessary indexes

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, in an RPG application). Furthermore, it was difficult to find all the indexes over a physical file. Indexes are created as part of a keyed physical file, a keyed logical file, a join logical file, an SQL index, a primary key or unique constraint, or a referential constraint. However, you can now easily find all indexes and retrieve statistics on index usage as a result of new V5R3 iSeries Navigator and i5/OS 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 this through the iSeries Navigator, navigate to: **Database** → **Schemas** → **Tables**. Right-click your table and select **Show Indexes**

Note: You can also view the statistics through the Retrieve Member Description (QUSRMBRD) API.

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

Last Query Use
States the timestamp when the index was last used to 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 will be 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 server, the statistics are reset.

Related reference

"Starting a summary monitor" on page 93
You can start a summary monitor from the iSeries Navigator interface.

Related information

Retrieve Member Description (QUSRMBRD) API

Change Object Description (CHGOBJD) command

Manage index rebuilds

You can manage the rebuild of your indexes using iSeries 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 iSeries 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 **Manage index rebuilds**.

The Access paths to rebuild dialog includes the following columns:

Table 34. Columns for access path rebuild dialog

Column	Description
SQL Name	Long name of access path being rebuilt
Schema	Schema name where the object is located
Type	Owning type of the object
Status	Status of the rebuild for this access path
Rebuild Priority	Priority in which the rebuild for this access path will be run. Also referred to as sequence number.
Rebuild Reason	Reason why this access path needs to be rebuilt.
Rebuild Reason Type	Reason type why this access path needs to be rebuilt.
Invalidate Reason	Reason why this access path was invalidated.
Invalidation Reason Type	Reason type why this access path was invalidated
Estimated Rebuild Time	Amount of time that it is estimated that the rebuild of the access path will take.
Rebuild Start Time	Time when the rebuild was started.
Elapsed Rebuild Time	Amount of time that has elapsed since the start of the rebuild of the access path.
Unique	Whether the rows in the access path are unique
Last Query Use	Time when the access path was last used.
Last Statistics Use	Time when the access path was last used for statistics
Query Use Count	Number of times the access path has been used for a query.
Statistics Use Count	Number of times the access path has been used for statistics.
Partition	Partition name of the object
Short Name	Short (system) name of access path being rebuilt
Text	Description of this access path if any.

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

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

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

Related information

Indexing and statistics strategies for DB2 UDB for iSeries

Reactive approach to tuning

To perform reactive tuning, build a prototype of the proposed application without any indexes and start running some queries or build an initial set of indexes and start running the application to see what gets used and what does not. Even with a smaller database, the slow running queries will become obvious very quickly.

The reactive tuning method is also used when trying to understand and tune an existing application that is not performing up to expectations. Using the appropriate debugging and monitoring tools, which are described in the next section, the database feedback messages that will tell basically three things can be viewed:

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

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

Proactive approach to tuning

Typically you will create an index for the most selective columns and create statistics for the least selective columns in a query. By creating an index, the optimizer knows that the column is selective and it also gives the optimizer the ability to 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 that will help you to design code which allows DB2 Universal Database for iSeries 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 Universal Database for iSeries does not use an index for the named column if the host variable or constant value has a greater precision than the precision of the column. If the two items being compared have different data types, DB2 Universal Database for iSeries will need to convert one or the other of the values, which can result in inaccuracies (because of limited machine precision).

To avoid problems for columns and constants being compared, use the following:

- same data type
- same scale, if applicable
- same precision, if applicable

For example, EDUCLVL is a halfword integer value (SMALLINT). When using SQL, specify:

```
... WHERE EDUCLVL < 11 AND
               EDUCLVL >= 2
```

instead of

```
... WHERE EDUCLVL < 1.1E1 AND
               EDUCLVL > 1.3
```

When using the OPNQRYF command, specify:

```
... QRYSLT('EDUCLVL *LT 11 *AND ENUCLVL *GE 2')
```

instead of

```
... QRYSLT('EDUCLVL *LT 1.1E1 *AND EDUCLVL *GT 1.3')
```

If an index was created over the EDUCLVL column, then the optimizer does not use the index in the second example because the precision of the constant is greater than the precision of the column. In the first example, the optimizer considers using the index, because the precisions are equal.

Avoid arithmetic expressions

Do not use an arithmetic expression as an operand to be compared to a column in a row selection predicate. The optimizer does not use an index on a column that is being compared to an arithmetic expression. While this may not cause an index over the column to become unusable, it will prevent any estimates and possibly the use of index scan-key positioning on the index. The primary thing that is lost is the ability to use and extract any statistics that might be useful in the optimization of the query.

For example, when using SQL, specify the following:

```
... WHERE SALARY > 16500
```

instead of

```
... WHERE SALARY > 15000*1.1
```

Avoid character string padding

Try to use the same data length when comparing a fixed-length character string column value to a host variable or constant value. DB2 Universal Database for iSeries does not use an index if the constant value or host variable is longer than the column length.

For example, EMPNO is CHAR(6) and DEPTNO is CHAR(3). For example, when using SQL, specify the following:

```
... WHERE EMPNO > '000300' AND
               DEPTNO < 'E20'
```

instead of

```
... WHERE EMPNO > '000300 ' AND
               DEPTNO < 'E20 '
```

When using the OPNQRYF command, specify:

```
... QRYSLT('EMPNO *GT "000300" *AND DEPTNO *LT "E20"')
```

instead of

```
... QRYSLT('EMPNO *GT "000300" *AND DEPTNO *LT "E20"')
```

Avoid the use of like patterns beginning with % or _

The percent sign (%), and the underline (_), when used in the pattern of a LIKE (OPNQRYF %WLDCRD) predicate, specify a character string that is similar to the column value of rows you want to select. They can take advantage of indexes when used to denote characters in the middle or at the end of a character string.

For example, when using SQL, specify the following:

```
... WHERE LASTNAME LIKE 'J%SON%'
```

When using the OPNQRYF command, specify the following:

```
... QRYSLT('LASTNAME *EQ %WLDCRD(''J*SON*'')')
```

However, when used at the beginning of a character string, they can prevent DB2 Universal Database for iSeries from using any indexes that might be defined on the LASTNAME column to limit the number of rows scanned using index scan-key positioning. Index scan-key selection, however, is allowed. For example, in the following queries index scan-key selection can be used, but index scan-key positioning cannot.

In SQL:

```
... WHERE LASTNAME LIKE '%SON'
```

In OPNQRYF:

```
... QRYSLT('LASTNAME *EQ %WLDCRD(''*SON'')')
```

Ideally, you should avoid patterns with a % so that you can get the best performance when you perform key processing on the predicate. If possible, you should try to get a partial string to search so that index scan-key positioning can be used.

For example, if you were looking for the name ″Smithers″, but you only type ″S%,″ this query returns all names starting with ″S.″ You should adjust the query to return all names with ″Smi%″. By forcing the use of partial strings, you may get better performance in the long term.

Using indexes with sort sequence

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

Using indexes and sort sequence with selection, joins, or grouping

Before using an existing index, DB2 Universal Database for iSeries ensures the attributes of the columns (selection, join, or grouping columns) match the attributes of the key columns in the existing index. The sort sequence table is an additional attribute that must be compared.

The sort sequence table associated with the query (specified by the SRTSEQ and LANGID parameters) must match the sort sequence table with which the existing index was built. DB2 Universal Database for iSeries compares the sort sequence tables. If they do not match, the existing index cannot be used.

There is an exception to this, however. If the sort sequence table associated with the query is a unique-weight sequence table (including *HEX), DB2 Universal Database for iSeries acts as though no sort sequence table is specified for selection, join, or grouping columns that use the following operators and predicates:

- equal (=) operator
- not equal (^= or <>) operator
- LIKE predicate (OPNQRYF %WLDCRD and *CT)
- IN predicate (OPNQRYF %VALUES)

When these conditions are true, DB2 Universal Database for iSeries is free to use any existing index where the key columns match the columns and either:

- The index does not contain a sort sequence table or
- The index contains a unique-weight sort sequence table

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 between them that is also referenced in the query selection, then those indexes must either use the same sort sequence table or use no sort sequence table.

Using indexes and sort sequence with ordering

Unless the optimizer chooses to do a sort to satisfy the ordering request, the sort sequence table associated with the index must match the sort sequence table associated with the query.

When a sort is used, the translation is done during the sort. Since the sort is handling the sort sequence requirement, this allows DB2 Universal Database for iSeries to use any existing index that meets the selection criteria.

Examples of indexes

The following index examples are provided to help you create effective indexes.

For the purposes of the examples, assume that three indexes are created.

Assume that an index HEXIX was created with *HEX as the sort sequence.

```
CREATE INDEX HEXIX ON STAFF (JOB)
```

Assume that an index UNQIX was created with a unique-weight sort sequence.

```
CREATE INDEX UNQIX ON STAFF (JOB)
```

Assume that an index SHRIX was created with a shared-weight sort sequence.

```
CREATE INDEX SHRIX ON STAFF (JOB)
```

Index example: Equals selection with no sort sequence table

Equals selection with no sort sequence table (SRTSEQ(*HEX)).

```
SELECT * FROM STAFF
  WHERE JOB = 'MGR'
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
   QRYSLT('JOB *EQ ''MGR''')
   SRTSEQ(*HEX)
```

The system can use either index HEXIX or index UNQIX.

Index example: Equals selection with a unique-weight sort sequence table

Equals selection with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF
 WHERE JOB = 'MGR'
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
   QRYSLT('JOB *EQ ''MGR''')
   SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

The system can use either index HEXIX or index UNQIX.

Index example: Equal selection with a shared-weight sort sequence table

Equal selection with a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT * FROM STAFF
 WHERE JOB = 'MGR'
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
   QRYSLT('JOB *EQ ''MGR''')
   SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

The system can only use index SHRIX.

Index example: Greater than selection with a unique-weight sort sequence table

Greater than selection with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF
  WHERE JOB > 'MGR'
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
   QRYSLT('JOB *GT ''MGR''')
   SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

The system can only use index UNQIX.

Index example: Join selection with a unique-weight sort sequence table

Join selection with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF S1, STAFF S2
  WHERE S1.JOB = S2.JOB
```

or the same query using the JOIN syntax.

```
SELECT *
  FROM STAFF S1 INNER JOIN STAFF S2
        ON S1.JOB = S2.JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE(STAFF STAFF)
   FORMAT(FORMAT1)
   JFLD((1/JOB 2/JOB *EQ))
   SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

The system can use either index HEXIX or index UNQIX for either query.

Index example: Join selection with a shared-weight sort sequence table

Join selection with a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT * FROM STAFF S1, STAFF S2
  WHERE S1.JOB = S2.JOB
```

or the same query using the JOIN syntax.

```
SELECT *
  FROM STAFF S1 INNER JOIN STAFF S2
        ON S1.JOB = S2.JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE(STAFF STAFF) FORMAT(FORMAT1)
   JFLD((1/JOB 2/JOB *EQ))
   SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

The system can only use index SHRIX for either query.

Index example: Ordering with no sort sequence table

Ordering with no sort sequence table (SRTSEQ(*HEX)).

```
SELECT * FROM STAFF
  WHERE JOB = 'MGR'
  ORDER BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
   QRYSLT('JOB *EQ ''MGR''')
   KEYFLD(JOB)
   SRTSEQ(*HEX)
```

The system can only use index HEXIX.

Index example: Ordering with a unique-weight sort sequence table

Ordering with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF
 WHERE JOB = 'MGR'
 ORDER BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
   QRYSLT('JOB *EQ ''MGR''')
   KEYFLD(JOB) SRTSEQ(*LANGIDUNQ) LANGID(ENU)
```

The system can only use index UNQIX.

Index example: Ordering with a shared-weight sort sequence table

Ordering with a shared-weight sort sequence table (SRTSEQ(*LANGIDSHR) LANGID(ENU)).

```
SELECT * FROM STAFF
 WHERE JOB = 'MGR'
 ORDER BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
   QRYSLT('JOB *EQ ''MGR''')
   KEYFLD(JOB) SRTSEQ(*LANGIDSHR) LANGID(ENU)
```

The system can only use index SHRIX.

Index example: Ordering with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table

Ordering with ALWCPYDTA(*OPTIMIZE) and a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT * FROM STAFF
 WHERE JOB = 'MGR'
 ORDER BY JOB
```

When using the OPNQRYF command, specify:

```
OPNQRYF FILE((STAFF))
   QRYSLT('JOB *EQ ''MGR''')
   KEYFLD(JOB)
   SRTSEQ(*LANGIDUNQ) LANGID(ENU)
   ALWCPYDTA(*OPTIMIZE)
```

The system can use either index HEXIX or index UNQIX for selection. Ordering 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 3 more indexes are created over columns JOB and SALARY. The CREATE INDEX statements precede the examples.

Assume an index HEXIX2 was created with *HEX as the sort sequence.

```
CREATE INDEX HEXIX2 ON STAFF (JOB, SALARY)
```

Assume that an index UNQIX2 was created and the sort sequence is a unique-weight sort sequence.

```
CREATE INDEX UNQIX2 ON STAFF (JOB, SALARY)
```

Assume an index SHRIX2 was created with a shared-weight sort sequence.

```
CREATE INDEX SHRIX2 ON STAFF (JOB, SALARY)
```

Index example: Ordering and grouping on the same columns with a unique-weight sort sequence table

Ordering and grouping on the same columns with a unique-weight sort sequence table (SRTSEQ(*LANGIDUNQ) LANGID(ENU)).

```
SELECT JOB, SALARY FROM STAFF
  GROUP BY JOB, SALARY
  ORDER BY JOB, SALARY
```

When using the OPNQRYF command, specify:

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

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

Application design tips for database performance

There are some design tips that you can apply when designing SQL applications to maximize your database performance.

Use live data

The term *live data* refers to the type of access that the database manager uses when it retrieves data without making a copy of the data. Using this type of access, the data, which is returned to the program, always reflects the current values of the data in the database. The programmer can control whether the database manager uses a copy of the data or retrieves the data directly. This is done by specifying the allow copy data (ALWCPYDTA) parameter on the precompiler commands or on the Start SQL (STRSQL) command.

Specifying ALWCPYDTA(*NO) instructs the database manager to always use live data. In most cases, forcing live data access is a detriment to performance as it severely limits the possible plan choices that the optimizer may use to implement the query. Consequently, in most cases it should be avoided. However, in specialized cases involving a simple query, live data access can be used as a performance advantage because the cursor does not need be closed and opened again to refresh the data being retrieved. An example application demonstrating this advantage is one that produces a list on a display. If the display screen can only show 20 elements of the list at a time, then, after the initial 20 elements are displayed, the application programmer can request that the next 20 rows be displayed. A typical SQL application designed for an operating system other than the i5/OS operating system, might be structured as follows:

```
EXEC SQL
    DECLARE C1 CURSOR FOR
    SELECT EMPNO, LASTNAME, WORKDEPT
      FROM CORPDATA.EMPLOYEE
     ORDER BY EMPNO
END-EXEC.

EXEC SQL
    OPEN C1
END-EXEC.

*    PERFORM FETCH-C1-PARA  20 TIMES.

     MOVE EMPNO to LAST-EMPNO.

EXEC SQL
    CLOSE C1
END-EXEC.

*    Show the display and wait for the user to indicate that
*    the next 20 rows should be displayed.

EXEC SQL
    DECLARE C2 CURSOR FOR
    SELECT EMPNO, LASTNAME, WORKDEPT
      FROM CORPDATA.EMPLOYEE
    WHERE EMPNO > :LAST-EMPNO
    ORDER BY EMPNO
END-EXEC.

EXEC SQL
    OPEN C2
END-EXEC.

*    PERFORM FETCH-C21-PARA  20 TIMES.

*    Show the display with these 20 rows of data.

EXEC SQL
    CLOSE C2
END-EXEC.
```

In the above example, notice that an additional cursor had to be opened to continue the list and to get current data. This can result in creating an additional ODP that increases the processing time on the iSeries server. In place of the above example, the programmer can design the application specifying ALWCPYDTA(*NO) with the following SQL statements:

```
EXEC SQL
    DECLARE C1 CURSOR FOR
    SELECT EMPNO, LASTNAME, WORKDEPT
      FROM CORPDATA.EMPLOYEE
     ORDER BY EMPNO
END-EXEC.

EXEC SQL
    OPEN C1
END-EXEC.

*    Display the screen with these 20 rows of data.

*    PERFORM FETCH-C1-PARA  20 TIMES.

*    Show the display and wait for the user to indicate that
*    the next 20 rows should be displayed.

*    PERFORM FETCH-C1-PARA  20 TIMES.

EXEC SQL
    CLOSE C1
END-EXEC.
```

In the above example, the query might perform better if the FOR 20 ROWS clause was used on the multiple-row FETCH statement. Then, the 20 rows are retrieved in one operation.

Related information

Start SQL (STRSQL) command

Reduce the number of open operations

The SQL data manipulation language statements must do database open operations in order to create an open data path (ODP) to the data. An open data path is the path through which all input/output operations for the table are performed. In a sense, it connects the SQL application to a table. The number of open operations in a program can significantly affect performance.

A database open operation occurs on:

- An OPEN statement
- SELECT INTO statement
- An INSERT statement with a VALUES clause
- An UPDATE statement with a WHERE condition
- An UPDATE statement with a WHERE CURRENT OF cursor and SET clauses that refer to operators or functions
- SET statement that contains an expression
- VALUES INTO statement that contains an expression
- A DELETE statement with a WHERE condition

An INSERT statement with a select-statement requires two open operations. Certain forms of subqueries may also require one open per subselect.

To minimize the number of opens, DB2 Universal Database for iSeries leaves the open data path (ODP) open and reuses the ODP if the statement is run again, unless:

- The ODP used a host variable to build a subset temporary index. The i5/OS database support may choose to build a temporary index with entries for only the rows that match the row selection specified

in the SQL statement. If a host variable was used in the row selection, the temporary index will not have the entries required for a different value contained in the host variable.
- Ordering was specified on a host variable value.
- An Override Database File (OVRDBF) or Delete Override (DLTOVR) CL command has been issued since the ODP was opened, which affects the SQL statement execution. The ODPs opened by DB2 Universal Database for iSeries

  Note: Only overrides that affect the name of the table being referred to will cause the ODP to be closed within a given program invocation.
- The join is a complex join that requires temporaries to contain the intermediate steps of the join.
- Some cases involve a complex sort, where a temporary file is required, may not be reusable.
- A change to the library list since the last open has occurred, which changes the table selected by an unqualified referral in system naming mode.
- The join was implemented by the CQE optimizer using hash join.

For embedded static SQL, DB2 Universal Database for iSeries only reuses ODPs opened by the same statement. An identical statement coded later in the program does not reuse an ODP from any other statement. If the identical statement must be run in the program many times, code it once in a subroutine and call the subroutine to run the statement.

The ODPs opened by DB2 Universal Database for iSeries are closed when any of the following occurs:
- A CLOSE, INSERT, UPDATE, DELETE, or SELECT INTO statement completes and the ODP required a temporary result that was not reusable or a subset temporary index.
- The Reclaim Resources (RCLRSC) command is issued. A 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) will not close 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
- When the last program that contains SQL statements on the call stack exits, except for ODPs created for programs precompiled using CLOSQLCSR(*ENDJOB) or modules precompiled using CLOSQLCSR(*ENDACTGRP).
- When a CONNECT (Type 1) statement changes the application server for an activation group, all ODPs created for the activation group are closed.
- When a DISCONNECT statement ends a connection to the application server, all ODPs for that application server are closed.
- When a released connection is ended by a successful COMMIT, all ODPs for that application server are closed.
- When the threshold for open cursors specified by the query options file (QAQQINI) parameter OPEN_CURSOR_THRESHOLD is reached.
- The SQL LOCK TABLE or CL ALCOBJ OBJ((filename *FILE *EXCL)) CONFLICT(*RQSRLS) command will close any psuedo-closed cursors associated with the specified table.

You can control whether the system keeps the ODPs open in the following ways:
- Design the application so a program that issues an SQL statement is always on the call stack
- Use the CLOSQLCSR(*ENDJOB) or CLOSQLCSR(*ENDACTGRP) parameter
- By specifying the OPEN_CURSOR_THRESHOLD and OPEN_CURSOR_CLOSE_COUNT parameters of the query options file (QAQQINI)

The system does an open operation for the first execution of each UPDATE WHERE CURRENT OF when any expression in the SET clause contains an operator or function. The open can be avoided by coding the function or operation in the host language code.

For example, the following UPDATE causes the system to do an open operation:

```
EXEC SQL
 FETCH EMPT INTO :SALARY
END-EXEC.

EXEC SQL
 UPDATE CORPDATA.EMPLOYEE
   SET SALARY = :SALARY + 1000
   WHERE CURRENT OF EMPT
END-EXEC.
```

Instead, use the following coding technique to avoid opens:

```
EXEC SQL
 FETCH EMPT  INTO  :SALARY
END EXEC.

ADD 1000 TO SALARY.

EXEC SQL
 UPDATE CORPDATA.EMPLOYEE
   SET SALARY = :SALARY
   WHERE CURRENT OF EMPT
END-EXEC.
```

You can determine whether SQL statements result in full opens in several ways. The preferred methods are to use the Database Monitor or by looking at the messages issued while debug is active. You can also use the CL commands Trace Job (TRCJOB) or Display Journal (DSPJRN).

Related information

Reclaim Resources (RCLRSC) command

Trace Job (TRCJOB) command

Display Journal (DSPJRN) command

ILE RPG

ILE COBOL

C and C++

Retain 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**

This is the default for all non-ILE precompilers. With this option, a cursor remains open and accessible only while the program that opened it is on the call stack. When the program ends, the

SQL cursor can no longer be used. Prepared statements are also lost when the program ends. Locks, however, remain until the last SQL program on the call stack has completed.

***ENDSQL**
: With this option, SQL cursors and prepared statements that are created by a program remain open until the last SQL program on the call stack has completed. They cannot be used by other programs, only by a different call to the same program. Locks remain until the last SQL program in the call stack completes.

***ENDJOB**
: This option allows you to keep SQL cursors, prepared statements, and locks active for the duration of the job. When the last SQL program on the stack has completed, any SQL resources created by *ENDJOB programs are still active. The locks remain in effect. The SQL cursors that were not explicitly closed by the CLOSE, COMMIT, or ROLLBACK statements remain open. The prepared statements are still usable on subsequent calls to the same program.

Related reference

“Effects of precompile options on database performance” on page 171
Several precompile options are available for creating SQL programs with improved performance. They are only options because using them may impact the function of the application. For this reason, the default value for these parameters is the value that will ensure successful migration of applications from prior releases. However, you can improve performance by specifying other options.

Retaining cursor positions across ILE program calls

For ILE program calls, the close SQL cursor (CLOSQLCSR) parameter allows you to specify the scope of the following:

- The cursors
- The prepared statements
- The locks

When used properly, the CLOSQLCSR parameter can reduce the number of SQL OPEN, PREPARE, and LOCK statements needed. It can also simplify applications by allowing you to retain cursor positions across program calls.

***ENDACTGRP**
: This is the default for the ILE precompilers. With this option, SQL cursors and prepared statements remain open until the activation group that the program is running under ends. They cannot be used by other programs, only by a different call to the same program. Locks remain until the activation group ends.

***ENDMOD**
: With this option, a cursor remains open and accessible only while the module that opened it is active. When the module ends, the SQL cursor can no longer be used. Prepared statements will also be lost when the module ends. Locks, however, remain until the last SQL program in the call stack completes.

General rules for retaining cursor positions for all program calls

When using programs compiled with either CLOSQLCSR(*ENDPGM) or CLOSQLCSR(*ENDMOD), a cursor must be opened every time the program or module is called, in order to access the data. If the SQL program or module is going to be called several times, and you want to take advantage of a reusable ODP, then the cursor must be explicitly closed before the program or module exits.

Using the CLOSQLCSR parameter and specifying *ENDSQL, *ENDJOB, or *ENDACTGRP, you may not need to run an OPEN and a CLOSE statement on every call. In addition to having fewer statements to run, you can maintain the cursor position between calls to the program or module.

The following examples of SQL statements help demonstrate the advantage of using the CLOSQLCSR parameter:

```
EXEC SQL
 DECLARE DEPTDATA CURSOR FOR
   SELECT EMPNO, LASTNAME
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = :DEPTNUM
END-EXEC.

EXEC SQL
 OPEN DEPTDATA
END-EXEC.

EXEC SQL
 FETCH DEPTDATA INTO :EMPNUM, :LNAME
END-EXEC.

EXEC SQL
 CLOSE DEPTDATA
END-EXEC.
```

If this program is called several times from another SQL program, it will be able to use a reusable ODP. This means that, as long as SQL remains active between the calls to this program, the OPEN statement will not require a database open operation. However, the cursor is still positioned to the first result row after each OPEN statement, and the FETCH statement will always return the first row.

In the following example, the CLOSE statement has been removed:

```
EXEC SQL
 DECLARE DEPTDATA CURSOR FOR
   SELECT EMPNO, LASTNAME
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = :DEPTNUM
END-EXEC.

  IF CURSOR-CLOSED IS = TRUE THEN
EXEC SQL
 OPEN DEPTDATA
END-EXEC.

EXEC SQL
 FETCH DEPTDATA INTO :EMPNUM, :LNAME
END-EXEC.
```

If this program is precompiled with the *ENDJOB option or the *ENDACTGRP option and the activation group remains active, the cursor position is maintained. The cursor position is also maintained when the following occurs:

- The program is precompiled with the *ENDSQL option.
- SQL remains active between program calls.

The result of this strategy is that each call to the program retrieves the next row in the cursor. On subsequent data requests, the OPEN statement is unnecessary and, in fact, fails with a -502 SQLCODE. You can ignore the error, or add code to skip the OPEN. You can do this by using a FETCH statement first, and then running the OPEN statement only if the FETCH operation failed.

This technique also applies to prepared statements. A program can first try the EXECUTE, and if it fails, perform the PREPARE. The result is that the PREPARE is only needed on the first call to the program, assuming the correct CLOSQLCSR option was chosen. Of course, if the statement can change between calls to the program, it should perform the PREPARE in all cases.

The main program might also control this by sending a special parameter on the first call only. This special parameter value indicates that because it is the first call, the subprogram should perform the OPENs, PREPAREs, and LOCKs.

Note: If you are using COBOL programs, do not use the STOP RUN statement. When the first COBOL program on the call stack ends or a STOP RUN statement runs, a reclaim resource (RCLRSC) operation is done. This operation closes the SQL cursor. The *ENDSQL option does not work as you wanted.

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 OPTIMIZE clause above are 1–9999999 or ALL.

The optimizer calculates the following costs.

The optimize ratio = optimize for n rows value / estimated number of rows in answer set.

```
Cost using a temporarily created index:

               Cost to retrieve answer set rows
            +  Cost to create the index
            +  Cost to retrieve the rows again
                  with a temporary index       * optimize ratio

Cost using a SORT:

               Cost to retrieve answer set rows
            +  Cost for SORT input processing
            +  Cost for SORT output processing  * optimize ratio


Cost using an existing index:

               Cost to retrieve answer set rows
               using an existing index          * optimize ratio
```

In the previous examples, the estimated cost to sort or to create an index is not adjusted by the optimize ratio. This enables the optimizer to balance the optimization and preprocessing requirements. If the optimize number is larger than the number of rows in the result table, no adjustments are made to the cost estimates. If the OPTIMIZE clause is not specified for a query, a default value is used based on the statement type, value of ALWCPYDTA specified, or output device.

Statement Type	ALWCPYDTA(*OPTIMIZE)	ALWCPYDTA(*YES or *NO)
DECLARE CURSOR	The number or rows in the result table.	3% or the number of rows in the result table.
Embedded Select	2	2

Statement Type	ALWCPYDTA(*OPTIMIZE)	ALWCPYDTA(*YES or *NO)
INTERACTIVE Select output to display	3% or the number of rows in the result table.	3% or the number of rows in the result table.
INTERACTIVE Select output to printer or database table	The number of rows in the result table.	The number of rows in the result table.

The OPTIMIZE clause influences the optimization of a query:
- To use an existing index (by specifying a small number).
- To enable the creation of an index or to run a sort or a hash by specifying a large number of possible rows in the answer set.

Related information

select-statement

Use FETCH FOR n ROWS

Applications that perform many FETCH statements in succession may be improved by using FETCH FOR n ROWS. With this clause, you can retrieve multiple rows of data from a table and put them into a host structure array or row storage area with a single FETCH.

An SQL application that uses a FETCH statement without the FOR n ROWS clause can be improved by using the multiple-row FETCH statement to retrieve multiple rows. After the host structure array or row storage area has been filled by the FETCH, the application can loop through the data in the array or storage area to process each of the individual rows. The statement runs faster because the SQL run-time was called only once and all the data was simultaneously returned to the application program.

You can change the application program to allow the database manager to block the rows that the SQL run-time retrieves from the tables.

In the following table, the program attempted to FETCH 100 rows into the application. Note the differences in the table for the number of calls to SQL run-time and the database manager when blocking can be performed.

Table 35. Number of Calls Using a FETCH Statement

	Database Manager Not Using Blocking	Database Manager Using Blocking
Single-Row FETCH Statement	100 SQL calls 100 database calls	100 SQL calls 1 database call
Multiple-Row FETCH Statement	1 SQL run-time call 100 database calls	1 SQL run-time call 1 database call

Related information

FETCH statement

Improve SQL blocking performance when using FETCH FOR n ROWS

Special performance considerations should be made for the following points when using FETCH FOR n ROWS.

You can improve SQL blocking performance with the following:
- The attribute information in the host structure array or the descriptor associated with the row storage area should match the attributes of the columns retrieved.
- The application should retrieve as many rows as possible with a single multiple-row FETCH call. The blocking factor for a multiple-row FETCH request is not controlled by the system page sizes or the SEQONLY parameter on the OVRDBF command. It is controlled by the number of rows that are requested on the multiple-row FETCH request.

- Single- and multiple-row FETCH requests against the same cursor should not be mixed within a program. If one FETCH against a cursor is treated as a multiple-row FETCH, all fetches against that cursor are treated as multiple-row fetches. In that case, each of the single-row FETCH requests is treated as a multiple-row FETCH of one row.
- The PRIOR, CURRENT, and RELATIVE scroll options should not be used with multiple-row FETCH statements. To allow random movement of the cursor by the application, the database manager must maintain the same cursor position as the application. Therefore, the SQL run-time treats all FETCH requests against a scrollable cursor with these options specified as multiple-row FETCH requests.

Use INSERT n ROWS

Applications that perform many INSERT statements in succession may be improved by using INSERT n ROWS. With this clause, you can insert one or more rows of data from a host structure array into a target table. This array must be an array of structures where the elements of the structure correspond to columns in the target table.

An SQL application that loops over an INSERT...VALUES statement (without the n ROWS clause) can be improved by using the INSERT n ROWS statement to insert multiple rows into the table. After the application has looped to fill the host array with rows, a single INSERT n ROWS statement can be run to insert the entire array into the table. The statement runs faster because the SQL run-time was only called once and all the data was simultaneously inserted into the target table.

In the following table, the program attempted to INSERT 100 rows into a table. Note the differences in the number of calls to SQL run-time and to the database manager when blocking can be performed.

Table 36. Number of Calls Using an INSERT Statement

	Database Manager Not Using Blocking	Database Manager Using Blocking
Single-Row INSERT Statement	100 SQL run-time calls 100 database calls	100 SQL run-time calls 1 database call
Multiple-Row INSERT Statement	1 SQL run-time call 100 database calls	1 SQL run-time call 1 database call

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 key columns for the index used by the cursor and by the number of key columns, if any, that are null capable.
- COMMIT(*CS) is specified, and ALWBLK(*ALLREAD) is not specified.
- COMMIT(*ALL) is specified, and the following are true:
  - A SELECT INTO statement or a blocked FETCH statement is not used
  - The query does not use column functions or specify group by columns.

– A temporary result table does not need to be created.

- COMMIT(*CHG) is specified, and ALWBLK(*ALLREAD) is not specified.
- The cursor contains at least one subquery and the outermost subselect provided a correlated reference for a subquery or the outermost subselect processed a subquery with an IN, = ANY, or < > ALL subquery predicate operator, which is treated as a correlated reference, and that subquery is not isolatable.

The SQL run-time automatically blocks rows with the database manager in the following cases:

- INSERT

  If an INSERT statement contains a select-statement, inserted rows are blocked and not actually inserted into the target table until the block is full. The SQL run-time automatically does blocking for blocked inserts.

  Note: If an INSERT with a VALUES clause is specified, the SQL run-time might not actually close the internal cursor that is used to perform the inserts until the program ends. If the same INSERT statement is run again, a full open is not necessary and the application runs much faster.
- OPEN

  Blocking is done under the OPEN statement when the rows are retrieved if all of the following conditions are true:

  – The cursor is only used for FETCH statements.

  – No EXECUTE or EXECUTE IMMEDIATE statements are in the program, or ALWBLK(*ALLREAD) was specified, or the cursor is declared with the FOR FETCH ONLY clause.

  – COMMIT(*CHG) and ALWBLK(*ALLREAD) are specified, COMMIT(*CS) and ALWBLK(*ALLREAD) are specified, or COMMIT(*NONE) is specified.

Related reference

"Effects of precompile options on database performance" on page 171
Several precompile options are available for creating SQL programs with improved performance. They are only options because using them may impact the function of the application. For this reason, the default value for these parameters is the value that will ensure successful migration of applications from prior releases. However, you can improve performance by specifying other options.

Related information

Override Database File (OVRDBF) command

Optimize the number of columns that are selected with SELECT statements

The number of columns that you specify in the select list of a SELECT statement causes the database manager to retrieve the data from the underlying tables and map the data into host variables in the application programs. By minimizing the number of columns that are specified, processing unit resource usage can be conserved.

Even though it is convenient to code SELECT *, it is far better to explicitly code the columns that are actually required for the application. This is especially important if index-only access is wanted or if all of the columns will participate in a sort operation (as happens for SELECT DISTINCT and for SELECT UNION).

This is also important when considering index only access, since you minimize the number of columns in a query and thereby increase the odds that an index can be used to completely satisfy the request for all the data.

Related information

select-statement

Eliminate redundant validation with SQL PREPARE statements

The processing which occurs when an SQL PREPARE statement is run is similar to the processing which occurs during precompile processing.

The following processing occurs for the statement that is being prepared:

- The syntax is checked.
- The statement is validated to ensure that the usage of objects are valid.
- An access plan is built.

Again when the statement is executed or opened, the database manager will re-validate that the access plan is still valid. Much of this open processing validation is redundant with the validation which occurred during the PREPARE processing. The DLYPRP(*YES) parameter specifies whether PREPARE statements in this program will completely validate the dynamic statement. The validation will be completed when the dynamic statement is opened or executed. This parameter can provide a significant performance enhancement for programs which use the PREPARE SQL statement because it eliminates redundant validation. Programs that specify this precompile option should check the SQLCODE and SQLSTATE after running the OPEN or EXECUTE statement to ensure that the statement is valid. DLYPRP(*YES) will not provide any performance improvement if the INTO clause is used on the PREPARE statement or if a DESCRIBE statement uses the dynamic statement before an OPEN is issued for the statement.

Related reference

"Effects of precompile options on database performance" on page 171
Several precompile options are available for creating SQL programs with improved performance. They are only options because using them may impact the function of the application. For this reason, the default value for these parameters is the value that will ensure successful migration of applications from prior releases. However, you can improve performance by specifying other options.

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 which dynamically retrieves the latest data directly from the table. Paging performance can be improved by specifying REFRESH(*FORWARD).

When interactively displaying data using REFRESH(*FORWARD), the results of a select-statement are copied to a temporary table as you page forward through the display. Other users sharing the table can make changes to the rows while you are displaying the select-statement results. If you page backward or forward to rows that have already been displayed, the rows shown are those in the temporary table instead of those in the updated table.

The refresh option can be changed on the Session Services display.

Related information

Start SQL (STRSQL) command

General DB2 UDB for iSeries 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 may impact the function of the application. For this reason, the default value for these parameters is the value that will ensure successful migration of applications from prior releases. However, you can improve performance by specifying other options.

The following table shows these precompile options and their performance impacts.

Some of these options may be suitable for most of your applications. Use the command CRTDUPOBJ to create a copy of the SQL CRTSQLxxx command. and the CHGCMDDFT command to customize the optimal values for the precompile parameters. The DSPPGM, DSPSRVPGM, DSPMOD, or PRTSQLINF commands can be used to show the precompile options that are used for an existing program object.

Precompile Option	Optimal Value	Improvements	Considerations
ALWCPYDTA	*OPTIMIZE (the default)	Queries where the ordering or grouping criteria conflicts with the selection criteria.	A copy of the data may be made when the query is opened.
ALWBLK	*ALLREAD (the default)	Additional read-only cursors use blocking.	ROLLBACK HOLD may not change the position of a read-only cursor. Dynamic processing of positioned updates or deletes might fail.
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 may 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 will take advantage of performance enhancements available in the current release.	The program object cannot be used on a server from a previous release.

Related reference

"Effects of the ALWCPYDTA parameter on database performance" on page 172
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 168
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 163
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 170
The processing which occurs when an SQL PREPARE statement is run is similar to 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 ''AOO'')
        KEYFLD(LASTNAME)
```

In the above example when ALWCPYDTA(*NO) or ALWCPYDTA(*YES) is specified, the database manager may try to create an index from the first index with a column named LASTNAME, if such an index exists. The rows in the table are scanned, using the index, to select only the rows matching the WHERE condition.

If ALWCPYDTA(*OPTIMIZE) is specified, the database manager uses an index with the first index column of WORKDEPT. It then makes a copy of all of the rows that match the WHERE condition. Finally, it may sort the copied rows by the values in LASTNAME. This row selection processing is significantly more efficient, because the index used immediately locates the rows to be selected.

ALWCPYDTA(*OPTIMIZE) optimizes the total time that is required to process the query. However, the time required to receive the first row may be increased because a copy of the data must be made before returning the first row of the result table. This initial change in response time may be important for applications that are presenting interactive displays or that retrieve only the first few rows of the query. The DB2 Universal Database for iSeries query optimizer can be influenced to avoid sorting by using the OPTIMIZE clause.

Queries that involve a join operation may also benefit from ALWCPYDTA(*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 171
Several precompile options are available for creating SQL programs with improved performance. They are only options because using them may impact the function of the application. For this reason, the default value for these parameters is the value that will ensure successful migration of applications from prior releases. However, you can improve performance by specifying other options.

"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 of the rows in the index will be sequentially processed, but the resulting row numbers will be 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 Radix Index Scan is that the rows being returned must first be identified by a probe operation to subset the rows being retrieved.

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 should be wide enough to incorporate at least 90% to 95% of the values for the column.

It is possible to balance space savings and performance. In the following example of an electronic telephone book, the following data is used:

- 8600 names that are identified by: last, first, and middle name
- The Last, First, and Middle columns are variable length.
- The shortest last name is 2 characters; the longest is 22 characters.

This example shows how space can be saved by using variable-length columns. The fixed-length column table uses the most space. The table with the carefully calculated allocate sizes uses less disk space. The table that was defined with no allocate size (with all of the data stored in the overflow area) uses the least disk space.

Variety of Support	Last Name Max/Alloc	First Name Max/Alloc	Middle Name Max/Alloc	Total Physical File Size	Number of Rows in Overflow Space
Fixed Length	22	22	22	567 K	0
Variable Length	40/10	40/10	40/7	408 K	73
Variable-Length Default	40/0	40/0	40/0	373 K	8600

In many applications, performance must be considered. If you use the default ALLOCATE(0), it will double the disk unit traffic. ALLOCATE(0) requires two reads; one to read the fixed-length portion of the row and one to read the overflow space. The variable-length implementation, with the carefully chosen ALLOCATE, minimizes overflow and space and maximizes performance. The size of the table is 28% smaller than the fixed-length implementation. Because 1% of rows are in the overflow area, the access requiring two reads is minimized. The variable-length implementation performs about the same as the fixed-length implementation.

To create the table using the ALLOCATE keyword:

```
CREATE TABLE PHONEDIR
         (LAST    VARCHAR(40) ALLOCATE(10),
          FIRST   VARCHAR(40) ALLOCATE(10),
          MIDDLE  VARCHAR(40) ALLOCATE(7))
```

If you are using host variables to insert or update variable-length columns, the host variables should be variable length. Because blanks are not truncated from fixed-length host variables, using fixed-length host variables can cause more rows to spill into the overflow space. This increases the size of the table.

In this example, fixed-length host variables are used to insert a row into a table:

```
01  LAST-NAME PIC X(40).
   ...
    MOVE "SMITH" TO LAST-NAME.
    EXEC SQL
      INSERT INTO PHONEDIR
       VALUES(:LAST-NAME, :FIRST-NAME, :MIDDLE-NAME, :PHONE)
    END-EXEC.
```

The host-variable LAST-NAME is not variable length. The string “SMITH”, followed by 35 blanks, is inserted into the VARCHAR column LAST. The value is longer than the allocate size of 10. Thirty of thirty-five trailing blanks are in the overflow area.

In this example, variable-length host variables are used to insert a row into a table:

```
01  VLAST-NAME.
    49 LAST-NAME-LEN PIC S9(4) BINARY.
    49 LAST-NAME-DATA PIC X(40).
   ...
    MOVE "SMITH" TO LAST-NAME-DATA.
    MOVE 5 TO LAST-NAME-LEN.
    EXEC SQL
      INSERT INTO PHONEDIR
     VALUES(:VLAST-NAME, :VFIRST-NAME, :VMIDDLE-NAME, :PHONE)
    END-EXEC.
```

The host variable VLAST-NAME is variable length. The actual length of the data is set to 5. The value is shorter than the allocated length. It can be placed in the fixed portion of the column.

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 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 of 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, you may 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

Database monitor: Formats

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

Database monitor SQL table format

The following figure shows the format used to create the QSYS/QAQQDBMN performance statistics table, that is shipped with the system.

```
  CREATE TABLE QSYS/QQQDBMN (
 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) NOT NULL DEFAULT '' ,
 QQSYS CHAR(8) NOT NULL DEFAULT '' ,
 QQJOB CHAR(10) NOT NULL DEFAULT '' ,
 QQUSER CHAR(10) NOT NULL DEFAULT '' ,
 QQJNUM CHAR(6) NOT NULL DEFAULT '' ,
 QQUCNT DECIMAL(15, 0) ,
 QQUDEF VARCHAR(100) ,
 QQSTN DECIMAL(15, 0) ,
 QQQDTN DECIMAL(15, 0) ,
 QQQDTL DECIMAL(15, 0) ,
 QQMATN DECIMAL(15, 0) ,
 QQMATL DECIMAL(15, 0) ,
 QQTLN CHAR(10) ,
 QQTFN CHAR(10) ,
 QQTMN CHAR(10) ,
 QQPTLN CHAR(10) ,
 QQPTFN CHAR(10) ,
 QQPTMN CHAR(10) ,
 QQILNM CHAR(10) ,
 QQIFNM CHAR(10) ,
 QQIMNM CHAR(10) ,
 QQNTNM CHAR(10) ,
 QQNLNM CHAR(10) ,
 QQSTIM TIMESTAMP ,
 QQETIM TIMESTAMP ,
 QQKP CHAR(1) ,
 QQKS CHAR(1) ,
 QQTOTR DECIMAL(15, 0) ,
 QQTMPR DECIMAL(15, 0) ,
 QQJNP DECIMAL(15, 0) ,
 QQEPT DECIMAL(15, 0) ,
 QQDSS CHAR(1) ,
 QQIDXA CHAR(1) ,
 QQORDG CHAR(1) ,
 QQGRPG CHAR(1) ,
 QQJNG CHAR(1) ,
 QQUNIN CHAR(1) ,
 QQSUBQ CHAR(1) ,
 QQHSTV CHAR(1) ,
 QQRCDS CHAR(1) ,
 QQRCOD CHAR(2) ,
 QQRSS DECIMAL(15, 0) ,
 QQREST DECIMAL(15, 0) ,
 QQRIDX DECIMAL(15, 0) ,
 QQFKEY DECIMAL(15, 0) ,
 QQKSEL DECIMAL(15, 0) ,
 QQAJN DECIMAL(15, 0) ,
 QQIDXD VARCHAR(1000) ALLOCATE(48) ,
 QQC11 CHAR(1) ,
 QQC12 CHAR(1) ,
 QQC13 CHAR(1) ,
 QQC14 CHAR(1) ,
 QQC15 CHAR(1) ,
 QQC16 CHAR(1) ,
```

```
QQC18 CHAR(1) ,
QQC21 CHAR(2) ,
QQC22 CHAR(2) ,
QQC23 CHAR(2) ,
QQI1 DECIMAL(15, 0) ,
QQI2 DECIMAL(15, 0) ,
QQI3 DECIMAL(15, 0) ,
QQI4 DECIMAL(15, 0) ,
QQI5 DECIMAL(15, 0) ,
QQI6 DECIMAL(15, 0) ,
QQI7 DECIMAL(15, 0) ,
QQI8 DECIMAL(15, 0) ,
QQI9 DECIMAL(15, 0) ,
QQIA DECIMAL(15, 0) ,
QQF1 DECIMAL(15, 0) ,
QQF2 DECIMAL(15, 0) ,
QQF3 DECIMAL(15, 0) ,
QQC61 CHAR(6) ,
QQC81 CHAR(8) ,
QQC82 CHAR(8) ,
QQC83 CHAR(8) ,
QQC84 CHAR(8) ,
QQC101 CHAR(10) ,
QQC102 CHAR(10) ,
QQC103 CHAR(10) ,
QQC104 CHAR(10) ,
QQC105 CHAR(10) ,
QQC106 CHAR(10) ,
QQC181 CHAR(18) ,
QQC182 CHAR(18) ,
QQC183 CHAR(18) ,
QQC301 VARCHAR(30) ALLOCATE(10) ,
QQC302 VARCHAR(30) ALLOCATE(10) ,
QQC303 VARCHAR(30) ALLOCATE(10) ,
QQ1000 VARCHAR(1000) ALLOCATE(48) ,
QQTIM1 TIMESTAMP ,
QQTIM2 TIMESTAMP ,
QVQTBL VARCHAR(128) ALLOCATE(10) ,
QVQLIB VARCHAR(128) ALLOCATE(10) ,
QVPTBL VARCHAR(128) ALLOCATE(10) ,
QVPLIB VARCHAR(128) ALLOCATE(10) ,
QVINAM VARCHAR(128) ALLOCATE(10) ,
QVILIB VARCHAR(128) ALLOCATE(10) ,
QVQTBLI CHAR(1) ,
QVPTBLI CHAR(1) ,
QVINAMI CHAR(1) ,
QVBNDY CHAR(1) ,
QVJFANO CHAR(1) ,
QVPARPF CHAR(1) ,
QVPARPL CHAR(1) ,
QVC11 CHAR(1) ,
QVC12 CHAR(1) ,
QVC13 CHAR(1) ,
QVC14 CHAR(1) ,
QVC15 CHAR(1) ,
QVC16 CHAR(1) ,
QVC17 CHAR(1) ,
QVC18 CHAR(1) ,
QVC19 CHAR(1) ,
QVC1A CHAR(1) ,
QVC1B CHAR(1) ,
QVC1C CHAR(1) ,
QVC1D CHAR(1) ,
QVC1E CHAR(1) ,
QVC1F CHAR(1) ,
QWC11 CHAR(1) ,
QWC12 CHAR(1) ,
```

```
QWC13 CHAR(1) ,
QWC14 CHAR(1) ,
QWC15 CHAR(1) ,
QWC16 CHAR(1) ,
QWC17 CHAR(1) ,
QWC18 CHAR(1) ,
QWC19 CHAR(1) ,
QWC1A CHAR(1) ,
QWC1B CHAR(1) ,
QWC1C CHAR(1) ,
QWC1D CHAR(1) ,
QWC1E CHAR(1) ,
QWC1F CHAR(1) ,
QVC21 CHAR(2) ,
QVC22 CHAR(2) ,
QVC23 CHAR(2) ,
QVC24 CHAR(2) ,
QVCTIM DECIMAL(15, 0) ,
QVPARD DECIMAL(15, 0) ,
QVPARU DECIMAL(15, 0) ,
QVPARRC DECIMAL(15, 0) ,
QVRCNT DECIMAL(15, 0) ,
QVFILES DECIMAL(15, 0) ,
QVP151 DECIMAL(15, 0) ,
QVP152 DECIMAL(15, 0) ,
QVP153 DECIMAL(15, 0) ,
QVP154 DECIMAL(15, 0) ,
QVP155 DECIMAL(15, 0) ,
QVP156 DECIMAL(15, 0) ,
QVP157 DECIMAL(15, 0) ,
QVP158 DECIMAL(15, 0) ,
QVP159 DECIMAL(15, 0) ,
QVP15A DECIMAL(15, 0) ,
QVP15B DECIMAL(15, 0) ,
QVP15C DECIMAL(15, 0) ,
QVP15D DECIMAL(15, 0) ,
QVP15E DECIMAL(15, 0) ,
QVP15F DECIMAL(15, 0) ,
QVC41 CHAR(4) ,
QVC42 CHAR(4) ,
QVC43 CHAR(4) ,
QVC44 CHAR(4) ,
QVC81 CHAR(8) ,
QVC82 CHAR(8) ,
QVC83 CHAR(8) ,
QVC84 CHAR(8) ,
QVC85 CHAR(8) ,
QVC86 CHAR(8) ,
QVC87 CHAR(8) ,
QVC88 CHAR(8) ,
QVC101 CHAR(10) ,
QVC102 CHAR(10) ,
QVC103 CHAR(10) ,
QVC104 CHAR(10) ,
QVC105 CHAR(10) ,
QVC106 CHAR(10) ,
QVC107 CHAR(10) ,
QVC108 CHAR(10) ,
QVC1281 VARCHAR(128) ALLOCATE(10) ,
QVC1282 VARCHAR(128) ALLOCATE(10) ,
QVC1283 VARCHAR(128) ALLOCATE(10) ,
QVC1284 VARCHAR(128) ALLOCATE(10) ,
QVC3001 VARCHAR(300) ALLOCATE(32) ,
QVC3002 VARCHAR(300) ALLOCATE(32) ,
QVC3003 VARCHAR(300) ALLOCATE(32) ,
QVC3004 VARCHAR(300) ALLOCATE(32) ,
QVC3005 VARCHAR(300) ALLOCATE(32) ,
```

```
QVC3006 VARCHAR(300) ALLOCATE(32) ,
QVC3007 VARCHAR(300) ALLOCATE(32) ,
QVC3008 VARCHAR(300) ALLOCATE(32) ,
QVC5001 VARCHAR(500) ALLOCATE(32) ,
QVC5002 VARCHAR(500) ALLOCATE(32) ,
QVC1000 VARCHAR(1000) ALLOCATE(48) ,
QWC1000 VARCHAR(1000) ALLOCATE(48) ,
QQINT01 INTEGER ,
QQINT02 INTEGER ,
 QQINT03 INTEGER ,
QQINT04 INTEGER ,
QQSMINT1 SMALLINT ,
QQSMINT2 SMALLINT ,
QQSMINT3 SMALLINT ,
QQSMINT4 SMALLINT ,
QQSMINT5 SMALLINT ,
QQSMINT6 SMALLINT ,
  QQ1000L CLOB(2M) ALLOCATE(48) ) ;

 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' ,
    QQSYS   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' ,
      QQRCDS  TEXT IS 'Row Selection' ,
      QQRCOD  TEXT IS 'Reason Code' ,
      QQRSS   TEXT IS 'Number of rows selected or sorted' ,
      QQREST  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' ,
      QQIDXD  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, Libary 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' ,
      QVPARPF TEXT IS 'Parallel Pre-Fetch' ,
      QVPARPL 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

The following 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 server, 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

```
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,
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,
QVC1000 as SQL_Path,
QWC1000 as SQL_Path_2,
QVC5001 as SQL_Path_3,
QVC5002 as SQL_Path_4,
QVC3001 as SQL_Path_5,
QVC3002 as SQL_Path_6,
QVC3003 as SQL_Path_7,
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,
        QQSMINT2 as IP_Port_Number,
        QVC3004 as NTS_Transaction_Id,
        QQSMINT3 as NTS_Format_Id_Length,
        QQSMINT4 as NTS_Transatction_ID_SubLength,
        QVRCNT as Unique_Refresh_Counter2,
        QVP15F as Times_Run,
        QVP15E as Full_Opens
 FROM   DbMonLib/DbMonTable
 WHERE  QQRID=1000)
```

Table 37. 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)
Statement_Function	QQC11	Statement function: • S - Select • U - Update • I - Insert • D - Delete • L - Data definition language • O - Other

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Statement_Operation	QQC21	Statement operation: • AD - Allocate descriptor • AL - Alter table • AP - Alter procedure • AQ - Alter sequence • CA - Call • CC - Create collection • CD - Create type • CF - Create function • CG - Create trigger • CI - Create index • CL - Close • CM - Commit • CN - Connect • CO - Comment on • CP - Create procedure • CQ - Create sequence • CS - Create alias/synonym • CT - Create table • CV - Create view • DA - Deallocate descriptor • DE - Describe • DI - Disconnect • DL - Delete • DM - Describe parameter marker • DP - Declare procedure • DR - Drop • DT - Describe table • EI - Execute immediate • EX - Execute • FE - Fetch • FL - Free locator • GR - Grant • GS - Get descriptor • HC - Hard close • HL - Hold locator

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Statement_Operation (continued)	QQC21	• IN - Insert • JR - Server job reused • LK - Lock • LO - Label on • MT - More text (Deprecated in V5R4) • OP - Open • PD - Prepare and describe • PR - Prepare • 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 • 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: • D - Dynamic statement • S - Static statement
Parse_Required	QQC13	Parse required (Y/N)
Package_Name	QQC103	Name of the package or name of the program that contains the current SQL statement

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
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 • S - Successful • U - Unsuccessful
Result_Rows	QQI2	Number of result rows returned
Dynamic_Replan_Reason_Code	QQC22	Dynamic replan (access plan rebuilt) • 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: – Object was deleted and recreated. – Object was saved and restored. – Library list was changed. – Object was renamed. – Object was moved. – Object was overridden to a different object. – This is the first run of this query after the object containing the query has been restored. • A2 - Access plan was built to use a reusable Open Data Path (ODP) and the optimizer chose to use a non-reusable ODP for this call. • A3 - Access plan was built to use a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP for this call. • A4 - The number of rows in the table member has changed by more than 10% since the access plan was last built. • A5 - A new index exists over one of the tables in the query. • A6 - An index that was used for this access plan no longer exists or is no longer valid. • A7 - i5/OS 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: – date format – date separator – time format – time separator

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Dynamic_Replan_Reason_Code (continued)	QQC22	• AA - The sort sequence table specified is different than the sort sequence table that was used when this access plan was created. • AB - Storage pool changed or DEGREE parameter of CHGQRYA command changed. • AC - The system feature DB2 Multisystem has been installed or removed. • AD - The value of the degree query attribute has changed. • AE - A view is either being opened by a high level language or a view is being materialized. • AF - A user-defined type or user-defined function is not the same object as the one referred to in the access plan, or, the SQL Path is not the same as when the access plan was built. • B0 - The options specified have changed as a result of the query options file. • B1 - The access plan was generated with a commitment control level that is different in the current job. • B2 - The access plan was generated with a static cursor answer set size that is different than the previous access plan. • 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 Materialized query tables have changed.
Data_Conversion_Reason_Code	QQC16	Data conversion • N - No. • 0 - Not applicable. • 1 - Lengths do not match. • 2 - Numeric types do not match. • 3 - C host variable is NUL-terminated. • 4 - Host variable or column is variable length and the other 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 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Data_Conversion_Reason_Code (continued)		• 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 and 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	Total time for this statement, in milliseconds. For fetches, this includes all fetches for this OPEN of the cursor.
Rows_Fetched	QQI3	Total rows fetched for cursor
End_Timestamp	QQETIM	Time SQL request completed
Total_Time_Microseconds	QQI6	Total time for this statement, in microseconds. For fetches, this includes all fetches for this OPEN of the cursor.
SQL_Statement_Length	QQI7	Length of SQL Statement
Insert_Unique_Count	QQI1	Unique query count for the QDT associated with the INSERT. QQUCNT contains the unique query count for the QDT associated with the WHERE part of the statement.
SQLCode	QQI8	SQL return code
SQLState	QQC81	SQLSTATE
Close_Cursor_Mode	QVC101	Close Cursor. Possible values are: • *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	ALWCPYDTA setting (Y/N/O) • Y - A copy of the data may be used. • N - Cannot use a copy of the data. • O - The optimizer can choose to use a copy of the data for performance.
PseudoOpen	QVC12	Pseudo Open (Y/N) for SQL operations that can trigger opens. • OP - Open • IN - Insert • UP - Update • DL - Delete • SI - Select Into • SV - Set • VI - Values into For all operations it can be blank.

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
PseudoClose	QVC13	Pseudo Close (Y/N) for SQL operations that can trigger a close. • CL - Close • IN - Insert • UP - Update • DL - Delete • SI - Select Into • SV - Set • VI - Values into For all operations it can be blank.
ODP_Implementation	QVC14	ODP implementation • R - Reusable ODP • N - Nonreusable ODP • ' ' - Column not used
Dynamic_Replan_SubCode	QVC21	Dynamic replan, subtype reason code
Commitment_Control_Level	QVC41	Commitment control level. Possible values are: • 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
Blocking_Type	QVC15	Type of blocking . Possible value are: • S - Single row, ALWBLK(*READ) • F - Force one row, ALWBLK(*NONE) • L - Limited block, ALWBLK(*ALLREAD)
Delay_Prepare	QVC16	Delay prepare (Y/N)
Explainable	QVC1C	The SQL statement is explainable (Y/N).
Naming_Convention	QVC17	Naming convention. Possibles values: • N - System naming convention • S - SQL naming convention
Dynamic_Processing_Type	QVC18	Type of dynamic processing. • E - Extended dynamic • S - System wide cache • L - Local prepared statement
LOB_Data_Optimized	QVC19	Optimize LOB data types (Y/N)

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Program_User_Profile_Used	QVC1A	User profile used when compiled programs are executed. Possible values are: • 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. • 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	QVC1000	Path used to find procedures, functions, and user defined types for static SQL statements.
SQL_Path_2	QWC1000	Continuation of SQL path, if needed. Contains bytes 1001-2000 of the SQL path.
SQL_Path_3	QVC5001	Continuation of SQL path, if needed. Contains bytes 2001-2500 of the SQL path.
SQL_Path_4	QVC5002	Continuation of SQL path, if needed. Contains bytes 2501-3000 of the SQL path.
SQL_Path_5	QVC3001	Continuation of SQL path, if needed. Contains bytes 3001-3300 of the SQL path.
SQL_Path_6	QVC3002	Continuation of SQL path, if needed. Contains bytes 3301-3600 of the SQL path.
SQL_Path_7	QVC3003	Continuation of SQL path, if needed. Contains bytes 3601-3900 of the SQL path.
Current_Schema	QVC1284	SQL Current Schema
Binding_Type	QQC18	Binding type: • C - Column-wise binding • R - Row-wise binding
Cursor_Type	QQC61	Cursor Type: • NSA - Non-scrollable, asensitive, forward only • NSI - Non-scrollable, sensitive, forward only • NSS - Non-scrollable, insensitive, forward only • SCA - scrollable, asensitive • SCI - scrollable, sensitive • SCS - scrollable, insensitive
Statement_Originator	QVC1D	SQL statement originator: • U - User • S - System

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Hard_Close_Reason_Code	QQC15	SQL cursor hardclose reason. Possible reasons are: • 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 hardclose reason subcode
Date_Format	QVC42	Date Format. Possible values are: • ISO • USA • EUR • JIS • JUL • MDY • DMY • YMD
Date_Separator	QWC11	Date Separator. Possible values are: • "/" • "." • "," • "-" • " "

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Time_Format	QVC43	Time Format. Possible values are: • ISO • USA • EUR • JIS • HMS
Time_Separator	QWC12	Time Separator. Possible values are: • ":" • "." • "," • " "
Decimal_Point	QWC13	Decimal Point. Possible values are: • "." • ","
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: • 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: – 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.

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
SQL_Access_Plan_Reason_Code (continued)		• A7 - i5/OS 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: – date format – date separator – time format – time separator. • AA - The sort sequence table specified is different than the sort sequence table that was used when this access plan was created. • AB - Storage pool changed or DEGREE parameter of CHGQRYA command changed. • AC - The system feature DB2 Multisystem has been installed or removed. • AD - The value of the degree query attribute has changed. • AE - A view is either being opened by a high level language or a view is being materialized. • AF - A user-defined type or user-defined function is not the same object as the one referred to in the access plan, or, the SQL Path is not the same as when the access plan was built. • B0 - The options specified have changed as a result of the query options file. • B1 - The access plan was generated with a commitment control level that is different in the current job. • B2 - The access plan was generated with a static cursor answer set size that is different than the previous access plan. • 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 Materialized query tables have changed.

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Access_Plan_Not_Saved_Reason_Code	QVC24	Access plan not saved reason code. Possible reasons are: • A1 - Failed to get a 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

Table 37. QQQ1000 - SQL Information (continued)

View Column Name	Table Column Name	Description
Fast_Delete_Count	QVP151	SQL fast count delete count. Possible values are: • 0 = *OPTIMIZE or *DEFAULT • 1-999,999,999,999 = User specified value • 'FFFFFFFFFFFFFFFF'x = *NONE
Statement_Max_Compression	QQF2	SQL statement maximum compression. Possible values are: • 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_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 sub-length
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.
Full_Opens	QVP15E	Number of runs that were processed as full opens. If Null, then the refresh count (qvrcnt) should be used to determine if the open was a full open (0) or a pseudo open (>0)

Database monitor view 3000 - Table Scan

```
Create View QQQ3000 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,
        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

 FROM   UserLib/DBMONTABLE
 WHERE  QQRID=3000)
```

Table 38. 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

Table 38. QQQ3000 - Table Scan (continued)

View Column Name	Table Column Name	Description
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_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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product

Table 38. QQQ3000 - Table Scan (continued)

View Column Name	Table Column Name	Description
Index_Advised_Columns_Count	QQI2	Number of advised columns that use index scan-key positioning
DataSpace_Selection	QQDSS	Dataspace selection • Y - Yes • N - No
Index_Advised	QQIDXA	Index advised • Y - Yes • N - No
Reason_Code	QQRCOD	Reason code • 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: • I - I/O bound • C - CPU bound
Unique_Refresh_Counter	QVRCNT	Unique refresh counter
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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)

Table 38. QQQ3000 - Table Scan (continued)

View Column Name	Table Column Name	Description
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	Pool size
Pool_Id	QVP155	Pool id
MQT_Replacement	QQC13	Materialized Query Table replaced queried table (Y/N)

Database monitor view 3001 - Index Used

```
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,
        QVc14 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
  FROM  UserLib/DBMONTable
  WHERE QQRID=3001)
```

Table 39. 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

Table 39. QQQ3001 - Index Used (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
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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join

Table 39. QQQ3001 - Index Used (continued)

View Column Name	Table Column Name	Description
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Index_Advised_Probe_Count	QQI2	Number of advised key columns that use index scan-key positioning
Index_Probe_Used	QQKP	Index scan-key positioning • 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 • Y - Yes • N - No
DataSpace_Selection	QQDSS	Dataspace selection • Y - Yes • N - No
Index_Advised	QQIDXA	Index advised • Y - Yes • N - No
Reason_Code	QQRCOD	Reason code • I1 - Row selection • I2 - Ordering/Grouping • I3 - Row selection and Ordering/Grouping • I4 - Nested loop join • I5 - Row selection using bitmap processing
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: • I - I/O bound • C - CPU bound

Table 39. QQQ3001 - Index Used (continued)

View Column Name	Table Column Name	Description
Unique_Refresh_Counter	QVRCNT	Unique refresh counter
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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: • 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: • P - Primary Index • T - Tertiary (AND/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 39. QQQ3001 - Index Used (continued)

View Column Name	Table Column Name	Description
Table_Column_For_Index_Scan	QVC3004	Columns used for index scan-key selection
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	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
MQT_Replacement	QQC13	Materialized Query Table replaced queried table (Y/N)

Database monitor view 3002 - Index Created

```
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,
         QQRCOD 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
  FROM   UserLib/DBMONTable
  WHERE  QQRID=3002)
```

Table 40. 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
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

Table 40. QQQ3002 - Index Created (continued)

View Column Name	Table Column Name	Description
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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Index_Advised_Probe_Count	QQI2	Number of advised key columns that use index scan-key positioning
Index_Probe_Used	QQKP	Index scan-key positioning • 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 • Y - Yes • N - No
DataSpace_Selection	QQDSS	Dataspace selection • Y - Yes • N - No
Index_Advised	QQIDXA	Index advised • Y - Yes • N - No
Reason_Code	QQRCOD	Reason code • I1 - Row selection • I2 - Ordering/Grouping • I3 - Row selection and Ordering/Grouping • I4 - Nested loop join

Table 40. QQQ3002 - Index Created (continued)

View Column Name	Table Column Name	Description
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: • I - I/O bound • C - CPU bound
Unique_Refresh_Counter	QVRCNT	Unique refresh counter
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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.
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: • 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.

Table 40. QQQ3002 - Index Created (continued)

View Column Name	Table Column Name	Description
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: • 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)

Database monitor view 3003 - Query Sort

```
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,
        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,
        QVJFANO as Join_Fanout,
        QVFILES as Join_Table_Count
  FROM  UserLib/DBMONTable
  WHERE QQRID=3003)
```

Table 41. QQQ3003 - Query Sort

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

Table 41. QQQ3003 - Query Sort (continued)

View Column Name	Table Column Name	Description
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
Reason_Code	QQRCOD	Reason code • F1 - Query contains grouping columns (GROUP BY) from more that 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 that 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)		• 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: • 51 - Query contains UNION and ORDER BY • 52 - Query contains UNION ALL
Bound	QVBNDY	I/O or CPU bound. Possible values are: • I - I/O bound • C - CPU bound

Table 41. QQQ3003 - Query Sort (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_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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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

Database monitor view 3004 - Temp Table

```
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_Subqery_QDT_Level,
       QQI8 as Materialized_Union_QDT_Level,
       QQC14 as View_Contains_Union
 FROM  UserLib/DBMONTable
 WHERE QQRID=3004)
```

Table 42. QQQ3004 - Temp Table

View Column Name	Table Column Name	Description
Row_ID	QQRID	Row identification
Time_Created	QQTIME	Time row was created

Table 42. QQQ3004 - Temp Table (continued)

View Column Name	Table Column Name	Description
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
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 • Y - Yes • N - No
Table_Rows	QQTMPR	Estimated number of rows in the temporary

Table 42. QQQ3004 - Temp Table (continued)

View Column Name	Table Column Name	Description
Reason_Code	QQRCOD	Reason code. Possible values are: • F1 - Query contains grouping columns (GROUP BY) from more that 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 that 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_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

Table 42. QQQ3004 - Temp Table (continued)

View Column Name	Table Column Name	Description
Temporary_Table_Schema	QQC102	Temporary table schema
Bound	QVBNDY	I/O or CPU bound. Possible values are: • I - I/O bound • C - CPU bound
Unique_Refresh_Counter	QVRCNT	Unique refresh counter
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Temporary_Table_Row_Size	QQI2	Row size of temporary table, in bytes
Temporary_Table_Size	QQI3	Estimated size of temporary table, in bytes.

Table 42. QQQ3004 - Temp Table (continued)

View Column Name	Table Column Name	Description
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_Subqery_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)

Database monitor view 3005 - Table Locked

```
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/DBMONTable
     WHERE  QQRID=3005)
```

Table 43. 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)
Unlock_Request	QQC12	Unlock request (Y/N)
Reason_Code	QQRCOD	Reason code • 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 43. QQQ3005 - Table Locked (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	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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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

```
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,
 FROM   UserLib/DBMONTable
 WHERE  QQRID=3006)
```

Table 44. 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
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 44. QQQ3006 - Access Plan Rebuilt (continued)

View Column Name	Table Column Name	Description
Reason_Code	QQRCOD	Reason code why access plan was rebuilt • 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: – 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 - i5/OS 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: – date format – date separator – time format – time separator.

Table 44. QQQ3006 - Access Plan Rebuilt (continued)

View Column Name	Table Column Name	Description
Reason_Code (continued)	QQRCOD	• 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.
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. • 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

Database monitor view 3007 - Optimizer Timed Out

```
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
 FROM   UserLib/DBMONTable
 WHERE  QQRID=3007)
```

Table 45. 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
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

Table 45. QQQ3007 - Optimizer Timed Out (continued)

View Column Name	Table Column Name	Description
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
Index_Names	QQ1000	Names of indexes not used and reason code. 1. Access path was not in a valid state. The system invalidated the access path. 2. Access path was not in a valid state. The user requested that the access path be rebuilt. 3. Access path is a temporary access path (resides in library QTEMP) and was not specified as the file to be queried. 4. The cost to use this access path, as determined by the optimizer, was higher than the cost associated with the chosen access method. 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. 6. The keys of the access path did not match the fields specified for the join criteria. 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. 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. 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. 10. The user specified ignore decimal data errors on the query. This disallows the use of permanent access paths.

Table 45. QQQ3007 - Optimizer Timed Out (continued)

View Column Name	Table Column Name	Description
Index_Names (continued)	QQ1000	• 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 45. QQQ3007 - Optimizer Timed Out (continued)

View Column Name	Table Column Name	Description
Join_Method	QQC21	Join method - when available • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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 3008 - Subquery Processing

```
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
 FROM   UserLib/DBMONTable
 WHERE  QQRID=3008)
```

Table 46. 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	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
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

Database monitor view 3010 - HostVar & ODP Implementation

```
Create View QQQ3010 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 Unqiue_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
 FROM   UserLib/DBMONTable
 WHERE  QQRID=3010)
```

Table 47. 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)
Unqiue_Refresh_Counter2	QQI5	Unique refresh counter
User_Defined	QQUDEF	User defined column
ODP_Implementation	QQC11	ODP implementation • R - Reusable ODP • N - Nonreusable ODP • ' ' - Column not used
Host_Variable_Implementation	QQC12	Host variable implementation • I - Interface supplied values (ISV) • V - Host variables treated as constants (V2) • U - Table management row positioning (UP)
Host_Variable_Values	QQ1000	Host variable values
Unique_Refresh_Counter	QVRCNT	Unique refresh counter

Database monitor view 3014 - Generic QQ Information

```
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
FROM   UserLib/DBMONTable
WHERE  QQRID=3014)
```

Table 48. 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
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
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 • 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)

Table 48. QQQ3014 - Generic QQ Information (continued)

View Column Name	Table Column Name	Description
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
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: • I - Index • S - Sort
Grouping_Implementation	QVC12	Grouping implementation. Possible values are: • I - Index • H - Hash grouping
Join_Implementation	QVC13	Join Implementation. Possible values are: • 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

Table 48. QQQ3014 - Generic QQ Information (continued)

View Column Name	Table Column Name	Description
Parallel_Degree	QVC81	Parallel Degree • *SAME - Don't change current setting • *NONE - No parallel processing is allowed • *I/O - Any number of tasks may be used for I/O processing. SMP parallel processing is not allowed. • *OPTIMIZE - The optimizer chooses the number of tasks to use for either I/O or SMP parallel processing. • *MAX - The optimizer chooses to use either I/O or SMP parallel processing. • *SYSVAL - Use the current system value to process the query. • *ANY - Has the same meaning as *I/O. • *NBRTASKS - The number of tasks for SMP parallel processing is specified in column QVTASKN.
Max_Number_of_Tasks	QQI3	Max number of tasks
Apply_CHGQRYA_Remote	QVC17	Apply CHGQRYA remotely (Y/N)
Async_Job_Usage	QVC82	Asynchronous job usage • *SAME - Don't change current setting • *DIST - Asynchronous jobs may be used for queries with distributed tables • *LOCAL - Asynchronous jobs may be used for queries with local tables only • *ANY - Asynchronous jobs may be used for any database query • *NONE - No asynchronous jobs are allowed
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: • *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: • O - Only reoptimize the access plan when absolutely required. Do not reoptimize for subjective reasons. • Y - Yes, force the access plan to be reoptimized. • N - No, do not reoptimize the access plan, unless optimizer determines that it is necessary. May reoptimize for subjective reasons.

Table 48. QQQ3014 - Generic QQ Information (continued)

View Column Name	Table Column Name	Description
Optimize_All_Indexes		Optimize all indexes requested • *SAME - Don't 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	This is 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. • *NO - Star join optimization will not be performed. • *COST - The optimizer will determine if any EVIs can be used for star join optimization. • *FORCE - The optimizer will add any EVIs that can be used for star join optimization.
Optimization_Goal	QVC23	Byte 1 = Optimization goal. Possible values are: • 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.
VE_Diagram_Type	QVC24	Byte 1 = Type of Visual Explain diagram. Possible values are: • D - Detail • B - Basic
Ignore_Like_Redunant_Shifts	QVC24	Byte 2 - Ignore LIKE redundant shifts. Possible values are: • O - Optimize, the query optimizer determines which redundant shifts to ignore. • A - All, all redundant shifts will be 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: • *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.

Table 48. QQQ3014 - Generic QQ Information (continued)

View Column Name	Table Column Name	Description
No_Parameter_Marker_Reason_Code	QQI6	Reason code for why Parameter Marker Conversion was not performed: 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.
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 will be 99999999999999.
MQT_Usage	QVC42,1,1	Byte 1 - Contains the MATERIALIZED_QUERY_TABLE_USAGE. Possible values are: • 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: • XL - Translation used in query • XU - Translation for UTF used in query • UF - User Defined Table Function used in query • LF - DDS logical file specified in query definition • LC - Lateral correlation • 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

Table 48. QQQ3014 - Generic QQ Information (continued)

View Column Name	Table Column Name	Description
SQE_NotUsed_Reason_Code (continued)		• 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)

Database monitor view 3015 - Statistics Information

```
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
   FROM   UserLib/DBMONTable
   WHERE  QQRID=3015)
```

Table 49. 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
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: • 'N' - No statistic • 'S' - Stale statistic • ' ' - Unknown
Statistic_Importance	QQI2	Importance of this statistic

Table 49. QQQ3015 - Statistic Information (continued)

View Column Name	Table Column Name	Description
Statistic_Columns	QQ1000	Columns for the statistic advised
Statistic_ID	QVC1000	Statistic identifier

Database monitor view 3018 - STRDBMON/ENDDBMON

```
Create View QQQ3018 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,
          QQC11 as Monitored_Job_type,
          QQC12 as Monitor_Command,
          QQC301 as Monitor_Job_Information,
          QQ1000L as STRDBMON_Command_Text
   FROM   UserLib/DBMONTable
   WHERE  QQRID=3018)
```

Table 50. 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 • C - Current • J - Job name • A - All
Monitor_Command	QQC12	Command type • S - STRDBMON • E - ENDDBMON
Monitor_Job_Information	QQC301	Monitored job information • * - Current job • Job number/User/Job name • *ALL - All jobs
STRDBMON_Command_Text	QQ1000L	STRDBMON command text.

Database monitor view 3019 - Rows retrieved

```
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
   FROM   UserLib/DBMONTable
   WHERE  QQRID=3019)
```

Table 51. 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 51. 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.

Database monitor view 3020 - Index advised (SQE)

```
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,
          QQIDXD 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
  FROM   UserLib/DBMONTable
  WHERE  QQRID=3020)
```

Table 52. 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
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)

Table 52. QQQ3020 - Index advised (SQE) (continued)

View Column Name	Table Column Name	Description
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 • 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: • B - Radix index • E - Encoded vector index
NLSS_Table	QQNTNM	Sort Sequence Table
NLSS_Library	QQNLNM	Sort Sequence Library

Related reference

"Query optimizer index advisor" on page 110
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

```
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,
          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,
        QVP151 as Bitmap_Count,
        QVC3001 as Bitmap_IDs
 FROM   UserLib/DBMONTable
 WHERE  QQRID=3021)
```

Table 53. 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
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

Table 53. QQQ3021 - Bitmap Created (continued)

View Column Name	Table Column Name	Description
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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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

Database monitor view 3022 - Bitmap Merge

```
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
 FROM   UserLib/DBMONTable
 WHERE  QQRID=3022)
```

Table 54. 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

Table 54. QQQ3022 - Bitmap Merge (continued)

View Column Name	Table Column Name	Description
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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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

Database monitor view 3023 - Temp Hash Table Created

```
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
   FROM   UserLib/DBMONTable
   WHERE  QQRID=3023)
```

Table 55. 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	QQJFLD	Join column (unique per job)
Relational_Database_Name	QQRDBN	Relational database name
System_Name	QQSYS	System name
Job_Name	QQJOB	Job name

Table 55. QQQ3023 - Temp Hash Table Created (continued)

View Column Name	Table Column Name	Description
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 • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join

Table 55. QQQ3023 - Temp Hash Table Created (continued)

View Column Name	Table Column Name	Description
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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 • 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

Database monitor view 3025 - Distinct Processing

```
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
  FROM  UserLib/DBMONTable
  WHERE QQRID=3025)
```

Table 56. 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)
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

Table 56. QQQ3025 - Distinct Processing (continued)

View Column Name	Table Column Name	Description
Estimated_Cumulative_Time	QVCTIM	Estimated cumulative time, in seconds
Estimated_Rows_Selected	QQREST	Estimated rows selected

Database monitor view 3026 - Set operation

```
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
   FROM   UserLib/DBMONTable
   WHERE  QQRID=3026)
```

Table 57. QQQ3026 - Set operatoin

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 57. QQQ3026 - Set operatoin (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 (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: • 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)
Set_in_a_View	QQC23	Set operation within a view. • 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.

Table 57. QQQ3026 - Set operatoin (continued)

View Column Name	Table Column Name	Description
Set_Operator	QQC22	Type of set operation. Possible values are: • UU - Union • UA - Union All • UR - Union Recursive • EE - Except • EA - Except All • II - Intersect • IA - Intersect All

Database monitor view 3027 - Subquery Merge

```
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
 FROM   UserLib/DBMONTable
 WHERE  QQRID=3027)
```

Table 58. 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
Estimated_Join_Rows	QQAJN	Estimated number of joined rows
Join_Position	QQJNP	Join position - when available
DataSpace_Number	QQI6	Dataspace number

Table 58. QQQ3027 - Subquery Merge (continued)

View Column Name	Table Column Name	Description
Join_Method	QQC21	Join method - when available • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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
Subquery_Operator	QQC101	Subquery operator. Possible values are: • 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

Table 58. QQQ3027 - Subquery Merge (continued)

View Column Name	Table Column Name	Description
Subquery_Type	QVC21	Subquery type. Possible values are: • 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

Database monitor view 3028 - Grouping

```
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
FROM    UserLib/DBMONTable
WHERE   QQRID=3028)
```

Table 59. 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
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

Table 59. QQQ3028 - Grouping (continued)

View Column Name	Table Column Name	Description
Join_Method	QQC21	Join method - when available • NL - Nested loop • MF - Nested loop with selection • HJ - Hash join
Join_Type	QQC22	Join type - when available • IN - Inner join • PO - Left partial outer join • EX - Exception join
Join_Operator	QQC23	Join operator - when available • EQ - Equal • NE - Not equal • GT - Greater than • GE - Greater than or equal • LT - Less than • LE - Less than or equal • CP - Cartesian product
Join_Fanout	QVJFANO	Join fan out. Possible values are: • 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 • ' ' - 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
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

Table 59. QQQ3028 - Grouping (continued)

View Column Name	Table Column Name	Description
STDDEV_Columns	QVC3007	STDDEV columns
VAR_Columns	QVC3008	VAR columns

Database monitor view 3030 - Materialized query tables

```
Create View QQQ3030 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,
          QQ1000 as Materialized_Query_Tables,
          QQC301 as MQT_Reason_Codes
   FROM   UserLib/DBMONTable
   WHERE  QQRID=3030)
```

Table 60. QQQ3030 - Materialized query tables

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

Table 60. QQQ3030 - Materialized query tables (continued)

View Column Name	Table Column Name	Description
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: • 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 60. QQQ3030 - Materialized query tables (continued)

View Column Name	Table Column Name	Description
Materialized_Query_Tables (continued)		• 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.
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)
QVRCNT	QVRCNT	Unique refresh counter

Database monitor view 3031 - Recursive common table expressions

```
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
 FROM   UserLib/DBMONTable
 WHERE  QQRID=3031)
```

Table 61. 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
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)

Table 61. QQQ3031 - Recursive common table expressions (continued)

View Column Name	Table Column Name	Description
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: • Y - checking for cyclic data • N - not checking for cyclic data
Recursive_Query_Search_Option	QQC15	SEARCH option: • 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.

Memory Resident Database Monitor: DDS

The following DDS statements are used to create the memory resident database monitor physical and logical files.

External table description (QAQQQRYI) - Summary Row for SQL Information

Table 62. QAQQQRYI - Summary Row for SQL Information

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQJOB	Job name
QQUSER	Job user
QQJNUM	Job number
QQTHID	Thread Id
QQUDEF	User defined column
QQPLIB	Name of the library containing the program or package
QQCNAM	Cursor name
QQPNAM	Name of the package or name of the program that contains the current SQL statement
QQSNAM	Name of the statement for SQL statement, if applicable
QQCNT	Statement usage count
QQAVGT	Average runtime (ms)
QQMINT	Minimum runtime (ms)
QQMAXT	Maximum runtime (ms)

Table 62. QAQQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQOPNT	Open time for most expensive execution (ms)
QQFETT	Fetch time for most expensive execution (ms)
QQCLST	Close time for most expensive execution (ms)
QQOTHT	Other time for most expensive execution (ms)
QQLTU	Time statement last used
QQMETU	Most expensive time used
QQAPRT	Access plan rebuild time
QQFULO	Number of full opens
QQPSUO	Number of pseudo-opens
QQTOTR	Total rows in table if non-join
QQRROW	Number of result rows returned
QQRROW	Statement function S - Select U - Update I - Insert D - Delete L - Data definition language O - Other
QQSTOP	Statement operation • AD - Allocate descriptor • AL - Alter table • AP - Alter procedure • AQ - Alter sequence • CA - Call • CC - Create collection • CD - Create type • CF - Create function • CG - Create trigger • CI - Create index • CL - Close • CM - Commit • CN - Connect • CO - Comment on • CP - Create procedure • CQ - Create sequence • CS - Create alias/synonym • CT - Create table • CV - Create view • DA - Deallocate descriptor • DE - Describe • DI - Disconnect • DL - Delete

Table 62. QAQQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQSTOP (continued)	• DM - Describe parameter marker • DP - Declare procedure • DR - Drop • 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 • JR - Server job reused • LK - Lock • LO - Label on • MT - More text (Depreciated in V5R4) • OP - Open • PD - Prepare and describe • PR - Prepare • 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 • UP - Update • VI - Values into • X0 - Unknown statement

Table 62. QAQQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQSTOP (continued)	• X1 - Unknown statement • X2 - DRDA (AS) Unknown statement • X3 - Unknown statement • X9 - Internal error • XA - X/Open API • ZD - Host server only activity
QQODPI	ODP implementation R - Reusable ODP (ISV) N - Non-reusable ODP (V2)
QQHVI	Host variable implementation I - Interface supplied values (ISV) V - Host variables treated as constants (V2) U - Table management row positioning (UP)
QQAPR	Access plan rebuilt • **A1** - A table or member is not the same object as the one referenced when the access plan was last built. Some reasons they may be different are: – 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** - i5/OS 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: – date format – date separator – time format – time separator

Table 62. QAQQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQAPR (continued)	• **AA** - The sort sequence table specified is different than the sort sequence table that was used when this access plan was created. • **AB** - Storage pool changed or DEGREE parameter of CHGQRYA command changed. • **AC** - The system feature DB2 multisystem has been installed or removed. • **AD** - The value of the degree query attribute has changed. • **AE** - A view is either being opened by a high level language or a view is being materialized. • **AF** - A sequence object or user-defined type or function is not the same object as the one referred to in the access plan; or, the SQL path used to generate the access plan is different than the current SQL path. • **B0** - The options specified have changed as a result of the query options file QAQQINI. • **B1** - The access plan was generated with a commitment control level that is different in the current job. • **B2** - The access plan was generated with a static cursor answer set size that is different than the previous access plan. • **B3** - The query was reoptimized because this is the first run of the query after a prepare. That is, it is the first run with real actual parameter marker values. • **B4** - The query was reoptimized because referential or check constraints have changed. • **B5** - The query was reoptimized because Materialized Query Tables have changed.
QQDACV	Data conversion • N - No. • 0 - Not applicable. • 1 - Lengths do not match. • 2 - Numeric types do not match. • 3 - C host variable is NUL-terminated. • 4 - Host variable or column is variable length and the other 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. • 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 and 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.
QQCTS	Statement table scan usage count
QQCIU	Statement index usage count
QQCIC	Statement index creation count
QQCSO	Statement sort usage count
QQCTF	Statement temporary table count
QQCIA	Statement index advised count

Table 62. QAQQQRYI - Summary Row for SQL Information (continued)

Column Name	Description
QQCAPR	Statement access plan rebuild count
QQARSS	Average result set size
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQC1000	Reserved

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

Table 63. QAQQTEXT - Summary Row for SQL Statement

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together with row identification
QQTIME	Time row was created
QQSTTX	Statement text
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3000) - Arrival sequence

Table 64. QAQQ3000 - Arrival sequence

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together with row identification
QQTIME	Time row was created
QQQDTN	QDT number (unique per ODT)
QQQDTL	QDT subquery nested level
QQMATN	Materialized view QDT number

Table 64. QAQQ3000 - Arrival sequence (continued)

Column Name	Description
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQPTLN	Physical library
QQPTFN	Physical table
QQTOTR	Total rows in table
QQREST	Estimated number of rows selected
QQAJN	Estimated number of joined rows
QQEPT	Estimated processing time, in seconds
QQJNP	Join position - when available
QQJNDS	Dataspace number
QQJNMT	Join method - when available NL - Nested loop MF - Nested loop with selection HJ - Hash join
QQJNTY	Join type - when available IN - Inner join PO - Left partial outer join EX - Exception join
QQJNOP	Join operator - when available EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product
QQDSS	Dataspace selection Y - Yes N - No
QQIDXA	Index advised Y - Yes N - No
QQRCOD	Reason code T1 - No indexes exist. T2 - Indexes exist, but none can be used. T3 - Optimizer chose table scan over available indexes.
QQLTLN	Library-long
QQLTFN	Table-long
QQLPTL	Physical library-long
QQLPTF	Table-long

Table 64. QAQQ3000 - Arrival sequence (continued)

Column Name	Description
QQIDXD	Key columns for the index advised
QQC11	Materialized query table
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Number of advised key columns that use index scan-key positioning.
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3001) - Using existing index

Table 65. QQQ3001 - Using existing index

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQPTLN	Physical library
QQPTFN	Physical table
QQILNM	Index library
QQIFNM	Index
QQTOTR	Total rows in table
QQREST	Estimated number of rows selected
QQFKEY	Number of key positioning keys
QQKSEL	Number of key selection keys
QQAJN	Join position - when available
QQEPT	Estimated processing time, in seconds
QQJNP	Join position - when available
QQJNDS	Dataspace number
QQJNMT	Join method - when available NL - Nested loop MF - Nested loop with selection HJ - Hash join

Table 65. QQQ3001 - Using existing index (continued)

Column Name	Description
QQJNTY	Join type - when available IN - Inner join PO - Left partial outer join EX - Exception join
QQJNOP	Join operator - when available EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product
QQIDXK	Number of advised key columns that use index scan-key positioning
QQKP	Index scan-key positioning Y - Yes N - No
QQKPN	Number of key positioning columns
QQKS	Index scan-key selection Y - Yes N - No
QQDSS	Dataspace selection Y - Yes N - No
QQIDXA	Index advised Y - Yes N - No
QQRCOD	Reason code I1 - Row selection I2 - Ordering/Grouping I3 - Row selection and Ordering/Grouping I4 - Nested loop join I5 - Row selection using bitmap processing
QQCST	Constraint indicator Y - Yes N - No
QQCSTN	Constraint name
QQLTLN	Library-long
QQLTFN	Table-long
QQLPTL	Physical library-long
QQLPTF	Table-long

Table 65. QQQ3001 - Using existing index (continued)

Column Name	Description
QQLILN	Index library – long
QQLIFN	Index – long
QQIDXD	Key columns for the index advised
QQC11	Materialized query table
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3002) - Index created

Table 66. QQQ3002 - Index created

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQPTLN	Physical library
QQPTFN	Physical table
QQILNM	Index library
QQIFNM	Index
QQNTNM	NLSS table
QQNLNM	NLSS library
QQTOTR	Total rows in table
QQRIDX	Number of entries in index created
QQREST	Estimated number of rows selected
QQFKEY	Number of index scan-key positioning keys
QQKSEL	Number of index scan-key selection keys
QQAJN	Estimated number of joined rows
QQJNP	Join position - when available
QQJNDS	Dataspace number

Table 66. QQQ3002 - Index created (continued)

Column Name	Description
QQJNMT	Join method - when available NL - Nested loop MF - Nested loop with selection HJ - Hash join
QQJNTY	Join type - when available IN - Inner join PO - Left partial outer join EX - Exception join
QQJNOP	Join operator - when available EQ - Equal NE - Not equal GT - Greater than GE - Greater than or equal LT - Less than LE - Less than or equal CP - Cartesian product
QQIDXK	Number of advised key columns that use index scan-key positioning
QQEPT	Estimated processing time, in seconds
QQKP	Index scan-key positioning Y - Yes N - No
QQKPN	Number of index scan-key positioning columns
QQKS	Index scan-key selection Y - Yes N - No
QQDSS	Dataspace selection Y - Yes N - No
QQIDXA	Index advised Y - Yes N - No
QQCST	Constraint indicator Y - Yes N - No
QQCSTN	Constraint name

Table 66. QQQ3002 - Index created (continued)

Column Name	Description
QQRCOD	Reason code I1 - Row selection I2 - Ordering/Grouping I3 - Row selection and Ordering/Grouping I4 - Nested loop join I5 - Row selection using bitmap processing
QQTTIM	Index create time
QQLTLN	Library-long
QQLTFN	Table-long
QQLPTL	Physical library-long
QQLPTF	Table-long
QQLILN	Index library-long
QQLIFN	Index-long
QQLNTN	NLSS table-long
QQLNLN	NLSS library-long
QQIDXD	Key columns for the index advised
QQCRTK	Key columns for index created
QQC11	Materialized query table
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3003) - Query sort

Table 67. QQQ3003 - Query sort

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTTIM	Sort time
QQRSS	Number of rows selected or sorted
QQSIZ	Size of sort space

Table 67. QQQ3003 - Query sort (continued)

Column Name	Description
QQPSIZ	Pool size
QQPID	Pool id
QQIBUF	Internal sort buffer length
QQEBUF	External sort buffer length
QQRCOD	Reason code **F1** - Query contains grouping columns (Group By) from more than one table, or contains grouping columns from a secondary table of a join query that cannot be reordered. **F2** - Query contains ordering columns (Order By) from more than one table, or contains ordering columns from a secondary table of a join query that cannot be reordered. **F3** - The grouping and ordering columns are not compatible. **F4** - DISTINCT was specified for the query. **F5** - UNION was specified for the query. **F6** - Query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 columns specified for ordering. **F7** - Query optimizer chose to use a sort rather than an index to order the results of the query. **F8** - Perform specified row selection to minimize I/O wait time. **FC** - The query contains grouping fields and there is a read trigger on at least one of the physical files in the query.
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3004) - Temporary table

Table 68. QQQ3004 - Temporary table

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQTTIM	Temporary table create time
QQTMPR	Number of rows in temporary

Table 68. QQQ3004 - Temporary table (continued)

Column Name	Description
QQRCOD	Reason code **F1** - Query contains grouping columns (Group By) from more than one table, or contains grouping columns from a secondary table of a join query that cannot be reordered. **F2** - Query contains ordering columns (Order By) from more than one table, or contains ordering columns from a secondary table of a join query that cannot be reordered. **F3** - The grouping and ordering columns are not compatible. **F4** - DISTINCT was specified for the query. **F5** - UNION was specified for the query. **F6** - Query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 columns specified for ordering. **F7** - Query optimizer chose to use a sort rather than an index to order the results of the query. **F8** - Perform specified row selection to minimize I/O wait time. **F9** - The query optimizer chose to use a hashing algorithm rather than an access path to perform the grouping for the query. **FA** - The query contains a join condition that requires a temporary file. **FB** - The query optimizer creates a run-time temporary file in order to implement certain correlated group by queries. **FC** - The query contains grouping fields and there is a read trigger on at least one of the physical files in the query. **FD** - The query optimizer creates a runtime temporary file for a static-cursor request. **H1** - Table is a join logical file and its join type does not match the join type specified in the query. **H2** - Format specified for the logical table references more than one base table. **H3** - Table is a complex SQL view requiring a temporary results of the SQL view. **H4** - For an update-capable query, a subselect references a column in this table which matches one of the columns being updated. **H5** - For an update-capable query, a subselect references an SQL view which is based on the table being updated. **H6** - For a delete-capable query, a subselect references either the table from which rows are to be deleted, an SQL view, or an index based on the table from which rows are to be deleted. **H7** - A user-defined table function was materialized.
QQDFVL	Default values may be present in temporary Y - Yes N - No
QQLTLN	Library-long
QQLTFN	Table-long
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved

Table 68. QQQ3004 - Temporary table (continued)

Column Name	Description
QQ1000	Reserved

External table description (QAQQ3007) - Optimizer information

Table 69. QQQ3007 - Optimizer information

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQTLN	Library
QQTFN	Table
QQPTLN	Physical library
QQPTFN	Table
QQTOUT	Optimizer timed out Y - Yes N - No.
QQIRSN	Reason code
QQLTLN	Library-long
QQLTFN	Table-long
QQPTL	Physical library-long
QQPTF	Table-long
QQIDXN	Index names
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3008) - Subquery processing

Table 70. QQQ3008 - Subquery processing

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together

Table 70. QQQ3008 - Subquery processing (continued)

Column Name	Description
QQTIME	Time row was created
QQQDTN	QDT number (unique per QDT)
QQQDTL	RQDT subquery nested level relational database name
QQMATN	Materialized view QDT number
QQMATL	Materialized view nested level
QQORGQ	Materialized view QDT number
QQMRGQ	Materialized view nested level
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved
QQ1000	Reserved

External table description (QAQQ3010) - Host variable and ODP implementation

Table 71. QQQ3010 - Host variable and ODP implementation

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created
QQHVAR	Host variable values
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQI1	Reserved
QQI2	Reserved
QQC301	Reserved
QQC302	Reserved

External table description (QAQQ3030) - Materialized query table implementation

Table 72. QQQ3030 - Materialized query table implementation

Column Name	Description
QQKEY	Join column (unique per query) used to link rows for a single query together
QQTIME	Time row was created

Table 72. QQQ3030 - Materialized query table implementation (continued)

Column Name	Description
QQQDTN	Unique subselect number
QQQDTL	Subselect nested level
QQMATN	Materialized view subselect number
QQMATL	Materialized view nested level
QQMRSN	List of unique reason codes used by the materialized query table (each materialized query table has a corresponding reason code associated with it)
QQMQTS	List of MQTs examined with a reason code indicating if the MQT was used and if not used, why not used.
QQC11	Reserved
QQC12	Reserved
QQC21	Reserved
QQC22	Reserved
QQCI1	Reserved
QQCI2	Reserved
QQC301	Reserved
QQC302	Reserved

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 given SQL statements in a program using informational messages put in the job log by the database manager.

The messages are issued for an SQL program or interactive SQL when running in the debug mode. The database manager may send any of the following messages when appropriate. The ampersand variables (&1, &X) are replacement variables that contain either an object name or some other substitution value when the message appears in the job log. These messages provide feedback on how a query was run and, in some cases, indicate the improvements that can be made to help the query run faster.

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

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

CPI4321 - Access path built for &18 &19

CPI4321	
Message Text:	Access path built for &18 &19.

CPI4321	
Cause Text:	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: 1. Perform specified ordering/grouping criteria. 2. Perform specified join criteria. 3. Perform specified record selection to minimize I/O wait time. The access path was built using the following key fields. The key fields and their corresponding sequence (ASCEND or DESCEND) will be shown:&17. A key field of *MAP indicates the key field is an expression (derived field). The access path was built using sequence table &13 in library &14. 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. 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. A file name starting with *QUERY or *N indicates the access path was built over a temporary file.
Recovery Text:	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. 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.

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

The time required to create an index on each run of a query can be significant. Consider creating a logical file (CRTLF) or an SQL index (CREATE INDEX SQL statement):
- Over the table named in the message help.
- With key columns named in the message help.
- With the ascending or descending sequencing specified in the message help.
- With the sort sequence table specified in the message help.

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

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

CPI4322 - Access path built from keyed file &1

CPI4322	
Message Text:	Access path built from keyed file &1.
Cause Text:	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: 1. Perform specified ordering/grouping criteria. 2. Perform specified join criteria. 3. Perform specified record selection to minimize I/O wait time The access path was built using the following key fields. The key fields and their corresponding sequence (ASCEND or DESCEND) will be shown: &17. A key field of *MAP indicates the key field is an expression (derived field). The temporary access path was built using sequence table &13 in library &14. 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. 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.
Recovery Text:	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. 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. 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.

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

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

CPI4323 - The query access plan has been rebuilt

CPI4323	
Message Text:	The query access plan has been rebuilt.

<table>
<tr><th colspan="2">CPI4323</th></tr>
<tr><td>Cause Text:</td><td>The access plan was rebuilt for reason code &13. The reason codes and their meanings follow:
1. A file or member is not the same object as the one referred to in the access plan. Some reasons include the object being re-created, restored, or overridden to a new object.
2. Access plan was using a reusable Open Data Path (ODP), and the optimizer chose to use a non-reusable ODP.
3. Access plan was using a non-reusable Open Data Path (ODP) and the optimizer chose to use a reusable ODP
4. The number of records in member &3 of file &1 in library &2 has changed by more than 10%.
5. A new access path exists over member &6 of file &4 in library &5.
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.
7. The query access plan had to be rebuilt because of system programming changes.
8. The CCSID (Coded Character Set Identifier) of the current job is different than the CCSID used in the access plan
9. The value of one of the following is different in the current job: date format, date separator, time format, or time separator.
10. The sort sequence table specified has changed.
11. The number of active processors or the size or paging option of the storage pool has changed.
12. The system feature DB2 UDB Symmetric Multiprocessing has either been installed or removed.
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.
14. A view is either being opened by a high level language open, or is being materialized.
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.
16. Query attributes have been specified from the query options file &15 in library &16.
17. The access plan was generated with a commitment control level that is different in the current job.
18. The access plan was generated with a different static cursor answer set size.
19. This is the first run of the query since a prepare or compile.
20. or greater. View the second level message text of the next message issued (CPI4351) for an explanation of these reason codes.

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.</td></tr>
<tr><td>Recovery Text:</td><td>Excessive rebuilds should be avoided and may indicate an application design problem.</td></tr>
</table>

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

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

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

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

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

Related reference

"CPI4349 - Fast past refresh of the host variables values is not possible" on page 297

"CPI434C - The query access plan was not rebuilt" on page 298

CPI4324 - Temporary file built for file &1

CPI4324	
Message Text:	Temporary file built for file &1.
Cause Text:	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: 1. The file is a join logical file and its join-type (JDFTVAL) does not match the join-type specified in the query. 2. The format specified for the logical file references more than one physical file. 3. The file is a complex SQL view requiring a temporary file to contain the results of the SQL view. 4. For an update-capable query, a subselect references a field in this file which matches one of the fields being updated. 5. For an update-capable query, a subselect references SQL view &1, which is based on the file being updated. 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. 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. 8. The file is a partition file requiring a temporary file for processing the grouping or join.
Recovery Text:	You may want to change the query to refer to a file that does not require a temporary file to be built.

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

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

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

CPI4325 - Temporary result file built for query

CPI4325	
Message Text:	Temporary result file built for query.
Cause Text:	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: 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. 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. 3. The grouping and ordering fields are not compatible. 4. DISTINCT was specified for the query. 5. Set operator (UNION, EXCEPT, or INTERSECT) was specified for the query. 6. The query had to be implemented using a sort. Key length of more than 2000 bytes or more than 120 key fields specified for ordering. 7. The query optimizer chose to use a sort rather than an access path to order the results of the query. 8. Perform specified record selection to minimize I/O wait time. 9. The query optimizer chose to use a hashing algorithm rather than an access path to perform the grouping for the query. 10. The query contains a join condition that requires a temporary file. 11. The query optimizer creates a run-time temporary file in order to implement certain correlated group by queries. 12. The query contains grouping fields (GROUP BY, MIN/MAX, COUNT, and so on) and there is a read trigger on one or more of the underlying physical files in the query. 13. The query involves a static cursor or the SQL FETCH FIRST clause.
Recovery Text:	For more information about why a temporary result was used, refer to the "Data access on DB2 UDB for iSeries: data access paths and methods" on page 8.

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

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

CPI4326 - &12 &13 processed in join position &10

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

CPI4326	
Cause Text:	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: 1. Perform specified record selection. 2. Perform specified ordering/grouping criteria. 3. Record selection and ordering/grouping criteria. 4. Perform specified join criteria. 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. A file name starting with *TEMPX for the access path indicates it is a temporary access path built over file &6. A file name starting with *N or *QUERY for the file indicates it is a temporary file. Index only access was used for this file within the query: &11. 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. 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.
Recovery Text:	Generally, to force a file to be processed in join position 1, specify an order by field from that file only. 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. 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. Refer to “Data access on DB2 UDB for iSeries: data access paths and methods” on page 8 for additional tips on optimizing a query’s join order and index only access.

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

CPI4327 - File &12 &13 processed in join position &10

CPI4327	
Message Text:	&12 &13 processed in join position &10.
Cause Text:	Arrival sequence access was used to select records from member &2 of file &13 in library &1. 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. A file name that starts with *QUERY for the file indicates it is a temporary file.
Recovery Text:	Generally, to force a file to be processed in join position 1, specify an order by field from that file only.

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

CPI4328	
Message Text:	Access path of file &3 was used by query.
Cause Text:	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: 1. Record selection. 2. Ordering/grouping criteria. 3. Record selection and ordering/grouping criteria. 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. Index only access was used for this query: &11. 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. 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.
Recovery Text:	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. Refer to “Data access on DB2 UDB for iSeries: data access paths and methods” on page 8 for additional tips on index only access.

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

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

CPI4329 - Arrival sequence access was used for &12 &13

CPI4329	
Message Text:	Arrival sequence access was used for &12 &13.
Cause Text:	Arrival sequence access was used to select records from member &2 of file &13 in library &1. 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. A file name starting with *N or *QUERY for the file indicates it is a temporary file.
Recovery Text:	The use of an access path may improve the performance of the query if record selection is specified. 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. 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. For more information refer to “Data access on DB2 UDB for iSeries: data access paths and methods” on page 8.

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

CPI432A - Query optimizer timed out for file &1

<table>
<tr><th colspan="2">CPI432A</th></tr>
<tr><td>Message Text:</td><td>Query optimizer timed out for file &1.</td></tr>
<tr><td>Cause Text:</td><td>The query optimizer timed out before it could consider all access paths built over member &3 of file &1 in library &2.

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 why the access path was not used. A reason code of 0 indicates that the access path was used to implement the query.

The reason codes and their meanings follow:
1. Access path was not in a valid state. The system invalidated the access path.
2. Access path was not in a valid state. The user requested that the access path be rebuilt.
3. Access path is a temporary access path (resides in library QTEMP) and was not specified as the file to be queried.
4. The cost to use this access path, as determined by the optimizer, was higher than the cost associated with the chosen access method.
5. The keys of the access path did not match the fields specified for the ordering/grouping criteria.
6. The keys of the access path did not match the fields specified for the join criteria.
7. Use of this access path would not minimize delays when reading records from the file as the user requested.
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.
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.
10. and greater - View the second level message text of the next message issued (CPI432D) for an explanation of these reason codes.</td></tr>
<tr><td>Recovery Text:</td><td>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.

The user may want to delete any access paths no longer needed.</td></tr>
</table>

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

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

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

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

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

Related reference

"CPI432D - Additional access path reason codes were used" on page 287

CPI432B - Subselects processed as join query

CPI423B	
Message Text:	Subselects processed as join query.
Cause Text:	Two or more SQL subselects were combined together by the query optimizer and processed as a join query. Processing subselects as a join query generally results in improved performance.
Recovery Text:	None — Generally, this method of processing is a good performing option.

CPI432C - All access paths were considered for file &1

CPI432C	
Message Text:	All access paths were considered for file &1.

CPI432C	
Cause Text:	The query optimizer considered all access paths built over member &3 of file &1 in library &2. 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 why the access path was not used. A reason code of 0 indicates that the access path was used to implement the query. The reason codes and their meanings follow: 1. Access path was not in a valid state. The system invalidated the access path. 2. Access path was not in a valid state. The user requested that the access path be rebuilt. 3. Access path is a temporary access path (resides in library QTEMP) and was not specified as the file to be queried. 4. The cost to use this access path, as determined by the optimizer, was higher than the cost associated with the chosen access method. 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. 6. The keys of the access path did not match the fields specified for the join criteria. 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. 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. 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. 10. and greater - View the second level message text of the next message issued (CPI432D) for an explanation of these reason codes.
Recovery Text:	The user may want to delete any access paths no longer needed.

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

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

Related reference

"CPI432D - Additional access path reason codes were used"

CPI432D - Additional access path reason codes were used

CPI432D	
Message Text:	Additional access path reason codes were used.

CPI432D	
Cause Text:	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. The reason codes and their meanings follow: • 10 - The user specified ignore decimal data errors on the query. This disallows the use of permanent access paths. • 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 could not 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 could not 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 could not 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.
Recovery Text:	See prior message CPI432A or CPI432C for more information.

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

Related reference

"CPI432A - Query optimizer timed out for file &1" on page 285

"CPI432C - All access paths were considered for file &1" on page 286

CPI432F - Access path suggestion for file &1

CPI432F	
Message Text:	Access path suggestion for file &1.

CPI432F	
Cause Text:	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. 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. 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: &11. 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.
Recovery Text:	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. For more information, refer to “Data access on DB2 UDB for iSeries: data access paths and methods” on page 8.

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

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

<table>
<tr><th colspan="2">CPI4330</th></tr>
<tr><td>Cause Text:</td><td>&6 is the average numbers of tasks used for a &10 scan of member &3 of file &1 in library &2.

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.

A file name starting with *QUERY or *N for the file indicates a temporary result file is being used.

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:
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.

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.</td></tr>
<tr><td>Recovery Text:</td><td>To disallow usage of parallel &10 scan, specify *NONE on the query attribute degree.

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:
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.</td></tr>
</table>

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

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

CPI4331	
Cause Text:	&6 is the average numbers of tasks used for an index created over member &3 of file &1 in library &2. 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. A file name starting with *QUERY or *N for the file indicates a temporary result file is being used. 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: 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. 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.
Recovery Text:	To disallow usage of parallel index build, specify *NONE on the query attribute degree. 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: 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

CPI4332	
Message Text:	&1 host variables used in query.
Cause Text:	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. The host variables values displayed above may have been special values. An explanation of the special values follow: • 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

CPI4333	
Message Text:	Hashing algorithm used to process join.
Cause Text:	The hash join method 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. Debug messages which explain the implementation of each hash join step follow this message in the joblog. 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. 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.
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

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

CPI4335 - Optimizer debug messages for hash join step &1 follow

CPI4335	
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 on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about hashing algorithm for join processing.

CPI4336 - Group processing generated

CPI4336	
Message Text:	Group processing generated.
Cause Text:	Group processing (GROUP BY) was added to the query step. Adding the group processing reduced the number of result records which should, in turn, improve the performance of subsequent steps.
Recovery Text:	For more information refer to "Data access on DB2 UDB for iSeries: data access paths and methods" on page 8

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

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

CPI4338	
Message Text:	&1 Access path(s) used for bitmap processing of file &2.
Cause Text:	Bitmap processing was used to access records from member &4 of file &2 in library &3. 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. 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. 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. 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. The list below shows the names of the access paths used in the bitmap processing: 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.
Recovery Text:	Refer to "Data access on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about bitmap processing.

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

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

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

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

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

CPI433D - Query options used to build the i5/OS query access plan

CPI433D	
Message Text:	Query options used to build the i5/OS query access plan.
Cause Text:	The access plan that was saved was created with query options retrieved from file &2 in library &1.
Recovery Text:	None

CPI433F - Multiple join classes used to process join

CPI433F	
Message Text:	Multiple join classes used to process join.
Cause Text:	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 46 for more information about join classes.

CPI4340 - Optimizer debug messages for join class step &1 follow

CPI4340	
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 46 for more information about join classes.

CPI4341 - Performing distributed query

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

CPI4342 - Performing distributed join for query

CPI4342	
Message Text:	Performing distributed join for query.
Cause Text:	Query contains join criteria over a distributed file and a distributed join was performed, in parallel, on the following nodes: &1. The library, file and member names of each file involved in the join follow: &2. 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.
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming.

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

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

CPI4345 - Temporary distributed result file &3 built for query

CPI4345	
Message Text:	Temporary distributed result file &3 built for query.
Cause Text:	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: 1. Data from member &2 of &7 &8 in library &1 was directed to other nodes. 2. Data from member &2 of &7 &8 in library &1 was broadcast to all nodes. 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. 4. Query contains join criteria over a distributed file and the query was processed in multiple steps. A library and member name of *N indicates the data comes from a query temporary distributed file. File &3 was built on nodes: &9. It was built using partitioning keys: &10. A partitioning key of *N indicates no partitioning keys were used when building the temporary distributed result file.

CPI4345	
Recovery Text:	If the reason code is: 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. 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. 3. Better performance may be achieved if grouping fields are specified that match the partitioning keys. 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. For more information about processing of distributed files, refer to the Distributed Database Programming,

CPI4346 - Optimizer debug messages for query join step &1 of &2 follow:

CPI4346	
Message Text:	Optimizer debug messages for query join step &1 of &2 follow:
Cause Text:	Query processed in multiple steps. The optimizer debug messages that follow provide the query optimization information about join step &1 of &2 total steps.
Recovery Text:	No recovery necessary.

CPI4347 - Query being processed in multiple steps

CPI4347	
Message Text:	Query being processed in multiple steps.
Cause Text	The original query will be subdivided into multiple steps. Each step will be optimized and processed separately. Debug messages which explain the implementation of each step 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 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.
Recovery Text:	No recovery necessary.

CPI4348 - The ODP associated with the cursor was hard closed

CPI4348	
Message Text:	The ODP associated with the cursor was hard closed.

CPI4348	
Cause Text:	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: 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. 2. An additional wildcard was specified in the LIKE pattern on this invocation of the cursor. 3. SQL indicated to the query optimizer that the cursor cannot be refreshed. 4. The system code could not obtain a lock on the file being queried. 5. The length of the host variable value is too large for the host variable as determined by the query optimizer. 6. The size of the ODP to be refreshed is too large. 7. Refresh of the local ODP of a distributed query failed. 8. SQL hard closed the cursor prior to the fast path refresh code.
Recovery Text:	In order for the cursor to be used in a reusable mode, the cursor cannot be hard closed. Look at the reason why the cursor was hard closed and take the appropriate actions to prevent a hard close from occurring.

CPI4349 - Fast past refresh of the host variables values is not possible

CPI4349	
Message Text:	Fast past refresh of the host variable values is not possible.
Cause Text:	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: 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. 2. The attributes of the new host variable value are not the same as the attributes of the old host variable value. 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. 4. The host variable has a data type of IGC ONLY and the length is not even or is less than 2 bytes. 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. 6. A translate table with substitution characters was used. 7. The host variable contains DBCS data and a CCSID translate table with substitution characters is required. 8. The host variable contains DBCS that is not well formed. That is, a shift-in without a shift-out or visa versa. 9. The host variable must be translated with a sort sequence table and the sort sequence table contains substitution characters. 10. The host variable contains DBCS data and must be translated with a sort sequence table that contains substitution characters. 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.
Recovery Text:	Look at the reason why fast path refresh could not be used and take the appropriate actions so that fast path refresh can be used on the next invocation of this statement or cursor.

CPI434C - The query access plan was not rebuilt

CPI434C	
Message Text:	The query access plan was not rebuilt.
Cause Text:	The access plan for this query was not rebuilt. The optimizer determined that this access plan should be rebuilt for reason code &13. However, the query attributes in the QAQQINI file disallowed the optimizer from rebuilding this access plan at this time. For a full explanation of the reason codes and their meanings, view the second level text of the message CPI4323.
Recovery Text:	Since the query attributes disallowed the query access plan from being rebuilt, the query will continue to be implemented with the existing access plan. This access plan may not contain all of the performance benefits that may have been derived from rebuilding the access plan. For more information about query attributes refer to "Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command" on page 115

Related reference

"CPI4323 - The query access plan has been rebuilt" on page 279

CPI4350 - Materialized query tables were considered for optimization

CPI4350	
Message Text:	Materialized query tables were considered for optimization.

CPI4350	
Cause Text:	The query optimizer considered usage of materialized query tables for this query. 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. The reason codes and their meanings follow: 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 or distinct specified in the materialized query table is not compatible with the grouping or distinct specified in the query. 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 FETCH FOR FIRST n ROWS clause of the materialized query table is not compatible with the query. 14. The QAQQINI options used to create the materialized query table are not compatible with the QAQQINI options used to run this query. 15. The materialized query table is not usable. 16. The UNION specified in the materialized query table is not compatible with the query. 17. The constants specified in the materialized query table are not compatible with host variable values specified in the query.
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.

CPI4351	
Message Text:	Additional reason codes for query access plan has been rebuilt.
Cause Text:	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: • 20 - Referential or check constraints for member &19 of file &17 in library &18 have changed since the access plan was generated. • 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.

CPI4351	
Recovery Text:	See the prior message CPI4323 for more information.

Related reference

"CPI4323 - The query access plan has been rebuilt" on page 279

Query optimization performance information messages and open data paths

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

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

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

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

SQL7910 - All SQL cursors closed

SQL7910	
Message Text:	SQL cursors closed.
Cause Text:	SQL cursors have been closed and all Open Data Paths (ODPs) have been deleted, except those that were opened by programs with the CLOSQLCSR(*ENDJOB) option or were opened by modules with the CLOSQLCSR(*ENDACTGRP) option. All SQL programs on the call stack have completed, and the SQL environment has been exited. This process includes the closing of cursors, the deletion of ODPs, the removal of prepared statements, and the release of locks.
Recovery Text:	To keep cursors, ODPs, prepared statements, and locks available after the completion of a program, use the CLOSQLCSR precompile parameter. • The *ENDJOB option will allow the user to keep the SQL resources active for the duration of the job • 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. • 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. • The *ENDMOD option causes all SQL resources to only be accessible by the same invocation of the module. • 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.

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

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

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

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

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

SQL7911 - ODP reused

SQL7911	
Message Text:	ODP reused.
Cause Text:	An ODP that was previously created has been reused. There was a reusable Open Data Path (ODP) found for this SQL statement, and it has been used. The reusable ODP may have been from the same call to a program or a previous call to the program. A reuse of an ODP will not generate an OPEN entry in the journal.
Recovery Text:	None

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

SQL7912 - ODP created

SQL7912	
Message Text:	ODP created.
Cause Text:	An Open Data Path (ODP) has been created. No reusable ODP could be found. This occurs in the following cases: • This is the first time the statement has been run. • A RCLRSC has been issued since the last run of this statement. • The last run of the statement caused the ODP to be deleted. • If this is an OPEN statement, the last CLOSE of this cursor caused the ODP to be deleted. • The Application Server (AS) has been changed by a CONNECT statement.
Recovery Text:	If a cursor is being opened many times in an application, it is more efficient to use a reusable ODP, and not create an ODP every time. This also applies to repeated runs of INSERT, UPDATE, DELETE, and SELECT INTO statements. If ODPs are being created on every open, see the close message to determine why the ODP is being deleted.

No ODP was found that could be used again. The first time that the statement is run or the cursor is opened for a process, an ODP will always have to be created. However, if this message appears on every run of the statement or open of the cursor, the tips recommended in “Retaining cursor positions for non-ILE program calls” on page 163 should be applied to this application.

SQL7913 - ODP deleted

SQL7913	
Message Text:	ODP deleted.
Cause Text:	The Open Data Path (ODP) for this statement or cursor has been deleted. The ODP was not reusable. This could be caused by using a host variable in a LIKE clause, ordering on a host variable, or because the query optimizer chose to accomplish the query with an ODP that was not reusable.
Recovery Text:	See previous query optimizer messages to determine how the cursor was opened.

For a program that is run only once per job, this message could be normal. However, if this message appears on every run of the statement or open of the cursor, then the tips recommended in “Retaining cursor positions for non-ILE program calls” on page 163 should be applied to this application.

SQL7914 - ODP not deleted

SQL7914	
Message Text:	ODP not deleted.
Cause Text:	The Open Data Path (ODP) for this statement or cursor has not been deleted. This ODP can be reused on a subsequent run of the statement. This will not generate an entry in the journal.
Recovery Text:	None

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

SQL7915 - Access plan for SQL statement has been built

SQL7915	
Message Text:	Access plan for SQL statement has been built.
Cause Text:	SQL had to build the access plan for this statement at run time. This occurs in the following cases: • 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.

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

SQL7916 - Blocking used for query

SQL7916	
Message Text:	Blocking used for query.

SQL7916	
Cause Text:	Blocking has been used in the implementation of this query. SQL will retrieve a block of records from the database manager on the first FETCH statement. Additional FETCH statements have to be issued by the calling program, but they do not require SQL to request more records, and therefore will run faster.
Recovery Text:	SQL attempts to utilize blocking whenever possible. In cases where the cursor is not update capable, and commitment control is not active, there is a possibility that blocking will be used.

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

SQL7917 - Access plan not updated

SQL7917	
Message Text:	Access plan not updated.
Cause Text:	The query optimizer rebuilt the access plan for this statement, but the program could not be updated. Another job may be running the program. The program cannot be updated with the new access plan until a job can obtain an exclusive lock on the program. The exclusive lock cannot be obtained if another job is running the program, if the job does not have proper authority to the program, or if the program is currently being saved. The query will still run, but access plan rebuilds will continue to occur until the program is updated.
Recovery Text:	See previous messages from the query optimizer to determine why the access plan has been rebuilt. To ensure that the program gets updated with the new access plan, run the program when no other active jobs are using it.

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

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

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

SQL7918 - Reusable ODP deleted

SQL7918	
Message Text:	Reusable ODP deleted. Reason code &1.

SQL7918	
Cause Text:	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: 1. Commitment control isolation level is not compatible. 2. The statement contains SQL special register USER or CURRENT TIMEZONE, and the value for one of these registers has changed. 3. The PATH used to locate an SQL function has changed. 4. The job default CCSID has changed. 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. 6. The file, library, or member for the original ODP was changed with an override. 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. 8. An OVRDBF or DLTOVR command has been issued, causing different override options, such as different SEQONLY or WAITRCD values. 9. An error occurred when attempting to verify the statement override information is compatible with the reusable ODP information. 10. The query optimizer has determined the ODP cannot be reused. 11. The client application requested not to reuse ODPs.
Recovery Text:	Do not change the library list, the override environment, or the values of the special registers if reusable ODPs are to be used.

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

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

SQL7919 - Data conversion required on FETCH or embedded SELECT

SQL7919	
Message Text:	Data conversion required on FETCH or embedded SELECT.

SQL7919	
Cause Text:	Host variable &2 requires conversion. The data retrieved for the FETCH or embedded SELECT statement cannot be directly moved to the host variables. The statement ran correctly. Performance, however, would be improved if no data conversion was required. The host variable requires conversion for reason &1 • Reason 1 - host variable &2 is a character or graphic string of a different length than the value being retrieved. • Reason 2 - host variable &2 is a numeric type that is different than the type of the value being retrieved. • 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. • Reason 4 - host variable &2 is a variable length string and the value being retrieved is not. • Reason 5 - host variable &2 is not a variable length string and the value being retrieved is. • 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. • Reason 7 - a data conversion was required on the mapping of the value being retrieved to host variable &2, such as a CCSID conversion • 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. • Reason 10 - the length of host variable &2 is too short to hold a TIME or TIMESTAMP value being retrieved. • Reason 11 - host variable &2 is of type DATE, TIME or TIMESTAMP, and the value being retrieved is a character string. • 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. • 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. • Reason 14 - a LOB Locator was used and the commitment control level of the process was not *ALL.
Recovery Text:	To get better performance, attempt to use host variables of the same type and length as their corresponding result columns.

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

SQL7939 - Data conversion required on INSERT or UPDATE

SQL7939	
Message Text:	Data conversion required on INSERT or UPDATE.

SQL7939	
Cause Text:	The INSERT or UPDATE values cannot be directly moved to the columns because the data type or length of a value is different than one of the columns. The INSERT or UPDATE statement ran correctly. Performance, however, would be improved if no data conversion was required. The reason data conversion is required is &1. • Reason 1 is that the INSERT or UPDATE value is a character or graphic string of a different length than column &2. • Reason 2 is that the INSERT or UPDATE value is a numeric type that is different than the type of column &2. • Reason 3 is that the INSERT or UPDATE value is a variable length string and column &2 is not. • Reason 4 is that the INSERT or UPDATE value is not a variable length string and column &2 is. • Reason 5 is that the INSERT or UPDATE value is a variable length string whose maximum length is different that the maximum length of column &2. • 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. • Reason 7 is that the INSERT or UPDATE value is a character string and column &2 is of type DATE, TIME, or TIMESTAMP. • Reason 8 is that the target table of the INSERT is not a SQL table.
Recovery Text:	To get better performance, try to use values of the same type and length as their corresponding columns.

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

PRTSQLINF message reference

The following are the messages returned from PRTSQLINF.

SQL400A - Temporary distributed result file &1 was created to contain join result

SQL400A	
Message Text:	Temporary distributed result file &1 was created to contain join result. Result file was directed
Cause Text:	Query contains join criteria over a distributed file and a distributed join was performed in parallel. A temporary distributed result file was created to contain the results of the distributed join.
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming information.

SQL400B - Temporary distributed result file &1 was created to contain join result

SQL400B	
Message Text:	Temporary distributed result file &1 was created to contain join result. Result file was broadcast
Cause Text:	Query contains join criteria over a distributed file and a distributed join was performed in parallel. A temporary distributed result file was created to contain the results of the distributed join.
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming information.

SQL400C - Optimizer debug messages for distributed query step &1 and &2 follow

SQL400C	
Message Text:	Optimizer debug messages for distributed query step &1 and &2 follow.
Cause Text:	A distributed file was specified in the query which caused the query to be processed in multiple steps. The optimizer debug messages that follow provide the query optimization information about the current step.
Recovery Text:	For more information about processing of distributed files, refer to the Distributed Database Programming information.

SQL400D - GROUP BY processing generated

SQL400D	
Message Text:	GROUP BY processing generated
Cause Text:	GROUP BY processing was added to the query step. Adding the GROUP BY reduced the number of result rows which should, in turn, improve the performance of subsequent steps.
Recovery Text:	For more information refer to the SQL Programming topic.

SQL400E - Temporary distributed result file &1 was created while processing distributed subquery

SQL400E	
Message Text:	Temporary distributed result file &1 was created while processing distributed subquery
Cause Text:	A temporary distributed result file was created to contain the intermediate results of the query. The query contains a subquery which requires an intermediate result.
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 the processing of distributed files, refer to the Distributed Database Programming information.

SQL4001 - Temporary result created

SQL4001	
Message Text:	Temporary result created.
Cause Text:	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: • 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

SQL4002	
Message Text:	Reusable ODP sort used
Cause Text:	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: • 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

SQL4003	
Message Text:	UNION
Cause Text:	A UNION, EXCEPT, or INTERSECT operator was specified in the query. The messages preceding this keyword delimiter correspond to the subselect preceding the UNION, EXCEPT, or INTERSECT operator. The messages following this keyword delimiter correspond to the subselect following the UNION, EXCEPT, or INTERSECT operator.
Recovery Text:	None

SQL4004 - SUBQUERY

SQL4004	
Message Text:	SUBQUERY
Cause Text:	The SQL statement contains a subquery. The messages preceding the SUBQUERY delimiter correspond to the subselect containing the subquery. The messages following the SUBQUERY delimiter correspond to the subquery.
Recovery Text:	None

SQL4005 - Query optimizer timed out for table &1

SQL4005	
Message Text:	Query optimizer timed out for table &1
Cause Text:	The query optimizer timed out before it could consider all indexes built over the table. This is not an error condition. The query optimizer may time out in order to minimize optimization time. The query can be run in debug mode (STRDBG) to see the list of indexes which were considered during optimization. The table number refers to the relative position of this table in the query.

SQL4005	
Recovery Text:	To ensure an index is considered for optimization, specify the logical file of the index as the table to be queried. The optimizer will first consider the index of the logical file specified on the SQL select statement. Note that SQL created indexes cannot be queried. An SQL index can be deleted and recreated to increase the chances it will be considered during query optimization. Consider deleting any indexes no longer needed.

SQL4006 - All indexes considered for table &1

SQL4006	
Message Text:	All indexes considered for table &1
Cause Text:	The query optimizer considered all index built over the table when optimizing the query. The query can be run in debug mode (STRDBG) to see the list of indexes which were considered during optimization. The table number refers to the relative position of this table in the query.
Recovery Text:	None

SQL4007 - Query implementation for join position &1 table &2

SQL4007	
Message Text:	Query implementation for join position &1 table &2
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 46 for more information about join optimization and tips to influence join order.

SQL4008 - Index &1 used for table &2

SQL4008	
Message Text:	Index &1 used for table &2
Cause Text:	The index was used to access rows from the table for one of the following reasons: • Row selection • Join criteria. • Ordering/grouping criteria. • Row selection and ordering/grouping criteria. • The table number refers to the relative position of this table in the query. The query can be run in debug mode (STRDBG) to determine the specific reason the index was used
Recovery Text:	None

SQL4009 - Index created for table &1

SQL4009	
Message Text:	Index created for table &1

SQL4009	
Cause Text:	A temporary index was built to access rows from the 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:	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

SQL401A	
Message Text:	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 information.

SQL401B - Temporary distributed result table &1 was created while processing grouping criteria

SQL401B	
Message Text:	Temporary distributed result table &1 was created while processing grouping criteria
Cause Text:	A temporary distributed result table was created to contain the intermediate results of the query. Either the query contains grouping columns (GROUP BY) that do not match the partitioning keys of the distributed table or the query contains grouping criteria but no grouping columns were specified.
Recovery Text:	For more information about processing of distributed tables, refer to the Distributed Database Programming information.

SQL401C - Performing distributed join for query

SQL401C	
Message Text:	Performing distributed join for query
Cause Text:	Query contains join criteria over a distributed table and a distributed join was performed in parallel. See the following SQL401F messages to determine which tables were joined together.
Recovery Text:	For more information about processing of distributed tables, refer to the Distributed Database Programming information.

SQL401D - Temporary distributed result table &1 was created because table &2 was directed

SQL401D	
Message Text:	Temporary distributed result table &1 was created because table &2 was directed
Cause Text:	Temporary distributed result table was created to contain the intermediate results of the query. Data from a distributed table in the query was directed to other nodes.
Recovery Text:	Generally, a table is directed when the join columns do not match the partitioning keys of the distributed table. When a table is directed, the query is processed in multiple steps and processed in parallel. A temporary distributed result file is required to contain the intermediate results for each step. For more information about processing of distributed tables, refer to the Distributed Database Programming information.

SQL401E - Temporary distributed result table &1 was created because table &2 was broadcast

SQL401E	
Message Text:	Temporary distributed result table &1 was created because table &2 was broadcast
Cause Text:	Temporary distributed result table was created to contain the intermediate results of the query. Data from a distributed table in the query was broadcast to all nodes.
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 information.

SQL401F - Table &1 used in distributed join

SQL401F	
Message Text:	Table &1 used in distributed join
Cause Text:	Query contains join criteria over a distributed table and a distributed join was performed in parallel.
Recovery Text:	For more information about processing of distributed tables, refer to the Distributed Database Programming information.

SQL4010 - Table scan access for table &1

SQL4010	
Message Text:	Table scan access for table &1
Cause Text:	Table scan access was used to select rows from the table. The table number refers to the relative position of this table in the query.
Recovery Text:	Table scan is generally a good performing option when selecting a high percentage of rows from the table. The use of an index, however, may improve the performance of the query when selecting a low percentage of rows from the table.

SQL4011 - Index scan-key row positioning used on table &1

SQL4011	
Message Text:	Index scan-key row positioning used on table &1
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 on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about index scan-key row positioning.

SQL4012 - Index created from index &1 for table &2

SQL4012	
Message Text:	Index created from index &1 for table &2
Cause Text:	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

SQL4013	
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

SQL4014	
Message Text:	&1 join column pair(s) are used for this join position
Cause Text:	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. If 0 join column pairs were specified then index scan-key row positioning with row selection was used instead of join selection.

SQL4014	
Recovery Text:	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. If 0 join column pairs were specified then index scan-key row positioning was used. Index scan-key row positioning is normally a good performing option. Message SQL4011 provides more information about index scan-key row positioning.

SQL4015 - From-column &1.&2, to-column &3.&4, join operator &5, join predicate &6

SQL4015	
Message Text:	From-column &1.&2, to-column &3.&4, join operator &5, join predicate &6
Cause Text:	Identifies which join predicate was implemented at the current join position. The replacement text parameters are: • &1: The join 'from table' number. The table number refers to the relative position of this table in the query. • &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). • &3: The join 'to table' number. The table number refers to the relative position of this table in the query. • &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). • &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). • &6. The join predicate number. Identifies the join predicate within this set of join pairs.
Recovery Text:	Refer to "Join optimization" on page 46 for more information about joins.

SQL4016 - Subselects processed as join query

SQL4016	
Message Text:	Subselects processed as join query
Cause Text:	The query optimizer chose to implement some or all of the subselects with a join query. Implementing subqueries with a join generally improves performance over implementing alternative methods.
Recovery Text:	None

SQL4017 - Host variables implemented as reusable ODP

SQL4017	
Message Text:	Host variables implemented as reusable ODP
Cause Text:	The query optimizer has built the access plan allowing for the values of the host variables to be supplied when the query is opened. This query can be run with different values being provided for the host variables without requiring the access plan to be rebuilt. This is the normal method of handling host variables in access plans. The open data path (ODP) that will be created from this access plan will be a reusable ODP.

SQL4017	
Recovery Text:	Generally, reusable open data paths perform better than non-reusable open data paths.

SQL4018 - Host variables implemented as non-reusable ODP

SQL4018	
Message Text:	Host variables implemented as non-reusable ODP
Cause Text:	The query optimizer has implemented the host variables with a non-reusable open data path (ODP).
Recovery Text:	This can be a good performing option in special circumstances, but generally a reusable ODP gives the best performance.

SQL4019 - Host variables implemented as file management row positioning reusable ODP

SQL4019	
Message Text:	Host variables implemented as file management row positioning reusable ODP
Cause Text:	The query optimizer has implemented the host variables with a reusable open data path (ODP) using file management row positioning.
Recovery Text:	Generally, a reusable ODP performs better than a non-reusable ODP.

SQL402A - Hashing algorithm used to process join

SQL402A	
Message Text:	Hashing algorithm used to process join
Cause Text:	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 dials. Debug messages detailing the implementation of each hash dial can be found in the joblog if the query is run in debug mode using the STRDBG CL command.
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 "Data access on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about hashing algorithm for join processing.

SQL402B - Table &1 used in hash join step &2

SQL402B	
Message Text:	Table &1 used in hash join step &2

SQL402B	
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 on DB2 UDB for iSeries: data access paths and methods” on page 8 for more information about hashing

SQL402C - Temporary table created for hash join results

SQL402C	
Message Text:	Temporary table created for hash join results
Cause Text:	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 on DB2 UDB for iSeries: data access paths and 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

SQL402D	
Message Text:	Query attributes overridden from query options file &2 in library &1
Cause Text:	None
Recovery Text:	None

SQL4020 - Estimated query run time is &1 seconds

SQL4020	
Message Text:	Estimated query run time is &1 seconds
Cause Text:	The total estimated time, in seconds, of executing this query.
Recovery Text:	None

SQL4021 - Access plan last saved on &1 at &2

SQL4021	
Message Text:	Access plan last saved on &1 at &2
Cause Text:	The date and time reflect the last time the access plan was successfully updated in the program object.
Recovery Text:	None

SQL4022 - Access plan was saved with SRVQRY attributes active

SQL4022	
Message Text:	Access plan was saved with SRVQRY attributes active
Cause Text:	The access plan that was saved was created while SRVQRY was active. Attributes saved in the access plan may be the result of SRVQRY.
Recovery Text:	The query will be re-optimized the next time it is run so that SRVQRY attributes will not be permanently saved.

SQL4023 - Parallel table prefetch used

SQL4023	
Message Text:	Parallel table prefetch used
Cause Text:	The query optimizer chose to use a parallel prefetch access method to reduce the processing time required for the table scan.
Recovery Text:	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: • 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. For more information about parallel table prefetch, refer to "Data access on DB2 UDB for iSeries: data access paths and methods" on page 8

SQL4024 - Parallel index preload access method used

SQL4024	
Message Text:	Parallel index preload access method used
Cause Text:	The query optimizer chose to use a parallel index preload access method to reduce the processing time required for this query. This means that the indexes used by this query will be loaded into active memory when the query is opened.
Recovery Text:	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: • 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. For more information about parallel index preload, refer to the "Data access on DB2 UDB for iSeries: data access paths and methods" on page 8.

SQL4025 - Parallel table preload access method used

SQL4025	
Message Text:	Parallel table preload access method used

SQL4025	
Cause Text:	The query optimizer chose to use a parallel table preload access method to reduce the processing time required for this query. This means that the data accessed by this query will be loaded into active memory when the query is opened.
Recovery Text:	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: • 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. For more information about parallel table preload, refer to "Data access on DB2 UDB for iSeries: data access paths and methods" on page 8.

SQL4026 - Index only access used on table number &1

SQL4026	
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 on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about index only access.

SQL4027 - Access plan was saved with DB2 UDB Symmetric Multiprocessing installed on the system

SQL4027	
Message Text:	Access plan was saved with DB2 UDB Symmetric Multiprocessing installed on the system
Cause Text:	Text: The access plan saved was created while the system feature DB2 UDB Symmetric Multiprocessing was installed on the system. The access plan may have been influenced by the presence of this system feature. 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 UDB Symmetric Multiprocessing can influence a query, refer to "Control parallel processing for queries" on page 132.

SQL4028 - The query contains a distributed table

SQL4028	
Message Text:	The query contains a distributed table

SQL4028	
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 information.

SQL4029 - Hashing algorithm used to process the grouping

SQL4029	
Message Text:	Hashing algorithm used to process the grouping
Cause Text:	The grouping specified within the query was implemented with a hashing algorithm.
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 ALWCPYDTA(*YES). Refer to "Data access on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about the hashing algorithm.

SQL4030 - &1 tasks specified for parallel scan on table &2.

SQL4030	
Message Text:	&1 tasks specified for parallel scan on table &2.
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 on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about parallel scan.

SQL4031 - &1 tasks specified for parallel index create over table &2

SQL4031	
Message Text:	&1 tasks specified for parallel index create over table &2
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.

SQL4031	
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 on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about parallel index create.

SQL4032 - Index &1 used for bitmap processing of table &2

SQL4032	
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 on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about bitmap processing.

SQL4033 - &1 tasks specified for parallel bitmap create using &2

SQL4033	
Message Text:	&1 tasks specified for parallel bitmap create using &2
Cause Text:	The query optimizer has calculated the optimal number of tasks to use to create the bitmap based on the query attribute degree.
Recovery Text:	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 on DB2 UDB for iSeries: data access paths and methods" on page 8 for more information about parallel scan.

SQL4034 - Multiple join classes used to process join

SQL4034	
Message Text:	Multiple join classes used to process join

SQL4034	
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 46 for more information about join classes.

SQL4035 - Table &1 used in join class &2

SQL4035	
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 46 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:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

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 i5/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

| DB2
| DB2 Universal Database
| DRDA
| i5/OS
| IBM
| iSeries
| Language Environment
| Net.Data
| SP
| WebSphere

Other company, product, and 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