DB2 Information Management Software

IBM

SQL Performance Diagnosis

on IBM DB2 Universal Database for iSeries

- Discover the tools to identify SQL performance problems
- Unleash the capabilities of the SQL Performance Monitors
- Learn to query the Database Monitor performance data

Hernando Bedoya
Elvis Budimlic
Morten Buur Rasmussen
Peggy Chidester
Fernando Echeveste
Birgitta Hauser
Kang Min Lee
Dave Squires

Redbooks

ibm.com/redbooks

International Technical Support Organization

SQL Performance Diagnosis on IBM DB2 Universal Database for iSeries

May 2006

SG24-6654-00

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

First Edition (May 2006)

This edition applies to Version 5, Release 3, Modification 0 of IBM i5/OS, product number 5722-SS1.

Contents

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 give 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.

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.

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

@server®	DB2 Universal Database™	Redbooks™
@server®	DB2®	System i5™
Redbooks (logo) ™	DRDA®	System/38™
iSeries™	First Run™	SQL/400®
i5/OS®	IBM®	
AS/400®	OS/400®	

The following terms are trademarks of other companies:

Java, JDBC, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Excel, Microsoft, Visual Basic, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Preface

The goal of database performance tuning is to minimize the response time of your queries. It is also to optimize your server's resources by minimizing network traffic, disk I/O, and CPU time.

This IBM® Redbook helps you to understand the basics of identifying and tuning the performance of Structured Query Language (SQL) statements using IBM DB2® Universal Database™ for iSeries™. DB2 Universal Database for iSeries provides a comprehensive set of tools that help technical analysts tune SQL queries. The SQL Performance Monitors are part of the set of tools that IBM i5/OS® provides for assisting in SQL performance analysis since Version 3 Release 6. These monitors help to analyze database performance problems after SQL requests are run.

This redbook also presents tips and techniques based on the SQL Performance Monitors and other tools, such as Visual Explain. You'll find this guidance helpful in gaining the most out of both DB2 Universal Database for iSeries and query optimizer when using SQL.

Note: In this book, we use the name "SQL Performance Monitors" when using iSeries Navigator. SQL Performance Monitor has two versions: Detailed Database Monitor and Summary Monitor. We refer to the tool as "Database Monitors" when using a green screen and accessing the tool by running the Start Database Monitor (STRDBMON) CL command.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Rochester Center.

Hernando Bedoya is an IT Specialist at the IBM ITSO, in Rochester, Minnesota. He writes extensively and teaches IBM classes worldwide in all areas of DB2 Universal Database for iSeries. Before joining the ITSO more than five years ago, he worked for IBM Colombia as an IBM AS/400® IT Specialist doing presales support for the Andean countries. He has 20 years of experience in the computing field and has taught database classes in Colombian universities. He holds a master degree in computer science from EAFIT, Colombia. His areas of expertise are database technology, application development, and data warehousing.

Elvis Budimlic is Director of Development at Centerfield Technology, Inc. in Rochester, Minnesota. His primary responsibility is development and support of Centerfield's insure/SQL toolset with a focus on DB2 Universal Database for iSeries SQL performance and optimization. His area of expertise are database performance, software engineering, problem diagnostics, and iSeries work management. Before joining Centerfield, he worked for IBM for four years on the database SLIC development group and Mylex RAID software development teams. He holds a bachelor degree in computer science from Winona State University in Minnesota.

Morten Buur Rasmussen is a senior IT Specialist at the IBM Client Technology Center (CTC), in La Gaude, France. He covers Europe and the Middle East in client facing areas of IBM @server iSeries, database, and WebSphere performance. Before joining the CTC, he worked for IBM Denmark and different European banks. He has 18 years of experience in the

computing field. His areas of expertise are database technology, application development, and iSeries work management.

Peggy Chidester is a Staff Software Engineer working on the database team in the IBM Rochester Support Center. She has been a member of this team since being hired by IBM seven years ago and specializes in query performance. She has also worked temporarily on the performance team in the Support Center as well as a liason with development. Peggy has given new employees an introduction to database and taught database support employees in Milan, Italy, about database performance. She has written numerous documents for the Software Knowledge Base.

Fernando Echeveste is a Staff Software Engineer for IBM in Rochester, Minnesota. He is a member of the IBM DB2 Universal Database for iSeries performance team. He has 12 years of experience in the computing field. His areas of expertise are database technology and client-server architectures. Before joining IBM in Rochester more than seven years ago, he worked for IBM Guadalajara as a software engineer doing development and support for Client Access/400 components such as Remote SQL, Transfer Function, and Open Database Connectivity (ODBC). He holds a bachelor of science degree in computer engineering from Western Institute of Advanced Technological Studies in Guadalajara, Mexico.

Birgitta Hauser has been a Software Engineer since 1996, focusing on RPG and SQL development on iSeries at Lunzer + Partner GmbH in Germany. She graduated with a degree in business economics, and started programming on the AS/400 in 1992. She is responsible for the complete RPG, ILE, and Database programming concepts for Lunzer + Partner's Warehouse Management Software Package. She also works in education as a trainer for RPG and SQL developers. Since 2002, she has frequently spoken at the COMMON User Group in Germany. In addition, she is chief editor of the iSeries Nation Network (iNN, the German part of iSeries Nation), eNews, and the author of several papers focusing on RPG and SQL.

Kang Min Lee is an IT Specialist at IBM, based in Korea. He has more than six years of experience in working with the IBM System i platform. Prior to joining IBM Korea, he had four years of experience in applications development using DB2 Universal Database for AS/400 and iSeries. His areas of expertise are database technology, database performance, and application development. Kang Min currently supports database and applications on @server i5 and iSeries in Korea and provides general System i technical support.

Dave Squires works in the iSeries Support Center database team in the United Kingdom (UK), where he provides first and second level support to clients in the UK and the Middle East. He has been working on the iSeries and System/38™ since he started working in the computing field in 1984.

Thanks to the following people for their contributions to this project:

Thomas Gray
Marvin Kulas
Joanna Pohl-Miszczyk
Jenifer Servais
ITSO, Rochester Center

Mark Anderson
Robest Bestgen
Michael Cain
Kevin Chidester
Daniel Cruikshank
Jim Flanagan
Kent Milligan

Brian Muras
Denise Voy Tompkin
IBM Rochester

Peter Bradley
IBM UK

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. JLU Building 107-2
  3605 Highway 52N
  Rochester, Minnesota 55901-7829

Part 1

Introduction to DB2 Universal Database and database performance tools

In this part, we introduce basic information about DB2 Universal Database for iSeries performance. We also introduce the different tools for analyzing database performance.

This part includes the following chapters:

Note: In this book, we use the name “SQL Performance Monitors” when using iSeries Navigator. SQL Performance Monitor has two versions: Detailed Database Monitor and Summary Monitor. We refer to the tool as “Database Monitors” when using a green screen and accessing the tool by running the Start Database Monitor (STRDBMON) CL command.

1

Determining whether you have an SQL performance problem

In this chapter, we explain how to determine if a performance problem is the result of poorly performing SQL queries. We describe the various system tools and a methodology to help you find an SQL performance problem.

This chapter guides you in:

- Asking the right questions to determine whether you have a performance problem
- Knowing what to check when you have a performance problem
- Determining if a performance problem is related to SQL

1.1 Questions to ask yourself

If you are reading this book, you most likely currently have an SQL performance problem, have experienced an SQL performance problem in the past, or are interested in learning how to diagnose SQL performance. An important step in looking into any SQL performance problem is to ask a few questions to determine whether the problem is with your SQL or a system-wide performance problem. To clarify the problem, you must ask yourself the following questions:

- How do you know that there is a problem?
- Where is the problem occurring?
- Did you ever have a good working situation?
  - If you have a job or program that used to run well but doesn't do so now, what changed since it last ran well?
- Do SQL queries appear to have a performance problem?
  - If the problem is with multiple queries, is there any commonality among the queries?
  - If the problem is with one query, what is the change in run time?
  - Have you ever tuned your queries for performance?

We explain each of these questions in more details in the sections that follow.

1.2 How do you know that there is a problem?

This question is very basic, yet vital. Identifying how you know whether there is a problem determines the steps to take to analyze the problem.

For example, you are reading a report that shows that CPU spiked during a time frame. Is this a problem? Is CPU spiking bad? Are you seeing a trend or is this a one-time occurrence? During the time frame, did you receive complaints from users saying that they were unable to get their work done? Such complaints would be a strong indicator that there is an actual problem.

We recommend that you create a spreadsheet to document and keep a history of problems. Table 1-1 shows an example of how problem record keeping can be helpful in problem solving.

Table 1-1 Example of problem record keeping

Date of problem	Time of problem	User reporting	Problem job	Reported problem	How problem discovered	How problem resolved	Data gathered
01/05/05	13:01:00	Sales Report	QZDASOINIT	Report taking longer to generate	User found	Note 1	SQL Performance Monitor started
Note 1: The queries to generate the sales report usually run in a dedicated memory pool. However, for some reason still unknown, the pool identifier in the QZDASOINIT prestart job was changed so that the queries started to run in *BASE along with some other high demanding jobs on the system. By changing the prestart job entry, the queries went back to run in their dedicated memory pool and the response time reverted to what it was in the past.							

In the example shown in Table 1-1, it could look like an SQL problem since an SQL query was taking a long time. However, further analysis showed that other work done in the pool was the cause of the problem. In Appendix A, "Tools to check a performance problem" on page 253,

we explain how to use the system commands and tools to investigate whether the problem is related to the system or to an SQL query. There are a number of solutions to try to resolve the problem shown in Table 1-1:

- Separate the QZDASOINIT jobs into their own pool.

  For more information, see Software Knowledge Base document “Separating Batch Work from *BASE”, document number 349033974 on the Web at:

  http://www-912.ibm.com/s_dir/slkbase.NSF/1ac66549a21402188625680b0002037e/9fa68bd7573e48af862565c2007d3d9b?OpenDocument

- Verify that the QPFRADJ system value is set to automatically adjust memory in the pools.
- Add more memory to the pool.

1.3 Where is the problem occurring?

Another important step in analyzing an SQL performance problem is to identify where the problem is occurring. It is helpful to understand the components of work involved whenever a user executes any SQL request. Figure 1-1 illustrates the different components of work involved in the execution of an SQL request.

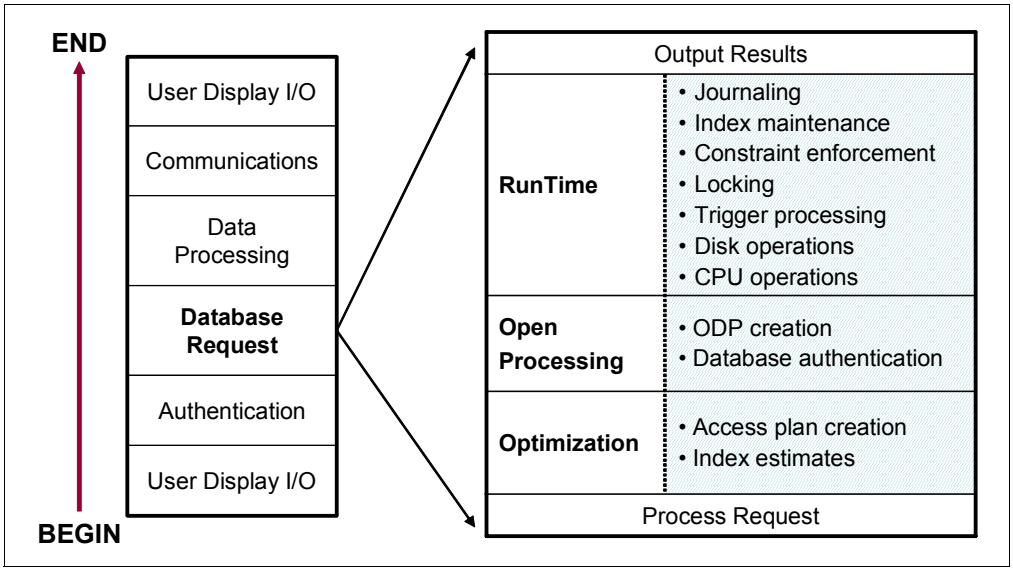


Figure 1-1 Components of work

Isolate the problem to the smallest number of factors as possible. Before we start analyzing the SQL or database request, we must understand that other variables are involved in the total response time such as:

- User display I/O
- Authentication
- Data processing
- Communications

You might ask some additional questions to find where the problem is occurring, such as:

- Is the problem occurring only in one pool?
  - What activity is occurring in the pool?
- Are you having a performance problem running tasks on the iSeries server, even simple commands?
- Is the problem occurring only within specific jobs, that is batch or QZDASOINIT?
- Do you hear only from one set of users when the problem occurs?
  - What is the commonality among the users?
- Does the problem occur only when going through a specific interface, such as WebSphere, SQL, or Query/400?
- Are the jobs having problems running remotely or locally?
- Can the problem be isolated to a specific program?
  - Is SQL embedded?
  - What type of program is it?
- Can the problem be isolated to a specific SQL statement?

You need to examine all of the answers to these questions to see if you can isolate the problem to jobs running SQL. If the jobs that have a problem are QZDASOINIT or QSQSRVR, then it is likely that they are running SQL. QRWTSRVR jobs are quite often used to run SQL, but are also used if you are using distributed data management (DDM) files. When the problem occurs only in certain jobs or for a certain group of users, you must review the environment, such as communications, pools, job priorities, and so on.

After you isolate the problem and are now certain that the problem is on the SQL request, you must understand what it takes to execute an SQL request. Upon the execution of an SQL request, three main operations are done as shown in Figure 1-2:

- Optimization time
- Open processing time
- Run time

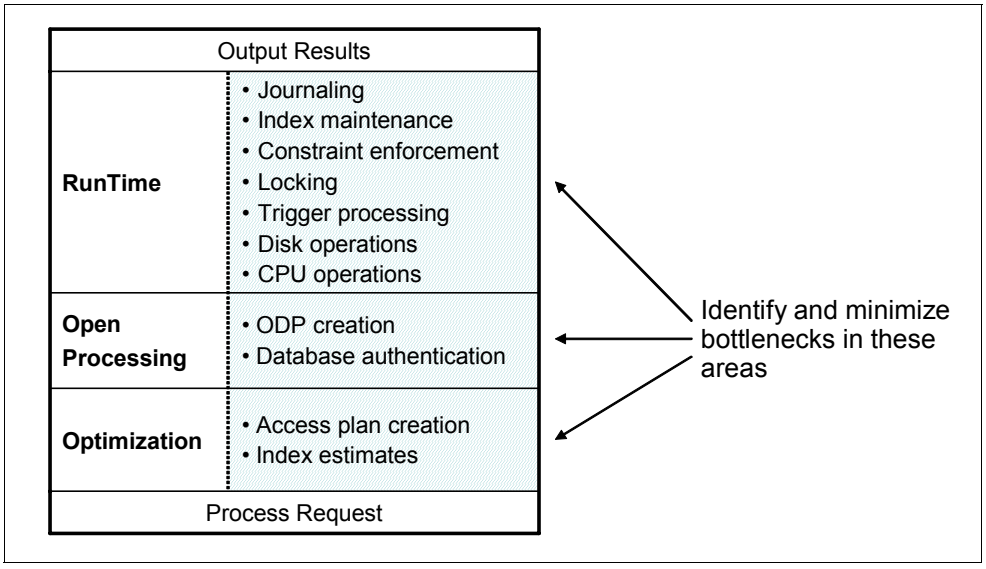


Figure 1-2 Components of work: Database request

In Figure 1-2, you can see which operations affect optimization time, open processing time, and run time. It is key that you identify and minimize the bottlenecks in these three main areas of the processing of an SQL request.

Understanding the tools that we describe in this book will help you to identify the bottlenecks in these areas.

1.4 Did you ever have a good working situation?

A *good working situation* is when the system or an application is running without any performance problems. Knowing if you had a good working situation involves understanding the history of an application or the system. Some SQL performance problems are caused by:

- The introduction of a new application
- The application of new program temporary fixes (PTFs)
- An upgrade to a newer i5/OS release
- Changes to system values

In this book, you learn how to use some tools to gather performance information prior to any major change on the system. You learn how to determine SQL performance problems in cases where previously there was a good working situation or where you are unsure whether you previously had a good working situation. In a scenario where you know that you had a good working condition, it is vital to document the timeline of what happened since the system or application last ran well. Make sure that you document any changes, such as those that we previously listed.

1.5 Do SQL queries appear to have performance problems?

It might be difficult to answer this question if just one SQL query has a problem or if multiple SQL queries have a problem, since the issue is often found at the job level. In the following chapters, we show you how to use a different set of tools to investigate which queries are having problems, if you have not already made that determination. It is important to differentiate between one query having a performance problem and many SQL queries having a problem.

- One SQL query having a performance problem

  When it appears that a single SQL query has a performance problem, you must know the run time of the specific query before the performance problem appeared. Additionally, you must know how you gather the runtime data.

- Multiple SQL queries having performance problems

  In situations where it appears that multiple SQL queries have performance problems, you must ask additional questions to try to find any commonality among the queries:

  - Do the queries that have the performance problem all use the same table?
  - Does the problem appear to be with a specific type of query, such as left outer joins or updates?
  - Do only specific users run the queries?
  - Do all the queries run in the same pool?
  - Have you ever tuned your queries for performance?

  When multiple queries have a problem, it indicates the need for you to review the environment and examine such aspects as communications, pools, job priorities, and so on. See Appendix A, “Tools to check a performance problem” on page 253, for more information.

One question to keep in mind when you are examining SQL performance problems is: “Did you ever tune your queries?” SQL queries require the existence of indexes to obtain statistics and for implementation (access of data) of the query request. For more information about an adequate indexing strategy, refer to the white paper *Indexing and statistics strategy for DB2 UDB for iSeries*

http://www.ibm.com/servers/enable/site/education/abstracts/indxng_abs.html

In the following chapters, we explain how to use SQL Performance Monitors and other tools to determine if your queries need to be tuned for performance. In Chapter 9, “Tips to prevent further database performance problems” on page 229, we provide tips to help you avoid SQL performance problems.

2

DB2 Universal Database for iSeries performance basics

In this chapter, we introduce some of the basic concepts of SQL performance on DB2 Universal Database for iSeries. We discuss the indexing technology on DB2 Universal Database for iSeries. We also introduce the query engines on DB2 Universal Database for iSeries, Classic Query Engine (CQE) and SQL Query Engine (SQE).

In addition, we discuss Star Join Schema support on DB2 Universal Database for iSeries with a key feature called *Lookahead Predicate Generation*.

2.1 Basics of indexing

DB2 Universal Database for iSeries has two kinds of persistent indexes:

- *Binary radix tree indexes*, which have been available since 1988
- *Encoded-vector indexes* (EVIs), which became available in 1998 with V4R3

Both types of indexes are useful in improving performance for certain kinds of queries. In this section, we introduce this indexing technology and how it can help you in SQL performance.

2.1.1 Binary radix tree 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 house the address of the rows in the base table that are associated with the key value. The *key value* is used to quickly navigate to the leaf node with a few simple binary search tests.

Figure 2-1 shows the structure of a binary radix tree index.

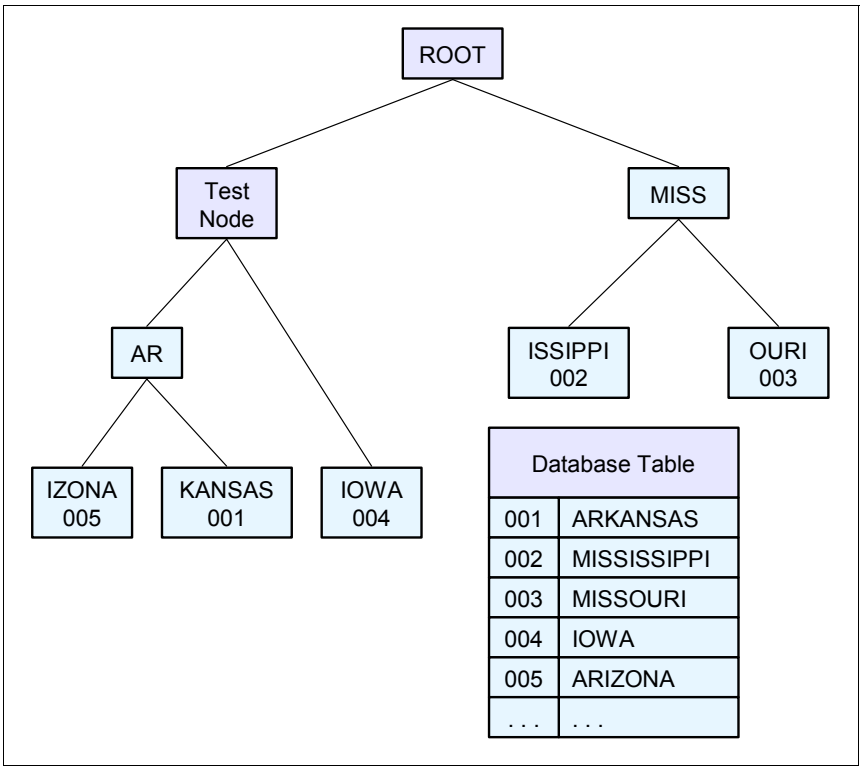


Figure 2-1 Binary radix tree index

Thus, a single key value can be accessed quickly with a small number of tests. This quick access is pretty consistent across all key values in the index, since the server keeps the depth of the index shallow and the index pages spread across multiple disk units.

The binary radix tree structure is good for finding a small number of rows because it can 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 online transaction processing (OLTP) request, such as “find the outstanding orders for a single customer,” results in fast performance. An index created over the customer number field is considered the perfect index for this type of

query because it allows the database to focus on the rows that it needs and perform a minimal number of I/Os.

2.1.2 Encoded-vector index

To understand EVIs, you should have a basic knowledge of bitmap indexing. DB2 Universal Database for iSeries does not create permanent bitmaps. SQL creates dynamic bitmaps temporarily for query optimization.

The need for newer index technologies has spawned the generation of a variety of similar solutions that can be collectively referred to as *bitmap indexes*. A bitmap index is an array of distinct values. For each value, the index stores a bitmap, where each bit represents a row in the table. If the bit is set on, then that row contains the specific key value.

Table 2-1 shows the bitmap representation of an index.

Table 2-1 Bitmap index

Key	Bit-Array
Arkansas	10000000110000010110
Arizona	01000100000011010000
. . .	
Virginia	00000011000000100000
Wyoming	00001000000100000001

With this indexing scheme, bitmaps can be combined dynamically using Boolean arithmetic (ANDing and ORing) to identify only those rows that are required by the query. Unfortunately, this improved access comes with a price. In a very large database (VLDB) environment, bitmap indexes can grow to an ungainly size. For example, in a one billion row table, you might have one billion bits for each distinct value. If the table contains many distinct values, the bitmap index quickly becomes enormous. Usually, relational database management systems (RDBMSs) rely on a type of compression algorithm to help alleviate this growth problem.

An EVI is created using the SQL command CREATE ENCODED VECTOR INDEX as in the following example:

```
CREATE ENCODED VECTOR INDEX MySchema.EVI_Name
    ON MySchema.Table_Name (MyColumn)
  WITH n DISTINCT VALUES
```

An 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 of bitmap indexing. Because of their compact size and relative simplicity, EVIs provide faster scans of a table that can also be processed in parallel.

An EVI is a data structure that is stored basically as two components:

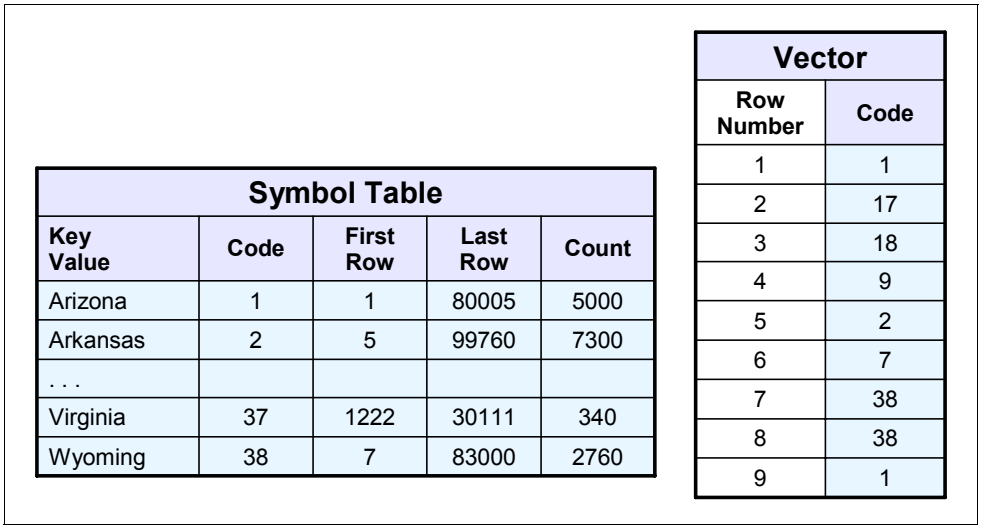
- Symbol table

  The symbol table contains a distinct key list, along with statistical and descriptive information about each distinct key value in the index. This table maps each distinct value to a unique code. The mapping of any distinct key value to a 1-, 2-, or 4-byte code provides a type of key compression. Any key value, of any length, can be represented by a small bytecode. Additional key information, such as first and last row and number of occurrences, helps to get faster access to the data.

- Vector

  The vector contains a bytecode value for each row in the table. This bytecode represents the actual key value found in the symbol table and the respective row in the database table. The bytecodes are in the same ordinal position in the vector, as the row it represents in the table. The vector does not contain any pointer or explicit references to the data in the table.

Figure 2-2 shows the components of an EVI.



Symbol Table

Key Value	Code	First Row	Last Row	Count
Arizona	1	1	80005	5000
Arkansas	2	5	99760	7300
. . .				
Virginia	37	1222	30111	340
Wyoming	38	7	83000	2760

Vector

Row Number	Code
1	1
2	17
3	18
4	9
5	2
6	7
7	38
8	38
9	1

Figure 2-2 Encoded-vector index

Note: Because the vector represents a relative record number list, an EVI cannot be used to order records. EVIs also have a limited use in joins.

When executing queries that contain joins, grouping, and ordering, a combination of binary radix indexes and EVIs might be used to implement the query. When the selected row set is relatively small, a binary radix index usually performs faster access. When the selected row set is roughly between 20% and 70% of the table being queried, table probe access using a bitmap, created from an EVI or binary radix index, is the best choice.

Also, the optimizer and database engine have the ability to use more than one index to help with selecting the data. This technique might be used when the local selection contains AND or OR conditions, a single index does not contain all the proper key columns, or a single index cannot meet all of the conditions. Single key EVIs can help in this scenario since the bitmaps or relative record number (RRN) lists created from the EVIs can be combined to narrow down the selection process.

Recommendation for EVI use

EVIs are a powerful tool for providing fast data access in decision support and query reporting environments. However, to ensure the effective use of EVIs, you must implement them using the guidelines:

Create EVIs on:

- Read-only tables or tables with a minimum of INSERT, UPDATE, and DELETE activity
- Key columns that are used in the WHERE clause: local selection predicates of SQL requests, and fact table join columns when using Star Join Schema support

- 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
- Nonunique key columns, with many duplicates

Create EVIs with the maximum bytecode size expected:

- Use the WITH n DISTINCT VALUES clause on the CREATE ENCODED VECTOR INDEX statement.
- If unsure, consider using a number greater than 65535 to create a 4-byte code, avoiding the EVI maintenance overhead of switching bytecode sizes as additional new distinct key values are inserted.

When loading data, keep in mind that:

- You drop EVIs, load the data, and then create EVIs.
- EVI bytecode size is assigned automatically based on the number of actual distinct key values found in the table.
- The symbol table contains all key values, in order; there are no keys in the overflow area.

2.2 Query engines: An overview

Data is the key. Quick and reliable access to business data is critical to making crucial business decisions. A robust database management system (DBMS) has excellent performance capabilities and automated, built-in management and administration functionality. It allows businesses to concentrate on making decisions based on the information contained in their database, rather than managing the database.

Integrated into IBM OS/400® (i5/OS), DB2 Universal Database for iSeries has its roots in the integrated relational database of the IBM System/38, the predecessor of the AS/400 and iSeries servers. Although the database was always relational in nature, native file operations were used to access the data.

With the debut of the AS/400 in 1988 came the introduction of SQL on the platform. SQL is an industry standard (SQL 2003) to define database objects (Data Definition Language (DDL)) and manipulate database data (Data Manipulation Language (DML)). SQL provides an alternative and additional method for accessing data. Both SQL and native methods can coexist. Focusing primarily on OLTP applications, the database has satisfied customer requirements for well over 20 years.

More recently a new breed of applications started to dominate development efforts. These applications are designed to accommodate rapidly changing business needs and processes. To address the issues and satisfy the demands of the new application world, IBM considered the following options:

- Continue to enhance the existing product
- Acquire a new database technology
- Re-engineer the existing product

The continual enhancement of the product did not seem to be a viable proposition. The increasing development resources required to maintain the existing code resulted in a reduction of resources available to provide new functionality in a timely manner.

Acquiring a new database technology would compromise the basic tenets that distinguish the iSeries from the rest of the industry. These include the integration of the database within OS/400 and the ease-of-use characteristics of the database that minimize administration efforts. Losing these characteristics would significantly reduce the cost of ownership benefits of the iSeries.

Re-engineering the existing product was a more practical solution. However, this could easily become an overwhelming and potentially unsustainable task if an attempt was made to re-engineer the entire product. It could also impact portions of the product that continue to provide solid and efficient support to existing applications and functions.

After considering the options, IBM chose to re-engineer the product. We did so with the added decision to focus only on those aspects of the product for which re-engineering offered the greatest potential. The potential offered the ability to:

- Support modern application, database, and transactional needs
- Allow the continued development of database functionality in an efficient and timely manner
- Maintain and enhance the self-managing value proposition of DB2 Universal Database for iSeries
- Provide a foundation to handle increasingly more complex query environments
- Improve query performance consistency and predictability
- Incorporate state-of-the-art techniques

In line with this decision, the query engine was identified as an area that would benefit substantially from such a re-engineering effort. The best current technologies and algorithms, coupled with modern object-oriented design concepts and object-oriented programming implementation, were applied in the redesign of the query engine and its components.

To guarantee existing applications continue to work and to make new or even existing applications profit from the new designed product, IBM decided to implement an additional query engine. The newly redesigned query engine in DB2 Universal Database for iSeries is the *SQL Query Engine*. The existing query engine is referred to as the *Classic Query Engine*. Both query engines coexist in the same system.

The staged implementation of SQE enabled a limited set of queries to be routed to SQE in V5R2. In general, read-only single table queries with a limited set of attributes were routed to SQE. Over time, more queries will use SQE, and increasingly fewer queries will use CQE. At some point, all queries, or at least those that originate from SQL interfaces, will use SQE.

Note: SQE processes queries only from SQL interfaces, such as interactive and embedded SQL, Open Database Connectivity (ODBC) and Java™ Database Connectivity (JDBC™).

2.2.1 Database architecture before V5R2M0

For systems prior to the release of V5R2M0, all database requests are handled by the CQE. Figure 2-3 shows a high-level overview of the architecture of DB2 Universal Database for iSeries before OS/400 V5R2. The optimizer and database engine are implemented at different layers of the operating system.

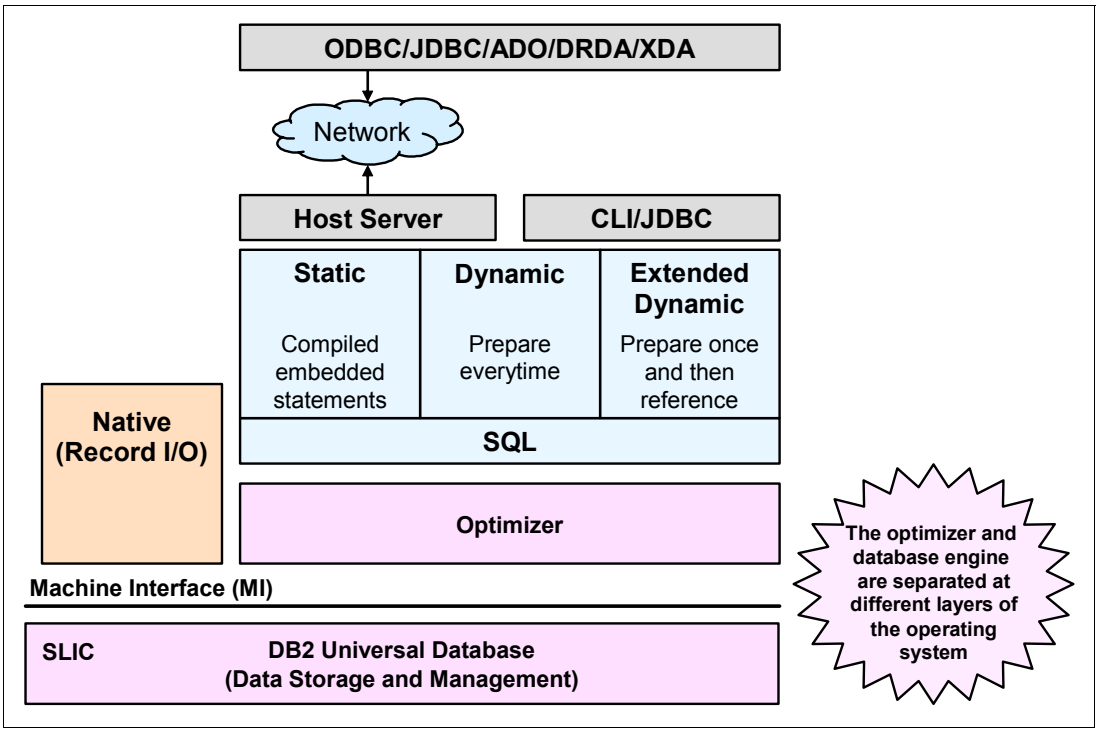


Figure 2-3 Database architecture before the release of V5R2M0: Classic Query Engine

Most CQE query decisions are made above the machine interface (MI) level. In CQE, the interaction between the optimizer and the query execution component occurs across the MI, resulting in interface-related performance overhead.

2.2.2 Current database architecture

With the release of V5R2M0, a new SQE was shipped. SQE and CQE coexist in the same database environment. Depending on the database requests, the Query Dispatcher (see 2.2.3, “Query Dispatcher” on page 17) decides to route the query to either the CQE or SQE.

While both the new SQE and the existing CQE can handle queries from start to finish, the redesigned engine simplifies and speeds up queries. In addition to providing the same functionality as CQE, SQE also performs these functions:

- Moves the optimizer below the MI for more efficient query processing
- Separates and moves improved statistics to the Statistics Manager dashboard
- Uses an object-oriented design that accelerates the delivery of new database functionality
- Uses more flexible, independent data access options to provide autonomous query cruise control
- Uses enhanced algorithms to provide greater responsiveness and query handling
- Provides enhanced performance on long-running complex query terrains
- Retains road maps to provide ease of use in query driving
- Provides additional and enhanced query feedback and debug information messages through the Database Monitor and Visual Explain interfaces

There are several new and updated components of SQE in OS/400 V5R2 and i5/OS V5R3, including:

- Query Dispatcher
- Statistics Manager
- SQE Optimizer
- Data Access Primitives
- Plan Cache

Figure 2-4 shows an overview of the DB2 Universal Database 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.

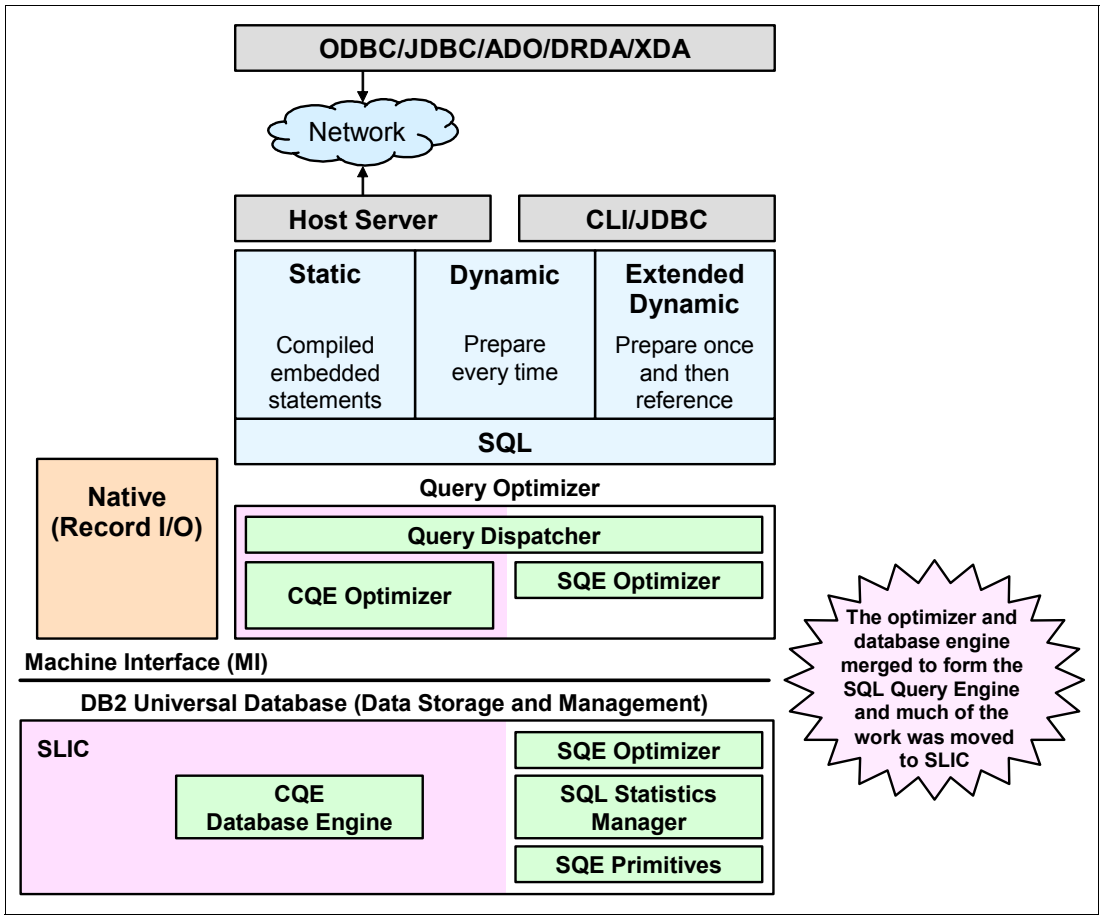


Figure 2-4 Current database architecture: Coexisting CQE and SQE

Note: Most of the SQE Optimizer components are implemented below the MI level, which translates into enhanced performance.

Object-oriented design

The SQE query optimizer was written using an object-oriented design and implementation approach. It uses a tree-based model of the query, where each node is an independent and reusable component. These components can interact and interface with each other in any given order or combination. Each node can be optimized and executed independently. This design allows greater flexibility when creating new methods for query implementation.

With this new design, the ability to use an index in reverse order can be easily implemented by simply adding a new node. The procedural nature of CQE prevents it from being easily enhanced to read an index in reverse order.

Another example to demonstrate how the object-oriented tree model makes SQE easier to enhance is SQE support for *nonsensical* queries. The term nonsensical describes a query statement that does not return any result rows, for example:

```
Select * from testtable where 1 = 0
```

Surprisingly, many applications use this type of query to force no rows to be returned. Because of the procedural nature of CQE, it is virtually impossible to enhance CQE to recognize the fact that 1 will never equal 0. Therefore, CQE implements this query using a table scan. In contrast, the tree node model of SQE easily allows a node to be added to check for nonsensical predicates before reading any rows in the specified table.

Figure 2-5 shows an example of the node-based implementation used in SQE. In this example, NODE1 represents a typical index probe access method. New nodes to check for nonsensical queries and to index in reverse order are added.

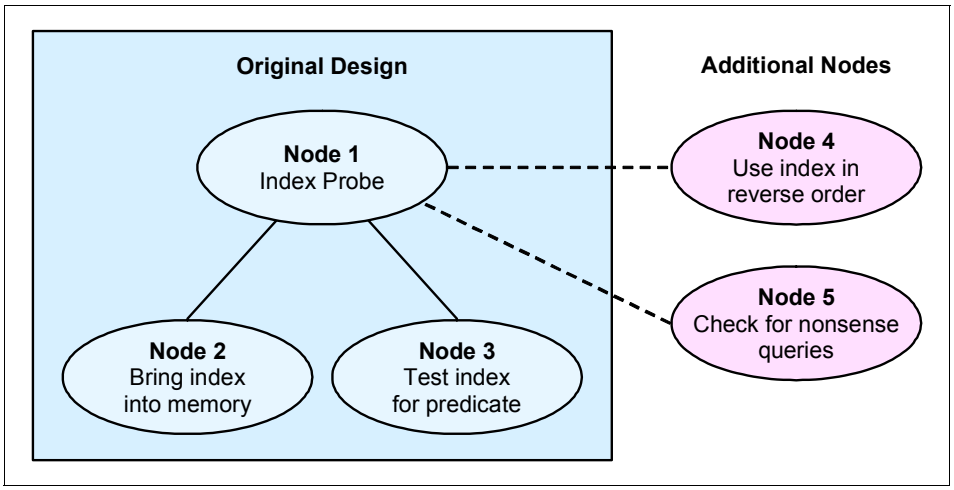


Figure 2-5 Object-oriented design: Tree-based model of queries

2.2.3 Query Dispatcher

The function of the Query Dispatcher is to route the query request to either CQE or SQE, depending on the attributes of the query. One of these attributes includes the interface from which the query originates, which is either SQL-based (embedded SQL, ODBC, or JDBC) or non-SQL based (OPNQRYF and Query/400). All queries, irrespective of the interface used, are therefore processed by the dispatcher. It is not possible for a user or application program to influence this behavior or to bypass the dispatcher.

Note: Only SQL queries are considered for the SQE. OPNQRYF and Query/400 are *not* SQL based.

Figure 2-6 illustrates how different database requests are routed to the different query engines.

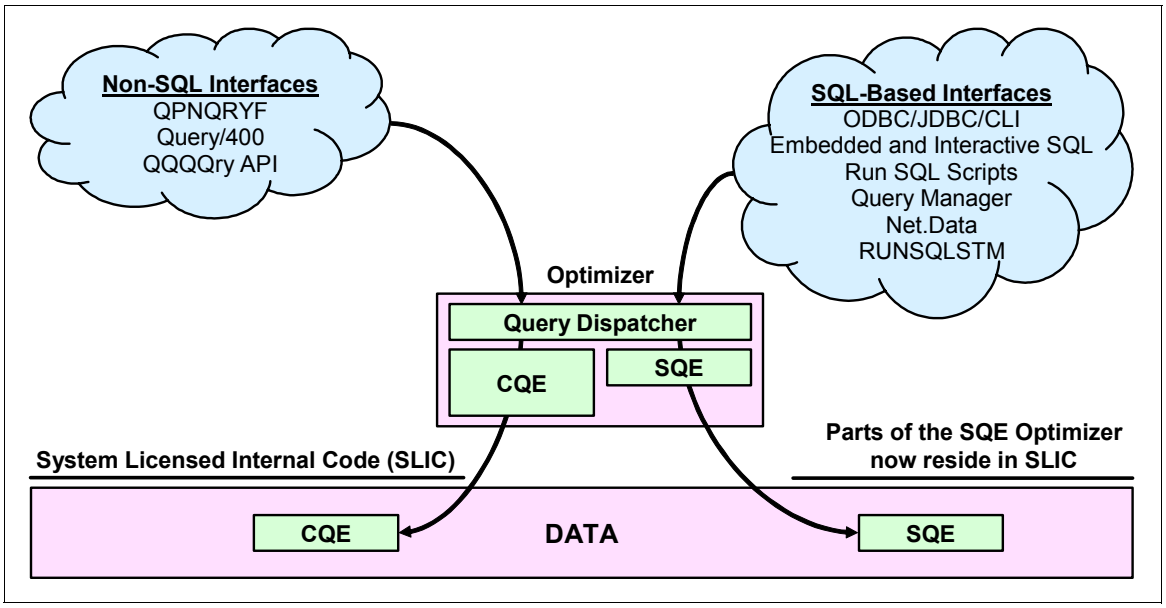


Figure 2-6 Query Dispatcher routing database requests to the query engines

The staged implementation of SQE enabled a limited set of queries to be routed to SQE in V5R2. In general, read-only single table queries with a limited set of attributes were routed to SQE. With the V5R2 enabling PTF applied (PTF SI07650), the dispatcher routes many more queries through SQE. More single table queries and a limited set of multi-table queries can take advantage of the SQE enhancements. Queries with OR and IN predicates might be routed to SQE with the enabling PTF as are SQL queries with the appropriate attributes on systems with symmetric multiprocessing (SMP) enabled.

Note: For more information about PTF SI07650, see Informational APAR II13486 on the Web at:

`http://www-912.ibm.com/n_dir/nas4apar.nsf/042d09dd32beb25d86256c1b004c3f9a/61ffe88d56a943ed86256c9b0041fbeb?OpenDocument`

In i5/OS V5R3, a much larger set of queries is implemented in SQE including those with the enabling PTF on V5R2 and many queries with the following types of attributes:

- Views
- Sub-Selects
- Common Table Expressions
- Derived Tables
- Unions
- Updates
- Deletes
- STAR_JOIN (without FORCE_JOIN_ORDER)

SQL queries that continue to be routed to CQE in i5/OS V5R3 have the following attributes:

- LIKE predicates
- Large Objects (LOB)
- Sensitive Cursor
- NLSS/CCSID translation between columns

- DB2 Multisystem
- ALWCPYDTA(*NO)
- References to logical files
- Tables with select/omit logical files over them

Note: The QAQQINI option IGNORE_ DERIVED_INDEX allows SQE to process the query even when a derived key or select/omit index exists over a table in the query. If allowed to run, SQE ignores the derived and select/ omit indexes.

Derived keys occur when either the Create Logical File (CRTLF) command's data description specification (DDS) specifies keys that are derivations, for example Substring, or if an NLSS sort sequence is active when the CREATE INDEX SQL statement is performed. By default, if one of these indexes exists over a table in the query, SQE is not allowed to process the query.

The dispatcher also has the built-in capability to reroute an SQL query to CQE that was initially routed to SQE. A query typically reverts to CQE from SQE whenever the optimizer processes table objects that define any of the following logical files or indexes:

- Logical files with the SELECT/OMIT DDS keyword specified
- Logical files built over multiple physical file members
- Nonstandard indexes or derived keys, such as logical files specifying the DDS keywords RENAME or Alternate Collating Sequence (ACS) on a field referenced in the key
- Sort sequence NLSS specified for the index or logical file

Note: SQL requests that are passed back to CQE from SQE might experience an overhead of up to 10 to 15% in the query optimization time. However, that overhead is not generated every time that an SQL statement is run. After the access plan is built by CQE, the Query Dispatcher routes the SQL request to CQE on subsequent executions. The overhead appears when the access plan is built the first time or rebuilt by the optimizer.

2.2.4 Statistics Manager

In releases before V5R2, the retrieval of statistics was a function of the CQE 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. Figure 2-7 illustrates how CQE relies on indexes for statistics.

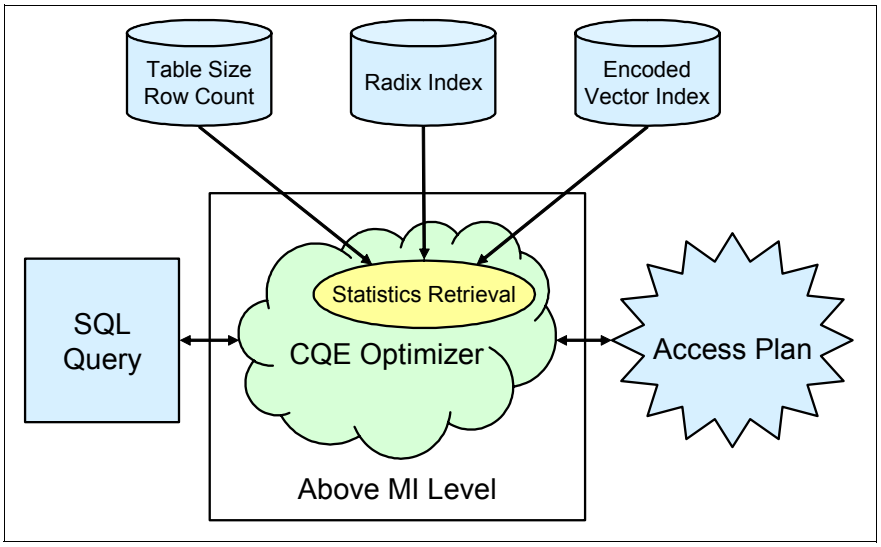


Figure 2-7 CQE Optimizer and Statistics Retrieval

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.

Note: This new statistical information is used only by the SQE. Queries that are dispatched to the CQE do not benefit from available statistics, nor do they trigger the collection of statistics.

Figure 2-8 shows the new design with Statistics Manager.

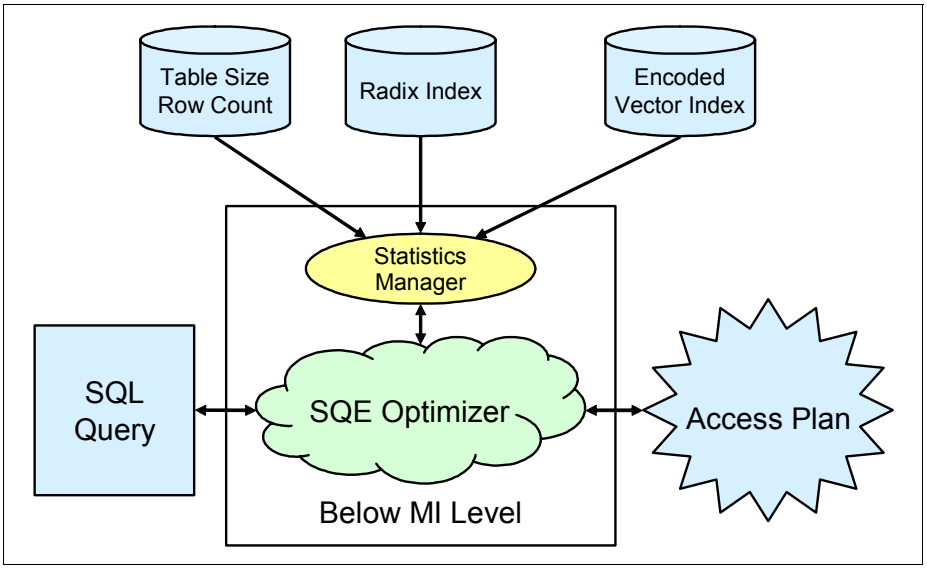


Figure 2-8 SQE Optimizer and Statistics Manager

The Statistics Manager controls the access to the metadata that is required to optimize the query. It uses this information to provide answers to the questions posed by the query optimizer. The Statistics Manager typically gathers and keeps track of the following information:

- Cardinality of values

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

- Selectivity

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

- Frequent values

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

- Metadata information

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

- Estimate of I/O operation

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

You can obtain the majority of this information from existing binary-radix indexes or encoded-vector indexes. An advantage of using indexes is that the information is available to the Statistics Manager as soon as the index is created or maintained.

2.2.5 SQE Optimizer

Like the CQE Optimizer, the SQE Optimizer controls the strategies and algorithms that are used to determine which data access methods should be employed to retrieve the required data. Its purpose is to find the best method to implement a given query.

A fundamental characteristic distinguishes the SQE Optimizer from the CQE Optimizer. The SQE Optimizer gets access to the statistic data collected by the Statistic Manager, by simply asking questions related to the system and the tables used in the query. Based on this information, the access method is determined based on the one that has the least amount of CPU utilization and I/O overhead costs.

Because most of the SQE Optimizer functionality is implemented beneath the MI and consequently closer to the data, the database management system allows for greater flexibility and increased performance.

The CQE Optimizer uses a clock-based timeout algorithm. The CQE Optimizer resequences the indexes, based on the number of matching index columns and the operators used in the WHERE clause of the SQL statement. This approach ensures that the most efficient indexes are optimized first, before the set time limit expired.

In contrary, the amount of time that the SQE Optimizer spends optimizing an access plan is unlimited. A check is done to determine if any indexes exist on the table, with keys built over the columns specified in WHERE or SELECT clauses of the SQL statement. These indexes

are then resequenced so that the most appropriate indexes are processed first and reorganized further based on index-only access, index probe selectivity, total index selectivity, and the size of the index keys.

Note: For SQE, the indexes are ordered in general so 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.

2.2.6 Data Access Primitives

The basic function of SQE Data Access Primitives is to implement the query. Using the data access methods derived from the object-oriented, tree-based architecture, Data Access Primitives provide the implementation plan of the query.

2.2.7 Access plan

The methods used for a specific SQL statement to get access to the data are stored in access plans. If the access plan does not exist, it is created the first time that an SQL statement is executed. If the access plan already exists, it is compared with the information provided by the Statistics Manager (SQE) or by the query optimizer in CQE. If the optimizer decides to use an other access path, the access plan is updated.

In contrary to CQE, the access plans that are created and used with the SQE are organized in a tree-based structure to allow for maximum flexibility.

If you use SQL statements in programs, there are different ways to embed, prepare, and execute your SQL statements. These different methods affect the creation time of the access plan for the specified SQL statements. All executable SQL statements must be prepared before they can be executed. The result of preparation is the executable or operational form of the statement. We can differentiate between the three methods:

- Static SQL

  In static SQL, the SQL statements that must be executed are already known at compile time. The precompiler checks the syntax and converts the SQL statement into an executable form, as well as creates an access plan that is embedded into the program object. If the access plan is changed because of an altered data pool or new indexes, the access plan is updated in the program object. In this way, a program object can grow over the time, even if no modifications are performed.

- Dynamic SQL

  Programs that contain embedded dynamic SQL statements must be precompiled like those that contain static SQL. Unlike static SQL, the dynamic SQL statements are checked, constructed, and prepared at run time. The source form of the statement is a character or graphic string that is passed to the database manager by the program that is using the static SQL PREPARE or EXECUTE IMMEDIATE statement. The operational form of the statement persists for the duration of the connection or until the last SQL program leaves the call stack. Access plans associated with dynamic SQL might not persist after a database connection or job is ended.

- Extended dynamic SQL

  An extended dynamic SQL statement is neither fully static nor fully dynamic. The Process Extended Dynamic SQL (QSQPRCED) API provides users with extended dynamic SQL capability. Like dynamic SQL, statements can be prepared, described, and executed using this API. Unlike dynamic SQL, SQL statements prepared into a package by this API persist until the package or statement is explicitly dropped.

The iSeries Access ODBC driver and JDBC driver both have extended dynamic SQL options available. They interface with the QSQPRCED API on behalf of the application program.

SQL packages

SQL packages are permanent objects with the object type *SQLPKG used to store information related to prepared, extended dynamic SQL statements. They can be used by the iSeries Access for Windows® ODBC driver and the IBM Toolbox for Java JDBC driver. They are also used by applications which use the QSQPRCED API interface.

The SQL package contains all the necessary information to execute the prepared statement. This includes registry of the statement name, the statement text, the internal parse tree for the statement, definitions of all the tables and fields involved in the statement, and the query access plan needed to access the tables at run time.

Note: When using embedded SQL, no separate SQL package is created, but the access plan is integrated into the program or service program object.

Creation time of SQL packages

In the case of ODBC and JDBC, the existence of the package is checked when the client application issues the first prepare of an SQL statement. If the package does not exist, it is created at that time, even though it might not yet contain any SQL statements. In the case of QSQPRCED, creation of the package occurs when the application calls QSQPRCED specifying function 1.

Advantages of SQL packages

Because SQL packages are a shared resource, the information built when a statement is prepared is available to all the users of the package. This saves processing time, especially in an environment when many users are using the same or similar statements. Because SQL packages are permanent, this information is also saved across job initiation or termination and across initial program loads (IPLs). In fact, SQL packages can be saved and restored on other systems. By comparison, dynamic SQL requires that each user go through the prepare processing for a particular statement and do this every time the user starts the application.

SQL packages also allow the system to accumulate statistical information about the SQL statements. Accumulating such information results in better decisions about how long to keep cursors open internally and how to best process the data needed for the query. As indicated previously, this information is shared across users and retained for future use. In the case of dynamic SQL, every job and every user must relearn this information.

Deletion of SQL packages

Packages must be deleted when the underlying metadata for statements stored in the package has been changed. If a table, view, procedure, or other SQL object is altered, the information in the package is not updated. If the package is not deleted, a variety of unusual errors can occur, including truncation, data mapping errors, incorrect describe information, and so on.

Delete packages whenever significant changes (those that might cause a large amount of access plan rebuilds) are made to the database, operating system, or hardware. Because extended dynamic SQL packages are recreated when the application is run, there is little harm in deleting them.

Plan Cache

The Plan Cache is a repository that contains query implementation plans for queries optimized by the SQE Optimizer. Query access plans generated by CQE are not stored in the Plan Cache. The architecture of DB2 Universal Database for iSeries allows for only one Plan Cache per iSeries server or logical partition (LPAR).

The purpose of the Plan Cache is to facilitate the reuse of a query access plan at some future stage when the same query, or a similar query, is executed. After an access plan is created, it is available for use by all users and all queries, irrespective of the interface from which the query originates.

CQE already uses plan caches, but queries from different interfaces each go to their own Plan Cache. Furthermore, when an access plan is tuned, for example when creating an index, all queries can benefit from this updated access plan. The update plan eliminates the need to reoptimize the query and results in greater efficiency and faster processing time. In the case of CQE, each query has to update its own access plan to benefit from the newly created index.

Before optimizing an incoming query, the optimizer looks for the query in the plan cache. If an equivalent query is found, and the associated query plan is found to be compatible with the current environment, the already-optimized plan is used, avoiding full optimization.

Figure 2-9 shows the concept of reusability of the query access plans stored in the Plan Cache. The Plan Cache is interrogated each time a query is executed using the SQE.

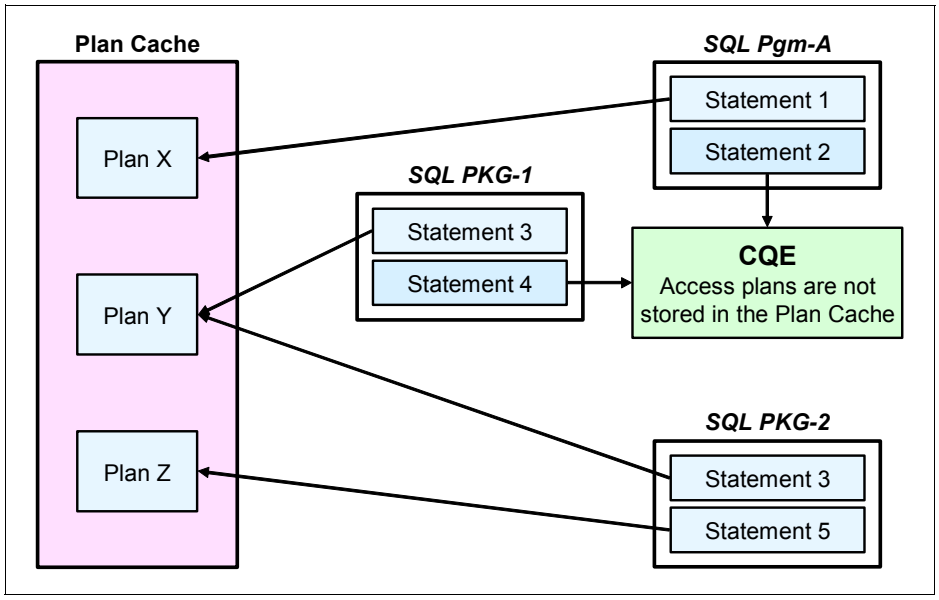


Figure 2-9 Plan Cache

Note: Access plans generated by CQE are *not* stored in the SQE Plan Cache.

In addition, unlike CQE, SQE can save multiple different plans for the same query. This method is useful in more dynamic environments where the plan changes depending on user inputs, available memory, and so on.

Note: The Plan Cache is cleared during an IPL or varying the independent auxiliary storage pool (IASP).

For more information about SQE and CQE, see the IBM Redbook *Preparing for and Tuning the V5R2 SQL Query Engine on DB2 Universal Database for iSeries*, SG24-6598, and *DB2 Universal Database for iSeries Database Performance and Query Optimization*, which is available in the iSeries Information Center on the Web at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzajq/rzajqmst.htm`

2.3 Star Join Schema

A *star schema* is a database model characterized by a large centralized table, called the *fact table*, surrounded by other highly normalized tables called *dimension tables*. A *Star Join Schema query* is a query over multiple tables of a star schema database with local selection specified on the dimension or dimensions and equi-join predicates between the fact table and the relevant dimension table or tables.

The attributes of a star schema database model usually include:

- A relatively large fact table that contains millions or billions of rows holding the measurable or additive “facts” such as sales type transactions or events
- Relatively small and highly normalized dimension tables that contain descriptive data about the “facts” (in the central fact table), such as customer or location information
- A central fact table that depends on the surrounding dimension tables using a parent/child relationship, with the fact table as the child and the dimension tables as the parent

If the dimension tables are further normalized, the results are dimensions that might have additional tables that support them. This is known as a “snowflake” schema or model.

Figure 2-10 shows the structure of a typical Star Join Schema.

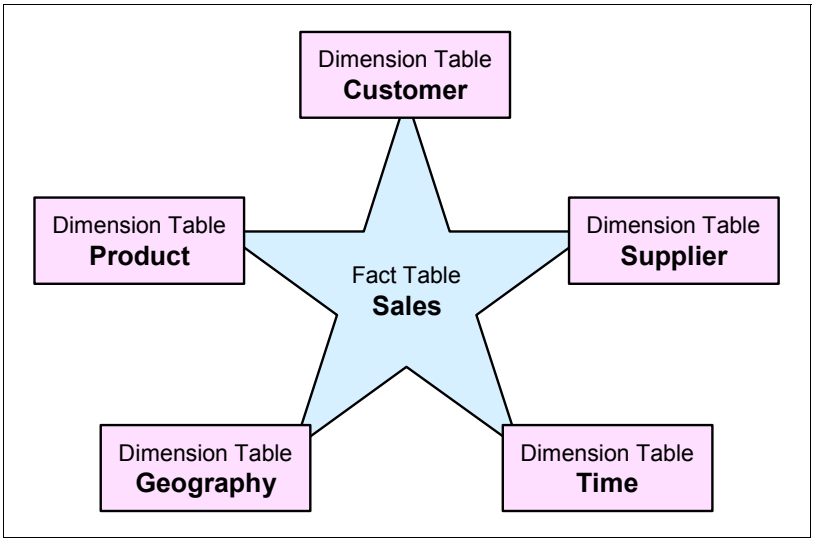


Figure 2-10 Star Join Schema

Starting with i5/OS V5R3, the SQE automatically supports optimizing Star Join Schema queries. This new support does not require the use of QAQQINI options. Recognizing and optimizing a star schema query is now a normal part of handling any query request.

Support is provided with DB2 Universal Database for iSeries CQE in OS/400 V5R2 and with CQE and SQE in i5/OS V5R3. To enable Star Join Schema support via QAQQINI options file, perform one of the following operations prior to running the query:

```
INSERT INTO library/QAQQINI VALUES('STAR_JOIN', '*COST', NULL);

INSERT INTO library/QAQQINI VALUES('STAR_JOIN', '*FORCE', NULL);
```

In the first operation, *COST is the recommended and preferred parameter value, since each EVI is considered based on its potential to further minimize the I/O. In the second operation, *FORCE is more aggressive, using a larger number of bitmaps with the CQE skip sequential access method, since all of the relevant EVIs are used for local selection, regardless of cost.

With SQE, the QAQQINI option STAR_JOIN is ignored. However, because a few queries might still be routed to CQE, users who have specified STAR_JOIN *COST in the past should continue to do so. That way CQE will not suffer and SQE will be unaffected. In addition, in the past, the FORCE_JOIN_ORDER option was specified in conjunction with the STAR_JOIN option. This option is no longer required for CQE and should be removed by customers who use it for star join queries. If this option is left in, the FORCE_ JOIN_ORDER is still honored by both CQE and SQE and might prevent the optimizer from generating optimal plans.

Note: Single key EVIs created over the foreign key columns of the fact table are required for the CQE Optimizer to implement the star join techniques. These same EVIs are optimal for the new SQE Optimizer to implement its Star Join Schema techniques.

2.3.1 Queries in a Star Join Schema

Queries in a Star Join Schema typically include the following characteristics:

- Multiple tables participating in the query
- Local selection predicates on the dimension tables rather than the larger fact table
- Equi-join predicates between the dimension tables and the fact table used to locate and select the relevant fact table rows and to decode and describe the fact table data

  The equi-join predicate between any one dimension table and the fact table might result in a very large number of fact table rows being selecting, while the intersection of the equi-join predicates of multiple dimension tables might result in a relatively small number of fact table rows being selected.

To achieve the optimal performance in joining the tables, you must consider these rules:

- For CQE, the join order of the query must be Fact_Table as the primary table, followed by the dimension tables. CQE identifies the fact table by determining the largest table in the query and that table is “pinned” in join position 1. Then the order of dimension tables (2 through n) is optimized.
- SQE also identifies the largest table in the query (that is, the fact table) but optimizes the join order of all the tables based on new strategies and methods. This might result in the fact table being placed somewhere other than the first join position. Since SQE can use new and different methods, it is advantageous to also create single key radix indexes on the foreign key columns of the fact table. This index is optimal if the fact table is being joined from a dimension table in join position 1.

2.3.2 Restrictions and considerations

When working with Star Join Schema support, consider these restrictions:

- For both CQE and SQE, the only sort sequence supported is *HEX. If the query or job specifies a sort sequence other than *HEX, the Star Join Schema support is not used.
- Single key column EVIs must be created over the join columns of the fact table. When using the star schema support, the CQE Optimizer does not use radix indexes to create the dynamic bitmaps for the skip sequential processing on the fact table. When a large range of key values are selected, using EVIs to create the dynamic bitmaps is more efficient and faster.
- Specifying the QAQQINI parameter STAR_JOIN within an OS/400 V5R2 database environment causes SQL query requests to use CQE instead of SQE. In V5R2, SQE is not specifically enhanced for Star Join Schema queries.

  In V5R3, SQE includes specific enhancements for optimizing and executing Star Join Schema queries, so the STAR_JOIN parameter is ignored by SQE. In V5R3, CQE continues to honor the STAR_JOIN parameter for those query requests that are optimized and executed by CQE. For example, a query that contains the LIKE operator does continue to be optimized and executed by CQE instead of SQE. To gain the benefit of the CQE star join support, the QAQQINI STAR_JOIN parameter is needed.

2.3.3 Lookahead Predicate Generation

The key feature of the new SQE support is referred to as *Lookahead Predicate Generation* (LPG). While the SQE LPG support looks similar to the older CQE support, it has some additional benefits and uses.

When a query in a Star Join Schema is optimized, a hash join is normally chosen as the method to join the fact table to the dimension tables. To accomplish this, the original query is broken into multiple parts or steps. For each dimension table, an internal query is executed to access the rows matching the local selection predicates, using the best available access method (scan or probe, with or without parallelism). The data required for the query is then used to build a hash table.

In addition to building the hash tables, the join key values of the selected dimension table rows are used to populate a list of distinct keys. This list represents all the join key values (that match the corresponding local selection) and is used to identify the join column values in the fact table.

After the hash table or tables and distinct key list or lists are built, the original query is rewritten. The distinct key list or lists are used to provide local selection on the fact table. In effect, it transfers the local selection from the dimension table or tables to the fact table. This transfer is referred to as *Lookahead Predicate Generation*.

For more information about the Star Join Schema and LPG, see the white paper *Star Schema Join Support within DB2 UDB for iSeries - Version 3* on the Web at:

`http://www-1.ibm.com/servers/enable/site/education/abstracts/16fa_abs.html`

For more information about the SQE, see the IBM Redbook *Preparing for and Tuning the V5R2 SQL Query Engine on DB2 Universal Database for iSeries*, SG24-6598.

Part 2

Gathering, analyzing, and querying database performance data

In this part, we describe and discuss the different ways to gather database performance data. After we explain how to gather this data, we describe the different ways to analyze it with the tools that DB2 Universal Database for iSeries has. Later in the part, we show you how to query the Database performance data and how to tie this data to one of the preferred tools called *Visual Explain*.

This part contains the following chapters:

3

Overview of tools to analyze database performance

Database performance is a high priority in any system. The objective is to maximize system resource utilization, while achieving maximum performance throughput. Therefore, analyzing your queries is the most important step to ensure that they are tuned for optimal performance.

You must select the proper tools for collecting and analyzing the database performance data first, to ensure that your queries are tuned for optimal performance. By using the following analysis tools to obtain an information through the monitoring processes, you should be able to take the appropriate corrective actions.

- Current SQL for a Job (iSeries Navigator)
- Print SQL information
- Debug messages
- Index Advisor
- Index Evaluator
- SQL Performance Monitor (Detailed Monitor)
- Memory-based Database Monitor (Summary Monitor)
- Visual Explain

In this chapter, we introduce and provide information about the tools for monitoring and analyzing the database performance data of your SQL queries.

3.1 Current SQL for a Job function in iSeries Navigator

You can use the Current SQL for a Job function to select any job running on the system and display the current SQL statement being run, if any. In addition to displaying the last SQL statement being run, you can edit and rerun it through the Run SQL Script option (linked automatically) and display the actual job log for the selected job or, even end the job. You can also use this function for database usage and performance analysis with the Visual Explain tool as explained in Chapter 8, “Analyzing database performance data with Visual Explain” on page 197.

To start the Current SQL for a Job function, in iSeries Navigator, in the left pane, right-click **Databases** and select **Current SQL for a Job** as shown in Figure 3-1.

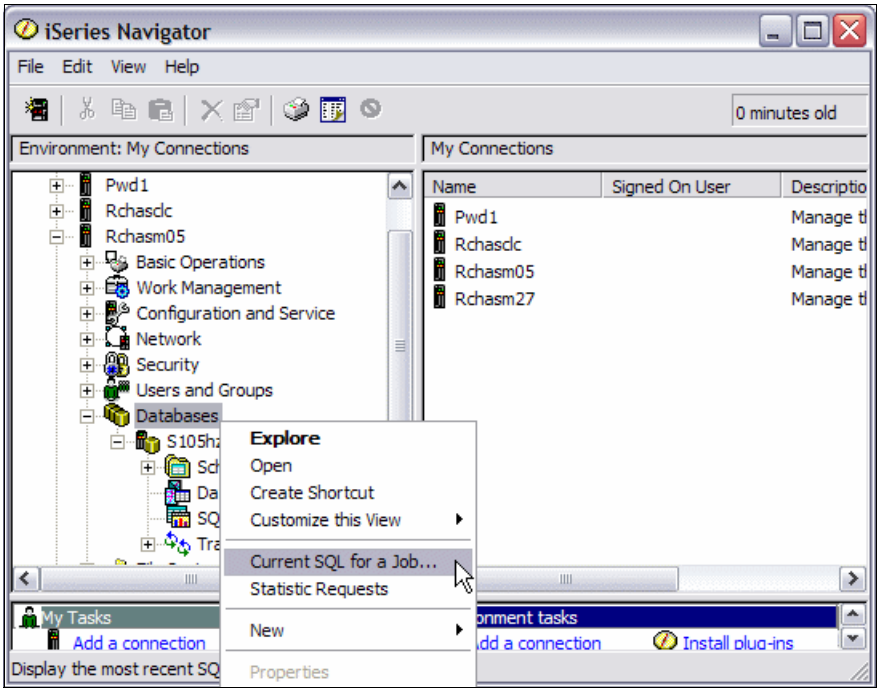


Figure 3-1 Selecting the Current SQL for a Job function in iSeries Navigator

Then the Current SQL window (Figure 3-2) opens. This window displays the name, user, job number, job subsystem, and current user for the available jobs on your system. You can select a job and display its job log, the SQL statement currently being run (if any), decide to reuse this statement in the Run SQL Script Center, or end the job, provided that you have sufficient authority.

Note: The line “`Last statement to finish as of 4:33:18 PM`” is the time that the SQL statement button was selected in iSeries Navigator. This is *not* an indication of when the SQL ran. The SQL statement displayed might be the current SQL the job is running or the last SQL statement ran in the job (7).

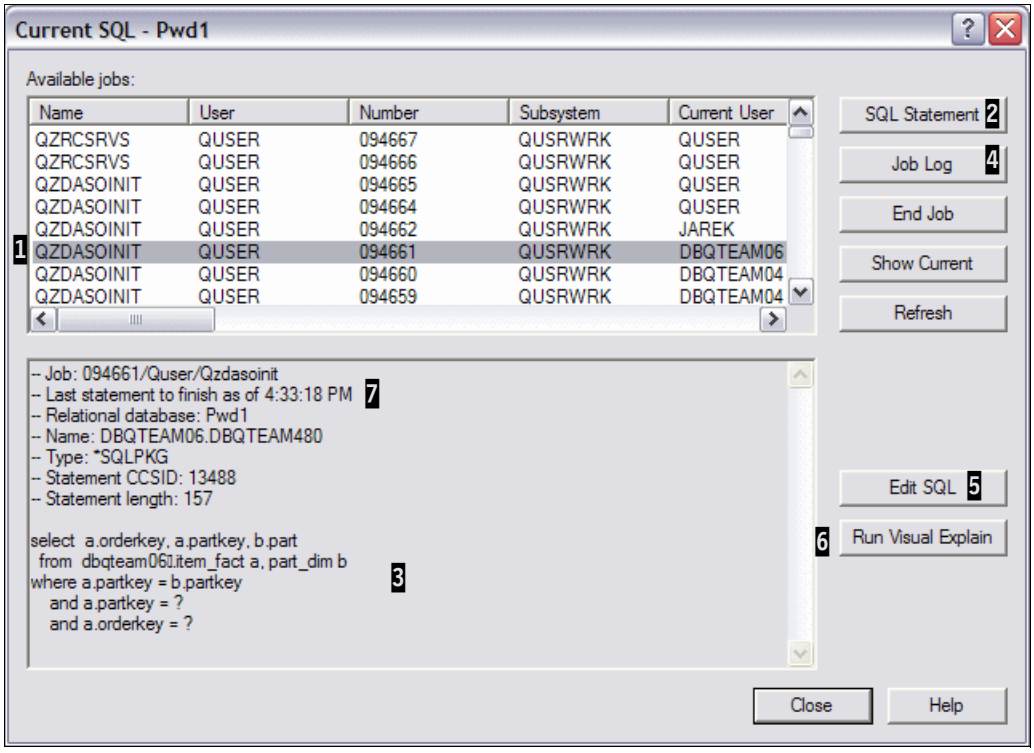


Figure 3-2 Current SQL for a job

In our example, we selected a Java Database Connectivity (JDBC) job (1) and clicked the SQL Statement button (2) to view the last SQL statement that it ran, in the bottom part of the panel (3). To go to its job log, we can click the Job Log button (4). After the SQL statement is displayed in the bottom part of the panel, we can click the Edit SQL button (5) to work on this same statement with the Run SQL Script center as shown in Figure 3-3.

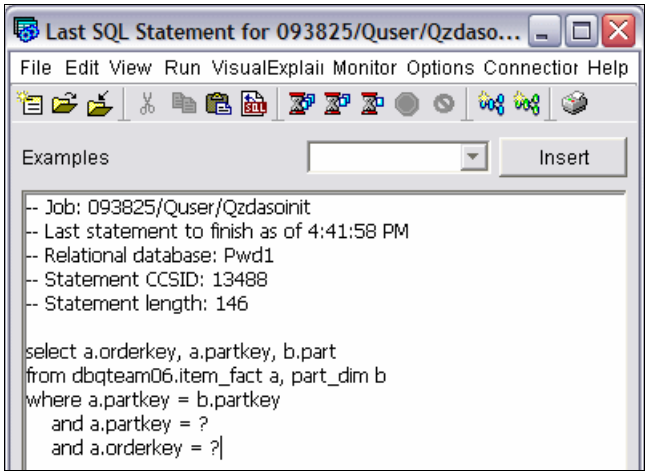


Figure 3-3 Run SQL Script with Current SQL for a job

We can click the Run Visual Explain button (6) to perform direct analysis of the SQL statement as shown in Figure 3-4.

Refer to Chapter 8, “Analyzing database performance data with Visual Explain” on page 197, for a more detailed explanation of how to analyze a query and use the functionality in Visual Explain within iSeries Navigator.

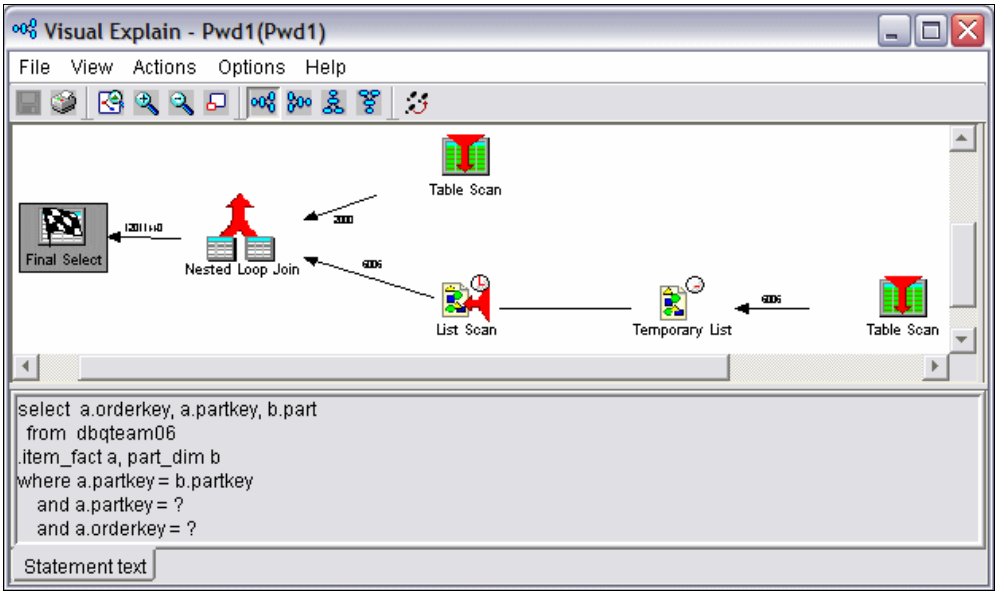


Figure 3-4 Running Visual Explain in Current SQL for a job

3.2 Print SQL information

The information contained in SQL packages, service programs, and embedded SQL statements can also assist in identifying potential performance problems in your queries.

To view the information pertaining to the implementation and execution of your query, select an SQL package from iSeries Navigator. Right-click the **SQL package name** and select **Explain SQL**, as shown in Figure 3-5.

This is equivalent to using the Print SQL Information (PRTSQLINF) CL command that extracts the optimizer access method information from the object and places that information in a spool file. The spool file contents can then be analyzed to determine if any changes are needed to improve performance.

The information in the SQL package is comparable to the debug messages discussed in 3.3, “Debug messages” on page 36. However, there is more detail in the first level SQLxxxx messages.

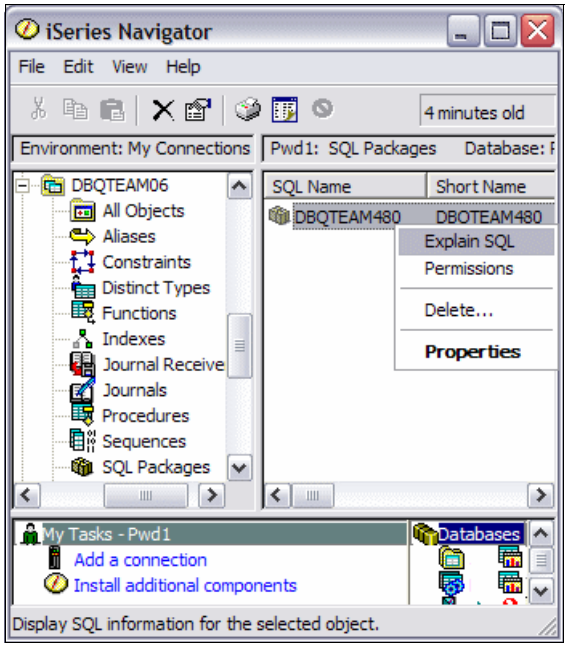


Figure 3-5 Selecting to print SQL information

Figure 3-6 shows an example of the SQL package information that can be displayed. As you can see in the example, the information pertaining to the join order of the tables, the access methods used in the implementation of the query, and runtime statistics are available. Notice the messages that indicate which QAQQINI query options file was used for executing the query and whether this query implementation used symmetric multiprocessing (SMP).

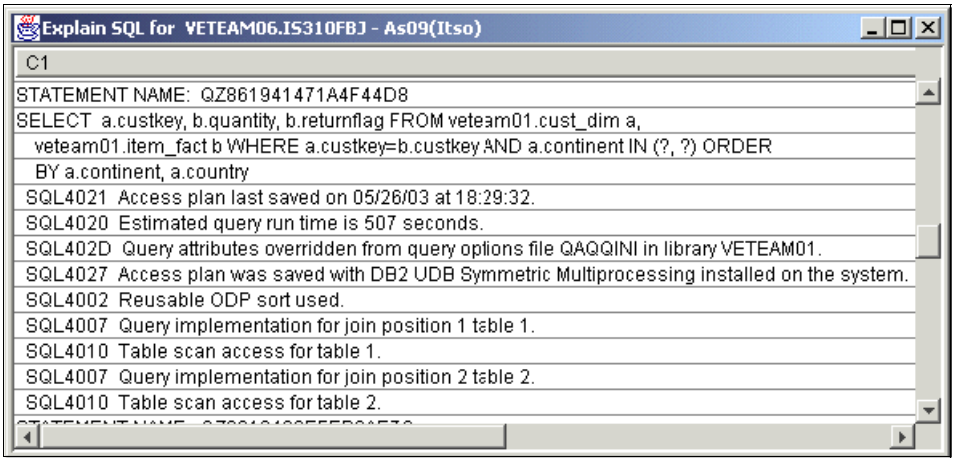


Figure 3-6 Viewing SQL package information

You can also obtain this information by using the following PRTSQLINF CL command:

```
PRTSQLINF OBJ(library_name/program_name or package_name) OBJTYPE(*SQLPKG)
```

Note: If you are unsure of the program or package name, look at the program name above the QSQ* module in the call stack when you use option 11 (Display call stack, if active) of the Work with Active Jobs (WRKACTJOB).

The PRTSQLINF CL command directs the output data to a spooled file, from where you can display or print the information.

Note: The information retrieved from an SQL package might not accurately reflect the access plan used by the last execution of the SQL query. For example, if circumstances at execution time cause the query to be reoptimized and the package or program is locked, then a new access plan is dynamically generated and placed in the Plan Cache (for SQL Query Engine (SQE) use). The version stored in the package or program is not updated.

3.3 Debug messages

Analyzing debug messages is another important tool for monitoring and tuning queries. When doing so, keep in mind the following points:

- The debug messages are no longer being enhanced (that is, no new messages are being added for queries that go through SQE).
- It is hard to tie a message to an SQL statement.
- It is difficult to search through all of the job log messages.

There are multiple methods of directing the system to generate debug messages while executing your SQL statements such as:

- Selecting the option in the Run SQL Scripts interface of iSeries Navigator
- Using the Start Debug (STRDBG) CL command
- Setting the QAQQINI table parameter
- Using Visual Explain

You can choose to write only the debug messages for one particular job to its job log. If you want to use iSeries Navigator to generate debug messages, in the Run SQL Scripts window, click **Options** → **Include Debug Messages in Job Log** as shown in Figure 3-7.

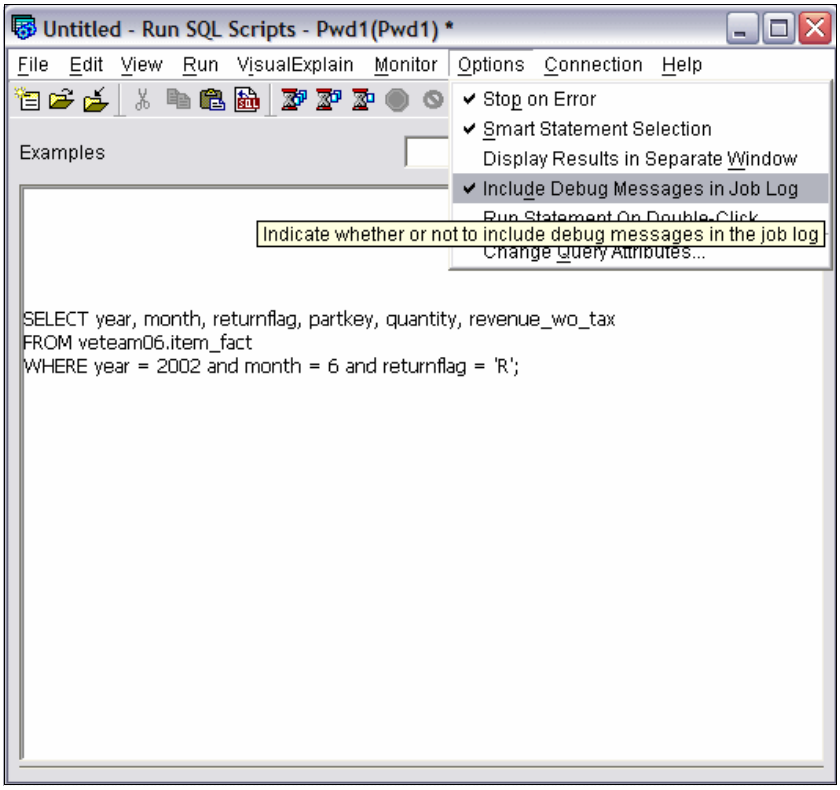


Figure 3-7 Enabling debug messages for a single job

After you run your query, in the Run SQL Scripts window, select **View** → **Joblog** to view the debug messages in the job log. In our example, we used the SQL statement shown in Example 3-1.

Example 3-1 Example SQL statement

```
SELECT year, month, returnflag, partkey, quantity, revenue_wo_tax
  FROM veteam06.item_fact
 WHERE year = 2002 and month = 6 and returnflag = 'R';
```

The detailed job log describes information that you can use to identify and analyze potential problem areas in your query such as:

- Indexes
- File join order
- Temporary result
- Access plans
- Open data paths (ODPs)

All of this information is written to the job log when under debug using the STRDBG command.

Figure 3-8 shows an example of the debug messages contained in the job log after you run the previous query.

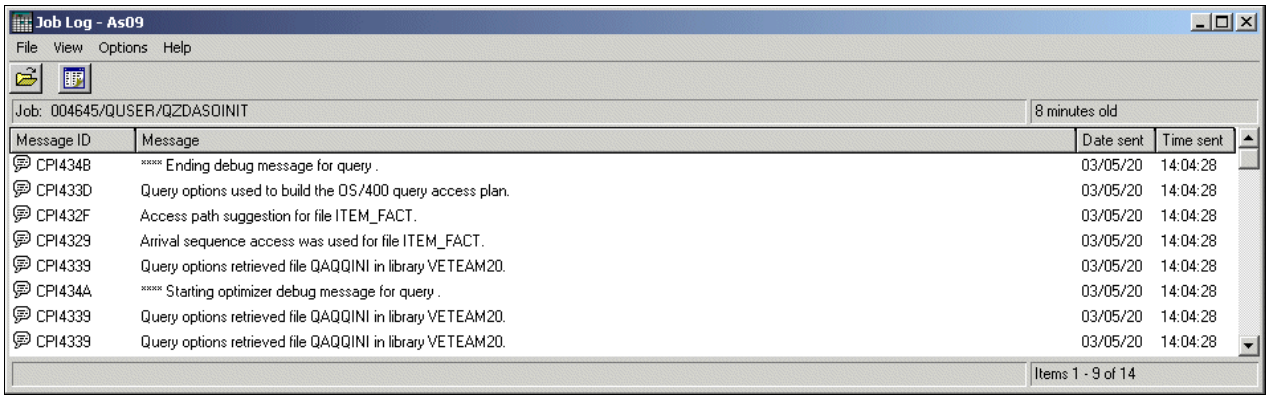


Figure 3-8 Job log debug messages

After you enable these settings for a particular job, only debug messages relating to queries running in that job are written to the job log. You see the same debug messages with this option as those explained later when the QAQQINI parameter MESSAGES_DEBUG is set to *YES. You also see additional SQL messages, such as "`SQL7913 - ODP deleted` and `SQL7959 - Cursor CRSRxxxx closed`", in the job log.

Select any of the debug messages displayed in the job log. Click **File** → **Details** to obtain more detailed information about the debug message. Figure 3-9 shows an example of a detailed debug message that is displayed.

By looking at the messages in the job log and reviewing the second-level text behind the messages, you can identify changes that might improve the performance of the query such as:

- Why index was or was not used
- Why a temporary result was required
- Join order of the file
- Index advised by the optimizer

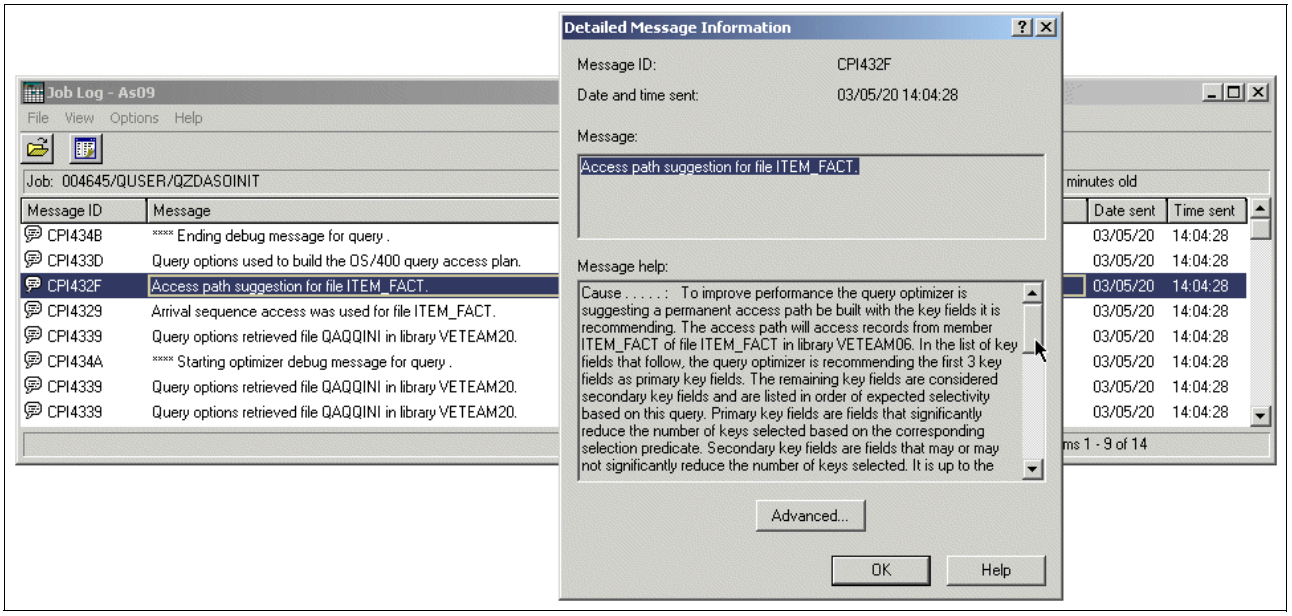


Figure 3-9 Detailed debug message information

When running SQL interactively, either through a 5250-session or via the Run SQL Scripts window in iSeries Navigator, you can also use the STRDBG CL command to generate debug messages in your job log.

Remember: You also must run the Start Server Job (STRSRVJOB) CL command if your query runs as a batch job.

By setting the value of the QAQQINI parameter MESSAGES_DEBUG to *YES, you can direct the system to write detailed information about the execution of your queries into the job's job log.

To activate this setting through the Run SQL Scripts interface of iSeries Navigator, select the **QAQQINI table** that you want to use as shown in Figure 3-10. Then, select the **MESSAGES_DEBUG** parameter and change the parameter value as shown in Figure 3-10. After you make the appropriate change, close the window, and in the Change Query Attributes window, click **OK** to save your changes.

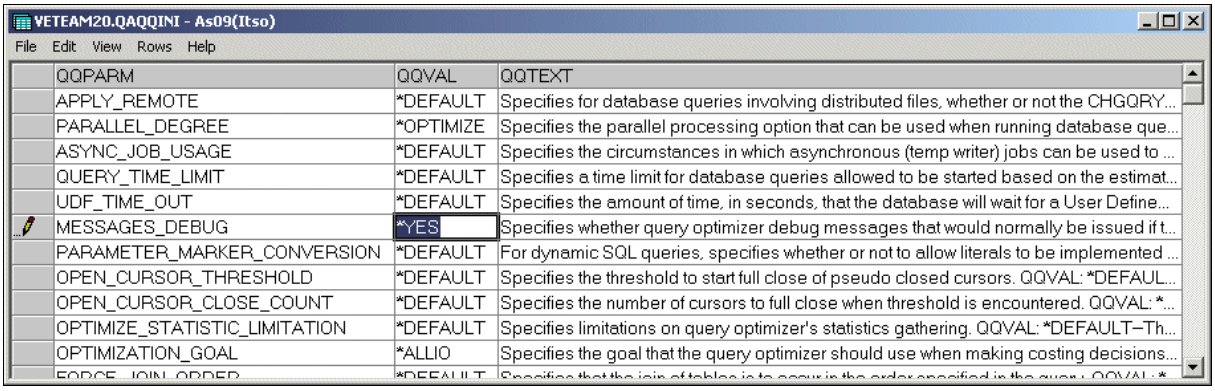


Figure 3-10 Enabling debug messages in QAQQINI

Important: Changes made to the QAQQINI table are effective immediately. They affect all users and queries that use this table. For example, if you set the MESSAGES_DEBUG parameter to *YES in a particular QAQQINI table, all queries that use that QAQQINI table write debug messages to their respective job logs.

The analysis of optimizer debug messages was made easier with the addition of a *Predictive Query Governor*. By specifying a time limit of zero in the Predictive Query Governor, query optimizer debug messages can be generated in the job log without running the query. The query time limit is checked against estimated query time before initiating your query since the optimization cost and access plan are determined prior to execution in cost-based optimization.

The time limit is set on a per-job basis such as:

- The QRYTIMLMT parameter on the CHGQRYA CL command
- The QUERY_TIME_LIMIT parameter in the QAQQINI file
- The QQRYTIMLMT system value (CHGSYSVAL QQRYTIMLMT)

That is, you can analyze a query, which might take 16 hours to run, in only a few seconds. Some changes can be made to the query or to the database. The effect can be modeled on the query in a few minutes. The query is then run when the optimum implementation is achieved.

One of the most important debug messages to look for is advise about the creation of indexes, since the objective of creating indexes is to improve the performance of your queries. The query optimizer analyzes the record selection in the query and determines, based on the default estimate, whether the creation of an index can improve performance. If a permanent index is beneficial, it returns the key fields necessary to create the suggested index. You can find Index Advisor information in the debug message CPI432F. This takes us to the next tool, which is the Index Advisor.

3.4 Index Advisor

In V5R3, the Index Advisor assists you more in suggesting indexes because the index advised code was improved to recommend more useful indexes. To take advantage of this enhancement, you are required to apply program temporary fix (PTF) MF34412.

Note: The improved Index Advisor is only for SQL requests that are routed to the SQE.

Index Advisor offers the following improvements:

- Advanced Radix Index advice
  - It provides advice for a more optimal index, even when a suboptimal index exists and is potentially used. For example, the query has two predicates, but an index exists on only one of the predicates.

    Note: This advanced function can increase the possibility of recommending an unnecessary index where a suboptimal index is sufficient. Therefore, you must analyze the recommendations that the advisor makes.

  - It improves the handling of advice regarding join predicate, grouping, ordering, and distinct clauses. A more complex combination of the record selection criteria can be advised.
  - Advise is based on a high level view of the query rather than the implementation chosen.
- Advice for an encoded-vector index (EVI)
  - EVI recommendations are made for certain grouping queries.
  - Recommendations for the EVI are made when Lookahead Predicate Generation (LPG) predicates are generated. For more information about LPG predicates, refer to 2.3.3, "Lookahead Predicate Generation" on page 27.

In the past, you might have received the CPI432F message, which advised the use of an index when an index already existed in key columns. Now the SQE Optimizer does not advise an index if one already exists, even when the existing index differs from the advised index. This occurs as a result of the one of the following reasons:

- The keys of the existing index match the keys of the advised index.
- The keys of the existing index are in a different, but acceptable, order than the advised index.

Note: There are times when the keys have to be in the same order, such as in sorting, so the SQE Optimizer advises an index if the existing one doesn't match the correct order.

- The order of the keys in the existing index are opposite of the advised index. For example, the existing index is in descending order, and the advised index wants the keys in ascending order.

3.5 Index Evaluator

Prior to V5R3, there was no easy way to determine if there were any unnecessary indexes over a physical file or table. No information specified whether an index was used by a query or when an index was used to give statistics to the query optimizer. The only information available was from the Last Used Date. Unfortunately, customers erroneously used the date to determine whether an index was used recently, and if it wasn't used, they deleted the index.

After the indexes were deleted, performance problems resulted because the optimizer no longer had the necessary indexes over the table for the query optimizer to use to make a good decision. In V5R3 iSeries Navigator and V5R3 i5/OS, the Index Evaluator was added to give statistics about index usage.

Note: To activate this feature, apply the PTFs for APAR SE14709, iSeries Access Service Pack 2 for V5R3, and PTF SI14782.

The new statistics fields support the following characteristics:

- There are two counters, one for when the index is used in the query implementation and one for when the index was used to gather statistics.
- Both counters are set by the two query engines, SQE and Classic Query Engine (CQE). The statistic fields are updated regardless of the query interface, such as Query/400, SQL/400®, OPNQRYF, or QQQQRY API, that is used.
- The statistics survive IPLs.

- Save/Restore does not reset the statistics on an index if an index is restored over an existing index.

  Note: If an index is restored that does not exist on the system, the statistics are reset.

- The statistics start counting after the PTFs are applied and active.
- For each counter, there is a corresponding time stamp to go with it from the last time the counter was bumped.
- The data is stored internally in the index object. At this time, there is no way to query this; to determine usage of an index over a specific time frame, look at each time stamp on each individual index.

The new statistics are query statistics only. Native RPG, COBOL, and similar OPEN operations that are *not* queries are not covered by the new statistic fields. However, we have had native OPEN usage statistics for many years; if a user wants to determine whether the index is used via these nonquery interfaces, look at the existing statistics via fields such as Object Last Used.

Note: You can also use the QUSRMBRD API to return statistics.

To find this information, in iSeries Navigator, right-click the desired table and then click **INDEXES**. The additional information is displayed as shown in Figure 3-11.

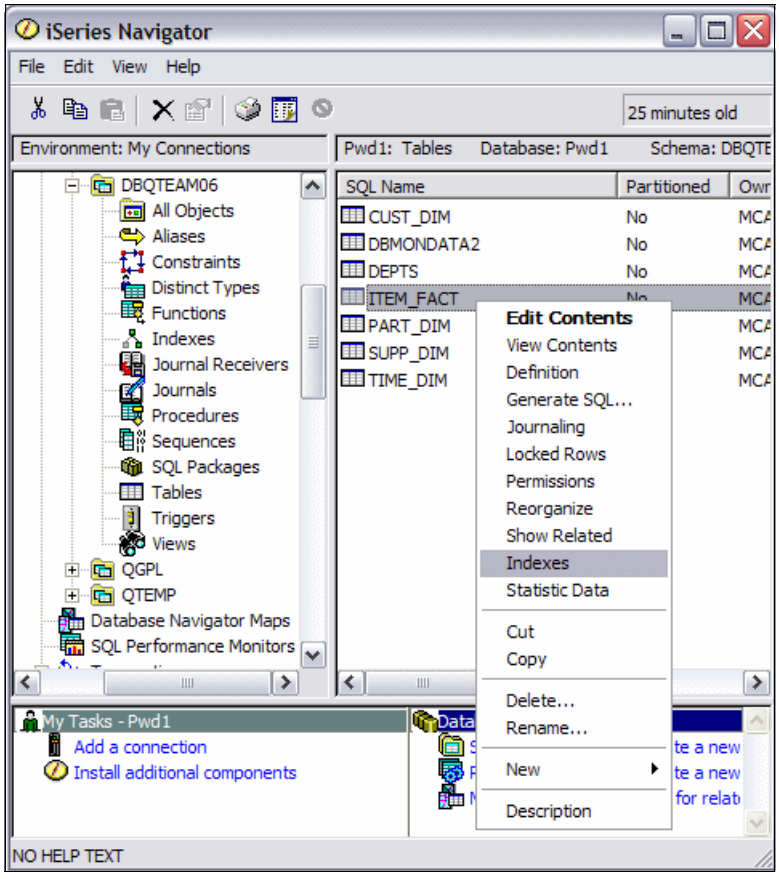


Figure 3-11 Getting index information in iSeries Navigator

In this example, you can see two indexes in the ITEM_FACT table as shown in Figure 3-12.

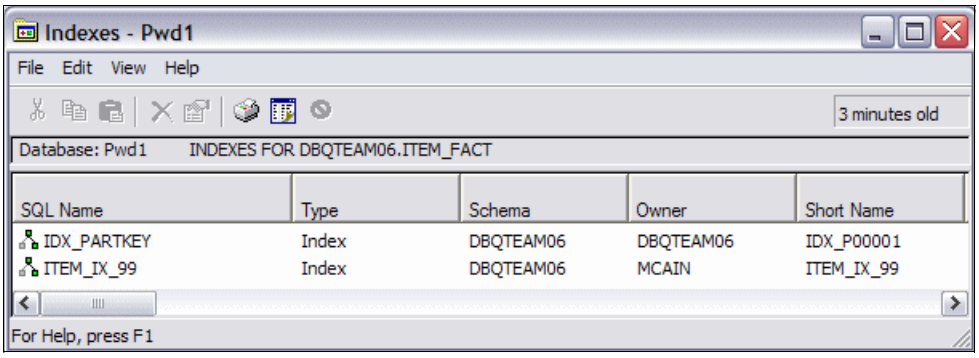


Figure 3-12 Two indexes in the ITEM_FACT table

The two counters update the columns:

- Last Query Use
- Last Query Statistic Use
- Query Use Count
- Query Statistics Use

Last Query Use is the column with the time stamp of when the index was used to access tables in a query, and Query Use Count is updated with a corresponding count. Last Query Statistics Use is updated when the optimizer uses the index to gather statistical information on a table, and Query Statistics Use is updated with a corresponding count as shown in Figure 3-13.

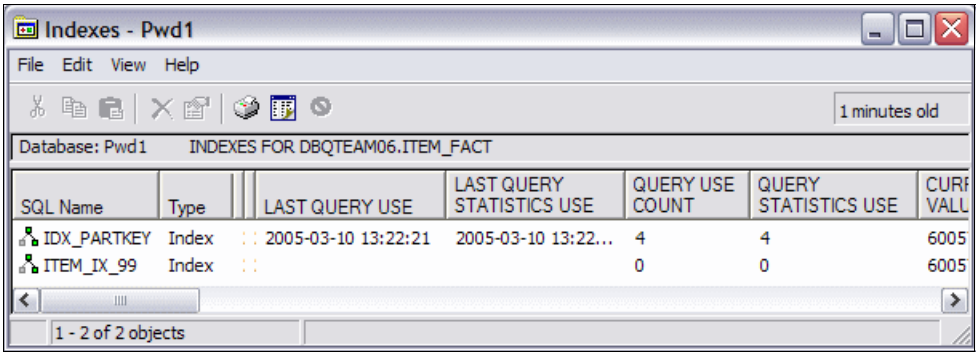


Figure 3-13 New information about indexes in V5R3 iSeries Navigator

To see the other information, such as an index definition and description, right-click the indexes. Figure 3-14 shows an example of a detailed index description.

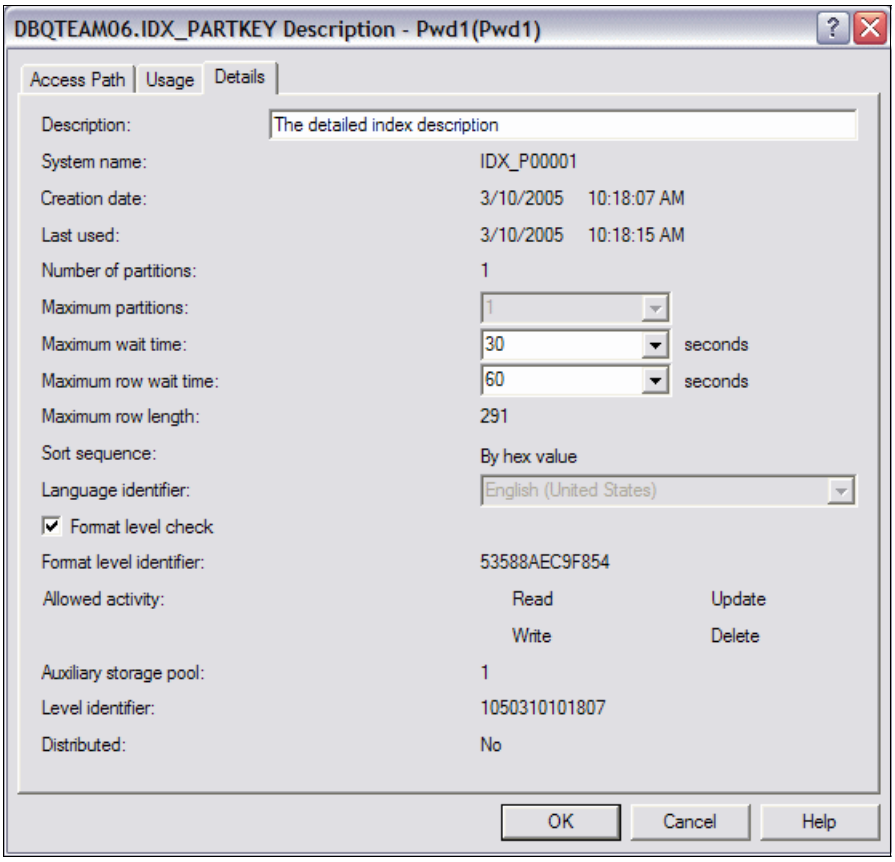


Figure 3-14 The detailed index descriptions

Note: You can also view the index information by using the Display File Description (DSPFD) command on the table. But the information does not include the newly added fields in V5R3.

3.6 The Database Performance Monitors

The *Database Performance Monitors* are integrated tools used to collect database-specific performance information for SQL requests being run on the iSeries. They let you keep track of resources that SQL statements use. The collected data is output to a database table or tables. Then reports or queries are run against the collected data from these output tables to analyze query optimization and the performance behavior of the database request. This analysis helps to identify and tune performance problem areas.

Note: The Database Performance Monitor name is also known as *SQL Performance Monitor*. From this point forward in the book, we use “SQL Performance Monitor” to refer to the iSeries Navigator function. We use “Database Monitors” when using a green screen and accessing the tool by running the Start Database Monitor (STRDBMON) CL command.

The SQL Performance Monitors can collect information about non-SQL query interfaces, such as OPNQRYF and Query for iSeries, but their primary usage is for SQL-based interfaces such as embedded SQL, Open Database Connectivity (ODBC), and JDBC.

The SQL Performance Monitors provide all the information that the STRDBG or PRTSQLINF commands provide plus additional information such as:

- System and job name
- SQL statement text
- Start and End time stamp
- 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
- ODP implementation
- QAQQINI settings

The collected information and statistics can be analyzed later to determine such information as:

- The number of queries that rebuild access plans
- The number of temporary indexes that have been created over a particular table
- The queries that are the most time consuming
- The user that is has the longest running queries
- The queries that were implemented using reusable ODPs
- Whether the implementation of a particular query changed with the application of a PTF or a new release (uses a before and after comparison of monitor data)

Two types of SQL Performance Monitors are available with OS/400 and i5/OS, both which are packaged with the operating system:

- Detailed Monitor
- Summary Monitor, also known as the *Memory-based Database Performance Monitor*

We discuss both monitors in the following sections.

3.6.1 Detailed Monitor

The Detailed Monitor gathers information about SQL requests and non-SQL queries, such as OPNQRYF. It has details about optimization and runtime behavior. The monitor data is dumped into a single output table.

Important: Use care when running the Detailed Monitor for long periods of time. The collection and output of performance data can consume both disk and CPU resources with its collection, resulting in system overhead.

STRDBMON and ENDDBMON

The CL commands Start Database Monitor (STRDBMON) and End Database Monitor (ENDDBMON) provide the interface for the Detailed Performance Monitors. The commands start and end the collection of database performance statistics.

The STRDBMON command gathers information about a query in real time and stores this information in an output table. The information can help you determine whether your system and your queries are performing as they should or whether they need fine tuning. The monitor

can be specified for a specific job or all jobs on system. Collected information and statistics are placed in a single output database table made up of different record types.

Consider following parameters before you start database monitoring:

- OUTFILE parameter
  - The file name is required; the library name is optional.
  - It is created if it doesn't exist or is reused if it does exit.
- OUTMBR parameter
  - Defaults to first member in file
- JOB parameter
  - Defaults to job issuing the STRDBMON command
  - Can specify one job or *ALL jobs (no subsetting allowed)
- TYPE parameter
  - The type of data collected is *SUMMARY(default) or *DETAIL.
  - *SUMMARY provides all necessary analysis data.
  - *DETAIL collects the same data as *SUMMARY plus the 3019 row.
    - It causes a little more overhead.
    - The Detailed SQL Performance Monitor interface in iSeries Navigator uses value of type (*DETAIL).

  Note: The *SUMMARY option in the TYPE parameter has *no relationship* to the detailed and summary SQL Performance Monitors in iSeries Navigator.

- FRCRCD parameter
  - This parameter uses the default value of *CALC.
  - A larger number reduces the overhead of the monitor; a smaller number increases it.
- COMMENT parameter: Description of collection

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

When collecting information for all jobs, the Database Monitor collects on previously started jobs or new jobs that are started after the monitor starts. Each job in the system can be monitored concurrently by two monitors:

- One monitor started specifically on that job, and another started for all jobs in the system.
- When monitored by two monitors and each monitor is logging to a different output file, monitor information is written to both files.

Database Monitors can generate significant CPU and disk storage overhead when in use. You can gather performance information for a specific query instead of every query to reduce this overhead on the system. You can also gather only specific monitoring data for smaller monitor collection since large monitor database files can slow analysis.

Consider the following guidelines:

- If possible, try collecting data only for the job that you want.
- Collect monitor data only for long running SQL statement based on the optimizer's estimated runtime.
- Eliminate the SQL statement generated by DB2.

- By using the table filter function, collect only statements that reference certain tables such as the one shown in the following example:

```
STRDBMON OUTFILE(o) COMMENT('TABLEFILTER(lib1/tab1,lib2/tab2)')
```

Note: To use the TABLE FILTER function, apply the following required PTFs in advance.

- *V5R2 PTFs*: SI15035, SI15139, SI15140, SI15142, SI15143, SI15154, and SI15155
- *V5R3 PTFs*: SI15955, SI16331, SI16332, and SI16333

For more information about valid parameter settings for the QAQQINI table, refer to "Monitoring your queries using Start Database Monitor (STRDBMON)" in *DB2 Universal Database for iSeries Database Performance and Query Optimization*, which is available in the iSeries Information Center on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzajq/rzajqmst.htm

3.6.2 Summary Monitor or Memory Resident Database Monitor

The Memory Resident Database Monitor is a tool that provides another method for monitoring database performance. This tool is intended only for the collection of performance statistics in SQL queries. To collect performance statistics for non-SQL queries, you should start a Detailed Database Monitor as explained in previous sections. The Memory-based Database Monitor or Summary Monitor, with the help of a set of APIs, manages database monitoring information for the user in memory. This memory-based monitor reduces CPU overhead and resulting table sizes.

The STRDBMON command 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 Summary Monitor collects much of the same information as the Detailed Database 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 Summary Monitor is not meant to replace the Detailed Database Monitor. There are circumstances where the loss of detail in the SQL Performance Monitor is not sufficient to fully analyze an SQL statement. In such cases, we recommend that you use the Detailed Database Monitor.

API support for the Memory Resident Database Monitor

A set of APIs provides support for the Summary Monitor or Memory-based Database Monitor that allow you to perform the activities listed in Table 3-1.

Table 3-1 External API description

API	Description
QQQSSDBM	This API starts the Memory-based Database Monitor. Database Monitor data is collected in the threaded process but summarized at the job level.
QQQCSDBM	This API clears and frees the associated memory are of the SQL monitor.
QQQDSDBM	This API dumps the contents of the SQL monitor table. The API does not force a clear operation (QQQCSDBM) of the memory. Data continues to be added to memory until the QQQCSDBM or QQQESDBM API is called.
QQQESDBM	This API ends the memory-based SQL monitor.
QQQQSDBM	This API queries the status of the Database Monitor, and returns information about the activity of the SQL and the original Database Monitor.

Figure 3-15 illustrates the different events in the Memory Resident Database Monitor life cycle and the APIs associated with each event.

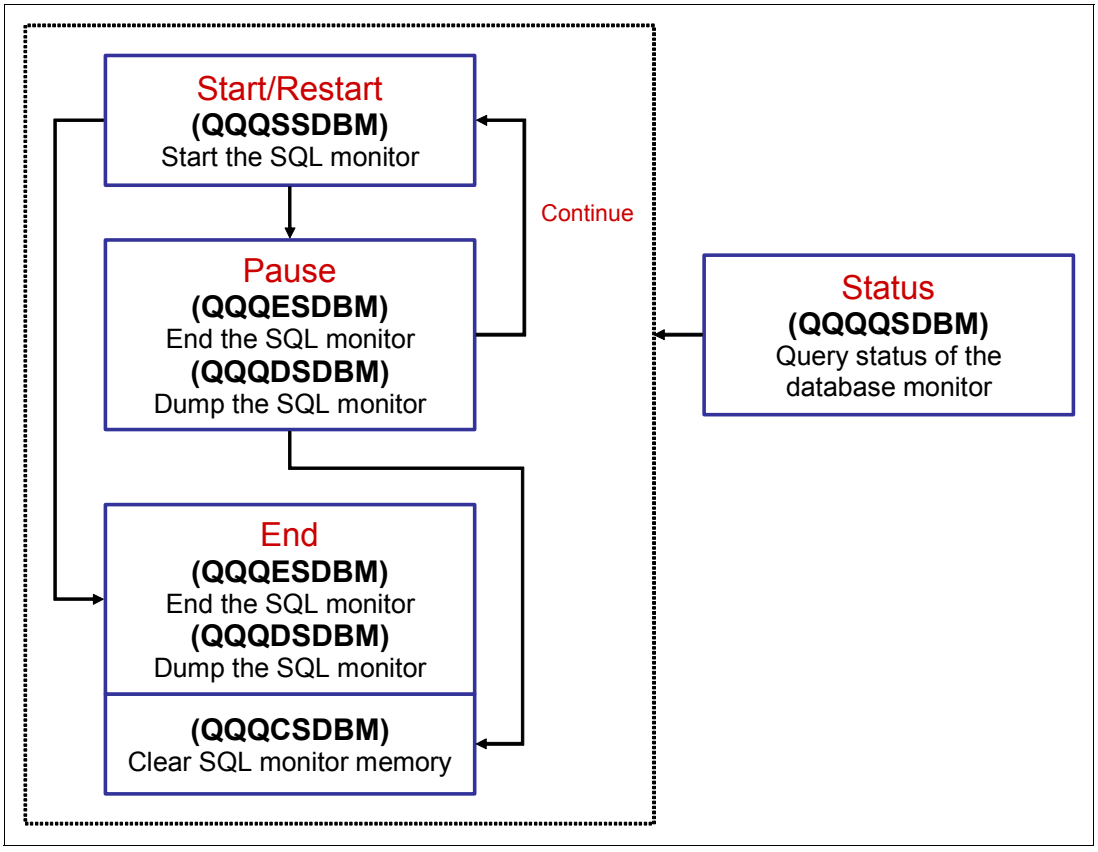


Figure 3-15 Memory Resident Database Monitor events and their APIs

For more information, search on `Memory Resident Database Monitor external API description` in the V5R3 iSeries Information Center at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp`

Notes:

- iSeries Navigator provides a graphical interface for these APIs, through the Summary SQL Performance Monitor to administer the memory-based collection mode and to run analytical dababase performance reports from the information collected.
- Unlike the Detailed Monitor, the Memory-based Database Monitor or Summary Monitor outputs or dumps the collected information into 10 separate, categorized output tables. To get the consolidated view of information collected for a single statement, you must run join queries.

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. The tables are:

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

For more information about the definitions of these tables, search on `Memory Resident Database Monitor: DDS` in the V5R3 iSeries Information Center:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp`

3.7 Visual Explain

Visual Explain provides a graphical representation of the optimizer implementation of a query request. The query request is broken down into individual components with icons that represent each unique component. Visual Explain also includes information about the database objects that are considered and chosen by the query optimizer. Visual Explain's detailed representation of the query implementation makes it easier to understand where the greatest cost is incurred.

Visual Explain shows the job run environment details and the levels of database parallelism that were used to process the query. It also shows the access plan in diagram form, which allows you to zoom to any part of the diagram for further details as shown in Figure 3-16.

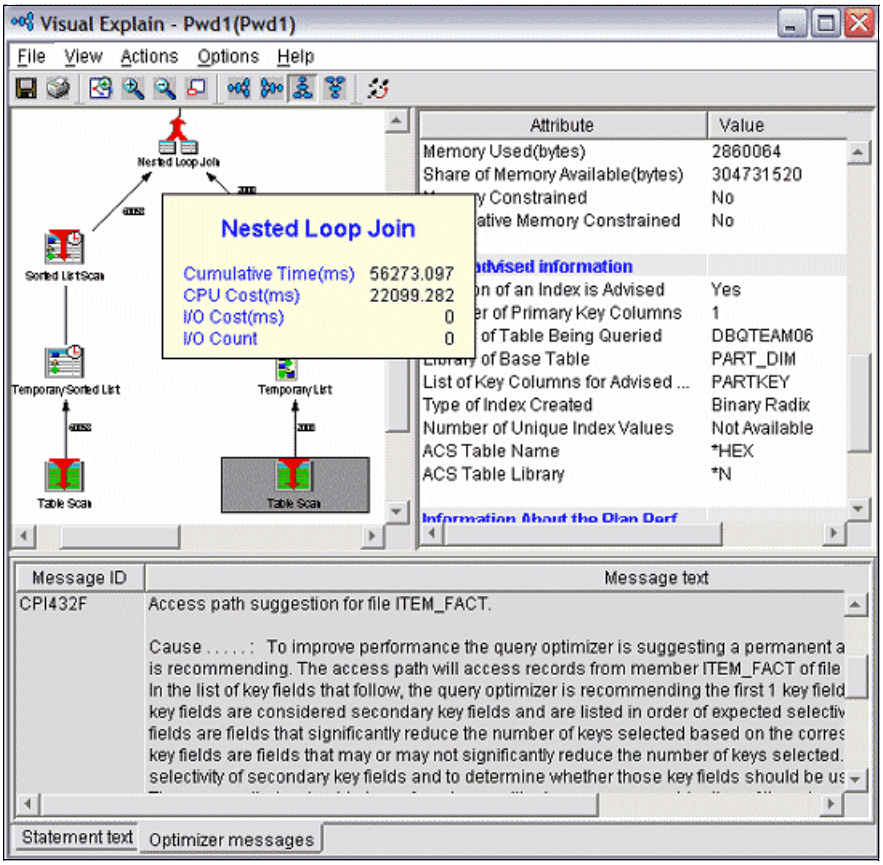


Figure 3-16 Visual Explain diagram

Visual Explain is a component of iSeries Navigator. From the Run SQL Script Center, you can access Visual Explain directly, either from the menu or from the toolbar.

Note: The Run SQL Script interface always deletes ODPs to force full optimization.

Another way to access Visual Explain is through SQL Performance Monitor. SQL Performance Monitor is used to create Database Monitor data and to analyze the monitor data with predefined reports. For more information about the SQL Performance Monitor, refer to Chapter 5, “Analyzing database performance data using iSeries Navigator” on page 93.

Visual Explain works with the monitor data that is collected by SQL Performance Monitor on that system or by the Start Database Monitor (STRDBMON) command. Visual Explain can also analyze Database Monitor data that is collected on other systems after data is restored on the iSeries server.

For more information about the Visual Explain, refer to Chapter 8, “Analyzing database performance data with Visual Explain” on page 197.

4

Gathering database SQL performance data

After you identify a possible Structured Query Language (SQL) performance problem, you must start gathering SQL performance data. In this chapter, we discuss the different tools that are available to gather this data. Among these tools are the Detailed Database Monitor and the Summary Database Monitor or Memory Resident Database Monitor.

We explain how to use iSeries Navigator to gather the Database Monitor data. We also describe the table layout of the Database Monitor and the different record layouts of the Database Monitor data.

4.1 Types of SQL Performance Monitors

Database Monitors have been part of the OS/400 operating system since V3R6. Database Monitors and SQL Performance Monitors are used to gather information about queries run in DB2 Universal Database for iSeries. This information can help you determine whether your system and your queries are performing as they should, or whether they need fine tuning. There are two types of SQL Performance Monitors:

- Detailed Database Monitors
- Summary Monitors (also known as *Memory Resident Database Monitors*)

The following sections describe the different ways that you can enable these two types of monitors to gather database performance data.

4.2 Collecting monitor data

There are different ways to start a Database Monitor, as we explain in the following sections, for:

- Open Database Connectivity (ODBC) clients
- Object Linking and Embedding (OLE) DB clients
- Java Database Connectivity (JDBC) clients
- Using an exit program

4.2.1 Starting a Detailed Database Monitor

The Start Database Monitor (STRDBMON) command starts a Detailed Database Monitor. STRDBMON starts the collection of database performance statistics for a specified job or all jobs on the system.

You can start the Detailed Database Monitor in a number of ways, all of which use the STRDBMON command as the basis. You can issue the STRDBMON command from a command line, a CL program, iSeries Navigator, and for ODBC, from within the data source name. In this section, we look at all the options that are available to the database administrator.

When you start a new monitor with the STRDBMON command, you must specify the file to which the performance statistics are to be written. If the file or member does not exist, one is created based on the QAQQDBMN file in library QSYS. If the file or member does exist, the record format of the file is checked to see if it is the same.

To start the Detailed Database Monitor, from a command line, type one of the following commands depending on whether you want to gather details for all the jobs on the system or a particular job:

```
STRDBMON OUTFILE(MYLIB/DBMON) JOB(*ALL) TYPE(*DETAIL)
STRDBMON OUTFILE(MYLIB/DBMON) JOB(999999/SQLUSER/IBM) TYPE(*DETAIL)
```

The commands use the following parameters:

- OUTFILE

  The file name for the results file is required, but the library name is optional. The file is created if it does not exist, but is reused if it already exists.

- OUTMBR

  This parameter defaults to the first member in the file. Specify the *ADD or *REPLACE option. *REPLACE is the default option for this parameter.

- JOB

  This parameter defaults to the job issuing the STRDBMON command. The user can specify a single job on the system or specify *ALL for all jobs.

  If the monitor is started on all jobs, any jobs waiting on job queues or any jobs started during the monitoring period have statistics gathered from them after they begin. If the monitor is started on a specific job, that job must be active in the server when the command is issued.

  Note: Each job in the server can be monitored concurrently by only two monitors:

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

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

- TYPE

  This parameter allows the user to specify the type of data to be collected:

  – *SUMMARY provides all necessary analysis data.
  – *DETAIL collects the same data as *SUMMARY plus the 3019 row. This type of collection causes a little more overhead on the system.

  Tip: It is important to clarify that the STRDBMON command is a Detailed Database Monitor regardless of which option is specified in the *DETAIL parameter. Specifying *DETAIL is only useful for non-SQL queries, which are those queries that do not generate a QQQ1000 row. For non-SQL queries, the only way to determine the number of rows returned and the total time to return those rows is to collect *detail rows*. Currently the only detail row is the 3019 row.

  While the detail row contains valuable information, it creates a slight performance degradation for each block of rows returned. Therefore you must closely monitor its use.

- FRCRCD

  This parameter allows you to control the number of rows that are kept in the row buffer of each job being monitored before forcing the rows to be written to the output table. The default value is *CALC.

  By specifying a force row write value of 1, FRCRCD(1) monitor rows are displayed in the log as soon as they are created. FRCRCD(1) also ensures that the physical sequence of the rows is most likely, but not guaranteed, to be in time sequence. However, FRCRCD(1) causes the most negative performance impact on the jobs being monitored. By specifying a larger number for the FRCRCD parameter, the performance impact of monitoring can be reduced.

- COMMENT

  This parameter allows you to add a meaningful description to the collection. It specifies the description that is associated with the Database Monitor record whose ID is 3018.

With STRDBMON, you can monitor SQL queries executed in DB2 Universal Database for iSeries that come from different SQL interfaces such as ODBC, IBM Toolbox for Java, native JDBC, or SQL call level interface (SQL CLI). Moreover, when you choose to collect detailed information, you can also monitor queries that come from non-SQL interfaces such as OPNQRYF and Query for iSeries.

4.2.2 Ending a Detailed Database Monitor

To end the collection of database performance data for a specified job or all jobs on the system, you must use the End Database Monitor (ENDDBMON) command. The following parameters are available for this command:

- JOB

  You can specify the job name or *ALL to end only the monitor that was started with that same job name. It is possible to end one monitor on a job and still have another one collecting on that same job.

- COMMENT

  This parameter enables you to describe the collection.

If an attempt is made to end the monitor on all jobs, then a STRDBMON command must have been issued previously for all jobs. If a particular job is specified in this command, the job must have the monitor started explicitly and specifically on that job.

For more information about the STRDBMON or ENDDBMON commands, visit the V5R3 iSeries Information Center at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp

4.2.3 Enabling Database Monitors in ODBC clients

You can start a Detailed Database Monitor before a client initiates a connection to the server, but using the STRDBMON command to monitor the client job. Other options are available for you to start a Database Monitor. The two ways to start a Detailed Database Monitor in ODBC clients are:

- Enable the Database Monitor option in the data source name
- Use an ODBC connection keyword to start the Database Monitor

ODBC data source name

To start a Detailed Database Monitor using the data source name (DSN):

1. In Microsoft® Windows XP, click **Start** → **Programs** → **IBM iSeries Access for Windows** and then select **ODBC administration**.
2. From the ODBC Data Source Administrator window (Figure 4-1), select the desired ***data source name*** and click the **Configure** button.

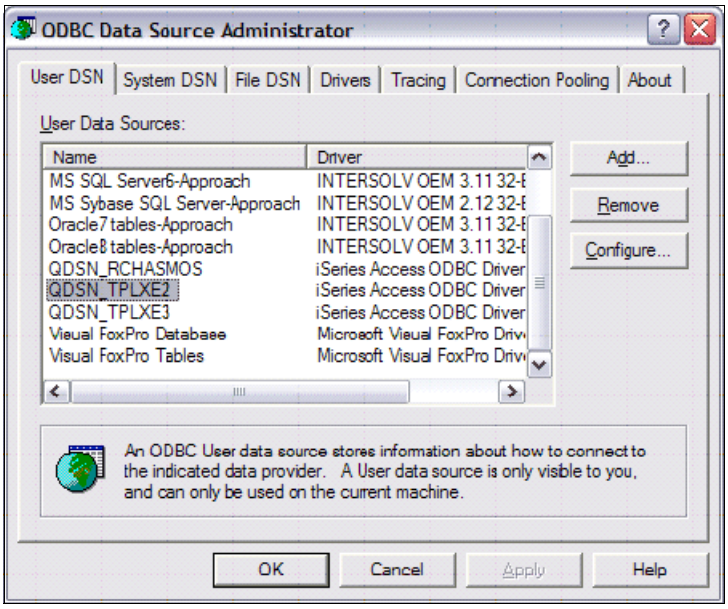


Figure 4-1 ODBC Data Source Administrator window

3. In the iSeries Access for Windows ODBC setup window (Figure 4-2) that opens, click the **Diagnostic** tab and then select the **Enable Database Monitor** option. Then click **OK**.

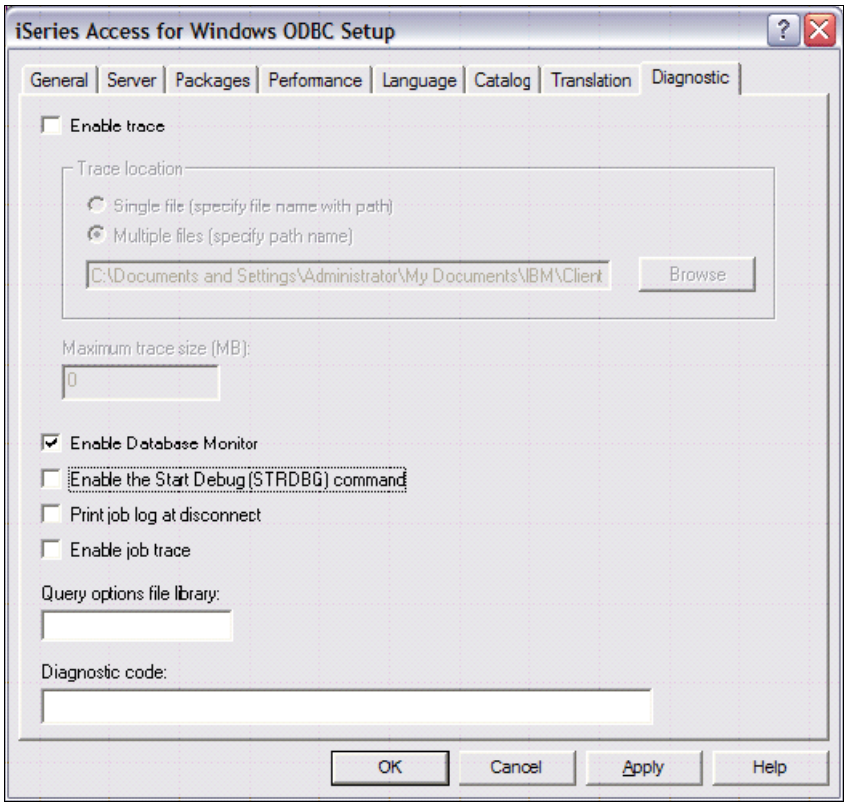


Figure 4-2 iSeries Access for Windows ODBC Setup window

The Enable Database Monitor option causes the ODBC driver to issue a call to STRDBMON for the job connecting to this data source name. The output file is created in the QUSRSYS

library starting with the prefix QODB and ending with the job number, for example QUSRSYS/QODB337344.

The ODBC driver attempts to end the Database Monitor when the application disconnects. If the application ends abnormally without issuing a disconnect, the Database Monitor might continue to run. In this case, you must manually end the monitor by entering the ENDDBMON command and specifying the job number being traced (or *ALL if no other Database Monitors are active).

ODBC connection keywords

One potential problem with redistributing an application that uses ODBC is that a data source might need to be created on each user's PC. Data sources are normally created using ODBC C APIs. However, this interface might be difficult to use for some programming languages.

An alternative for this potential problem is for a client to connect to the server without using an ODBC data source and to use connection keywords instead. The iSeries Access ODBC driver has many connection string keywords that can be used to change the behavior of the ODBC connection. These same keywords and their values are also stored when an ODBC data source is setup.

Tip: When an ODBC application makes a connection, any keywords specified in the connection string override the values that are specified in the ODBC data source.

The connection keyword to enable the Database Monitor is the TRACE keyword. Currently the TRACE keyword supports the following options:

- 0 = No tracing
- 1 = Enable internal driver tracing
- 2 = Enable Database Monitor
- 4 = Enable the Start Debug (STRDBG) command
- 16 = Enable job trace

To specify multiple trace options, add together the values for the options that you want. For example, if you want to activate the Database Monitor (2) and the STRDBG command (4) on the server, then the value that you specify is 6 (2+ 4=6).

Important: Use these options only when debugging problems because they can adversely affect performance.

The part of a Microsoft Excel® macro in Example 4-1 uses a data source name and the TRACE connection keyword to enable the Database Monitor and to enable the STRDBG command. Because this example uses both a data source name and a connection keyword, the value in the TRACE keyword overrides the values specified in the data source name.

Example 4-1 Microsoft Excel macro using a data source name and TRACE connection keyword

```
'ODBC connection.
   'The system is specified in the B1 cell in the Settings worksheet
   ConnectStr = connectStr & "DSN=QDSN_TPLXE2;System=" &
Worksheets("Settings").Range("B1").Value & ";TRACE=6"
   CnMYAS400.Open connectStr
```

The TRACE keyword, including the value to Enable Database Monitor, causes the ODBC driver to issue a call to STRDBMON for the current job. The output file is created in the

QUSRSYS library, starting with the prefix QODB, and ending with the job number, for example QUSRSYS/QODB337344.

4.2.4 Enabling Database Monitors in OLE DB clients

There are two ways you can start a Detailed Database Monitor in OLE DB clients:

- Using an OLE DB connection property
- Using an OLE DB connection keyword

OLE DB connection properties

A set of custom properties (IBMDA400, IBMDARLA, and IBMDASQL) is available for the OLE DB providers shipped with iSeries Access for Windows. The trace property (available in V5R3) is used to enable diagnostic traces when troubleshooting errors. It is an integer property, and several numeric constants are defined for various trace options.

To determine the value this property should contain, select the desired trace options and add the constant values. The resulting number is the value that should be specified. The constants are:

- 0 = No trace options (the default value)
- 1 = Enable Database Monitor
- 2 = Enable the STRDBG command
- 4 = Print Job Log at disconnect
- 8 = Enable Job trace via the Start Trace (STRTRC) command

Example 4-2 shows how to enable the Database Monitor using Visual Basic®.

Example 4-2 Enabling the Database Monitor using Visual Basic

```
Dim cnAS400 as ADODB.Connection
   Dim strJobName as String

   Set cnAS400 = New ADODB.Connection

   'Set the provider to Client Access
   cnAS400.Provider = "IBMDA400"

   'Set custom properties.
   cnAS400.Properties("Block Fetch") = True
   cnAS400.Properties("Catalog Library List") = "LIBRARY1, LIBRARY2"
   cnAS400.Properties("Convert Date Time To Char") = "FALSE"
   cnAS400.Properties("Default Collection") = "MYLIB"
   cnAS400.Properties("Force Translate") = 0
   cnAS400.Properties("Cursor Sensitivity") = 0
   cnAS400.Properties("Data Compression") = True
   cnAS400.Properties("Hex Parser Option") = 0
   cnAS400.Properties("Initial Catalog") = "*SYSBAS"
   cnAS400.Properties("Maximum Decimal Precision") = 31
   cnAS400.Properties("Maximum Decimal Scale") = 31
   cnAS400.Properties("Minimum Divide Scale") = 0
   cnAS400.Properties("Query Options File Library") = "QUSRSYS"
   cnAS400.Properties("SSL") = "FALSE"
   cnAS400.Properties("Trace") = 1 'Enable Database Monitor

   'Open the connection
   cnAS400.Open "Data Source=MySystem;", "USERID", "PWD"
   strJobName = cnAS400.Properties("Job Name")
```

OLE DB connection keywords

In addition to using the trace property to enable Database Monitor in OLE DB, you can use the Trace connection keyword. In Example 4-3, we illustrate an Excel macro that uses the Trace connection keyword to enable Database Monitor at connection time.

Example 4-3 Excel macro using the Trace connection keyword

```
'OLE DB Connection
    connectStr = connectStr & "provider=IBMDA400;data source=" &
Worksheets("Settings").Range("B1").Value & ";TRACE=1"
   cnMYAS400.Open connectStr
```

The trace property or the trace connection keyword, including the value to Enable Database Monitor, causes the OLE DB Provider to issue a call to STRDBMON for the current job. The output file is created in the QUSRSYS library, starting with the prefix QODB and ending with the job number, for example QUSRSYS/QODB337344.

4.2.5 Enabling Database Monitors in JDBC clients

The JDBC driver shipped with the Developer Kit for Java (commonly known as the *native JDBC driver*) and the IBM Toolbox for Java JDBC driver support a server trace connection property. Among other options, it includes an option to start a Detailed Database Monitor.

Important: These tracing options *work* correctly only if you are using an *SQL naming convention*. They do not work if the JDBC connection is created using a system naming convention. A future version of the drivers will address this issue with these options.

In Example 4-4, the Java code uses a properties object to enable tracing. The example uses native JDBC and the IBM Toolbox for Java JDBC concurrently.

Example 4-4 Java code using JDBC

```
// Register both drivers.
   try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      Class.forName("com.ibm.as400.access.AS400JDBCDriver");
   } catch (ClassNotFoundException cnf) {
      System.out.println("ERROR: One of the JDBC drivers did not load.");
      System.exit(0);
   }

   Connection conn1, conn2;
   Properties props = new Properties();
   props.setProperty("user", "MYUSER");
   props.setProperty("password", "MYPASSWORD");
   props.setProperty("server trace", "n");

   try {
      // Obtain a connection with each driver.
      conn1 = DriverManager.getConnection("jdbc:db2://localhost", props);
      conn2 = DriverManager.getConnection("jdbc:as400://localhost", props);

      conn1.close();
      conn2.close();
   } catch (SQLException e) {
      System.out.println("ERROR: " + e.getMessage());
   }
```

In this example, *n* is a numeric bitmap of selected trace options. The options that can be specified include:

- 1 = Client trace
- 2 = Enable Database Monitor
- 4 = Debug job log
- 8 = Save job log
- 16 = Job trace
- 32 = Save SQL output

The Database Monitor option causes the drivers to issue a call to STRDBMON for the current job. The output file is created in the QUSRSYS library, starting with the prefix QJT for the toolbox driver and the prefix QSQL for the native driver, and ending with the job number for example, QUSRSYS/QJT744340 or QUSRSYS/QSQL744340.

4.2.6 Enabling Database Monitors using an exit program

An exit program provides another way to start a Detailed Database Monitor for clients using the QZDASOINIT, QZDASSINIT, or QZDAINIT prestart jobs, such as ODBC, OLE DB, or IBM Toolbox for Java clients. The database server has five different exit points defined. QIBM_QZDA_INIT is one of those exit points and is called when the prestart jobs are started.

By using the Work with Registration Info (WRKREGINF) command, you can add or remove the exit program, which has a STRDBMON command, to the exit point. You must end and restart the prestart jobs using the Start Prestart Jobs (STRPJ) and End Prestart Jobs (ENDPJ) commands for the change to take effect. While we do not discuss this option further, we strongly recommend that you use one of the options previously described. To learn more about exit programs, search on `register exit programs` in the V5R3 iSeries Information Center at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp`

4.3 Collecting monitor data using iSeries Navigator

In this section, we explain how to start the two types of SQL Performance Monitors using iSeries Navigator.

4.3.1 Starting a Memory Resident or Summary Database Monitor

To start the Memory Resident or Summary Database Monitor within iSeries Navigator, right-click **SQL Performance Monitors** and select **New** → **Summary** as shown in Figure 4-3.

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, the data is written to hard disk and can then be analyzed. Because the monitor stores its information in memory, the performance impact to your system is minimized.

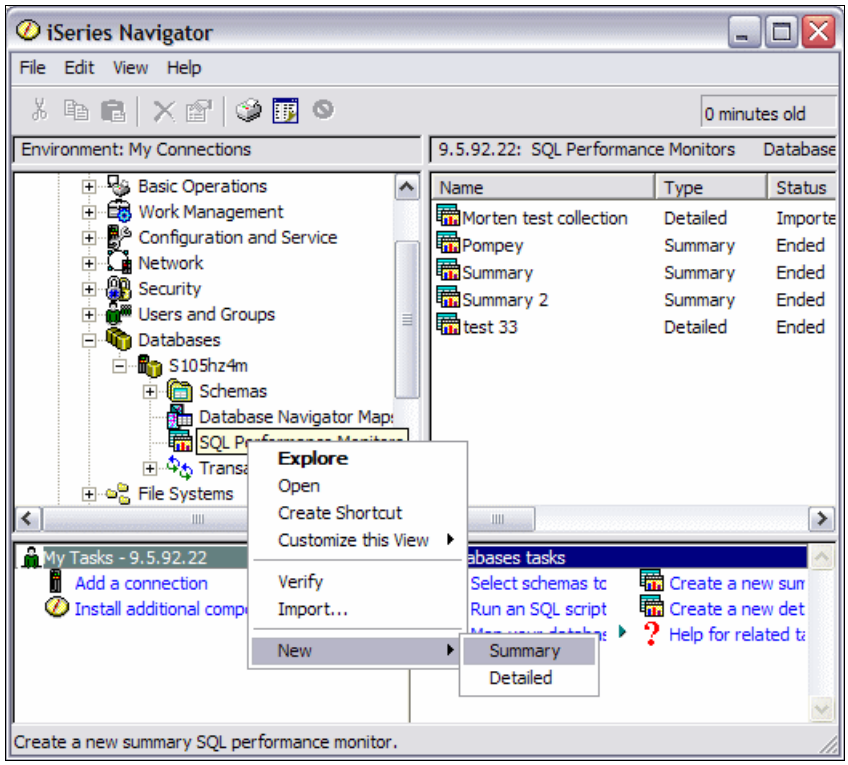


Figure 4-3 Starting a Summary Database Monitor using iSeries Navigator

In the New Summary SQL Performance Monitor window (Figure 4-4) that opens, you can name the Summary Monitor and select the jobs that you want to monitor and the specific data that you require the monitor to collect.

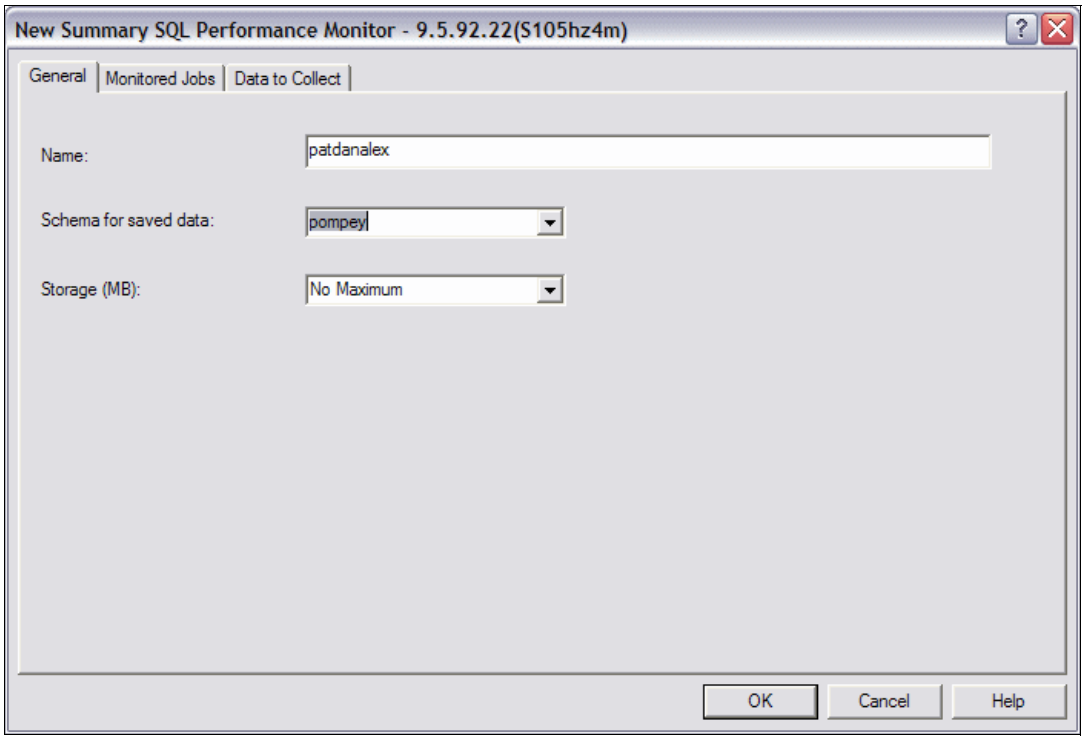


Figure 4-4 General tab

Figure 4-4 shows the General tab on which you provide the following information:

- Name (of the summary monitor)

  In this field, you specify the name of the monitor. This name must be unique in the schema in which it is created, can be up to 30 characters long, and exists only in iSeries Navigator. This is a required field.

- Schema (where the monitor will reside on the iSeries server)

  In this field, you specify the schema where the monitor is saved. The system automatically folds all lowercase characters to uppercase. You can change this by placing the schema name with special characters within double quotation marks. This field is required to start the monitor.

- Storage (amount that you will allow the monitor to you)

  This field specifies the size limit of the monitor. The default value is not the limit. If the limit is reached, the monitor continues collecting the data at the beginning of the table, overwriting previous data.

The Monitored Jobs tab, shown in Figure 4-5, allows you to specify which jobs to monitor. You can choose which jobs you want to monitor or choose to monitor all jobs.

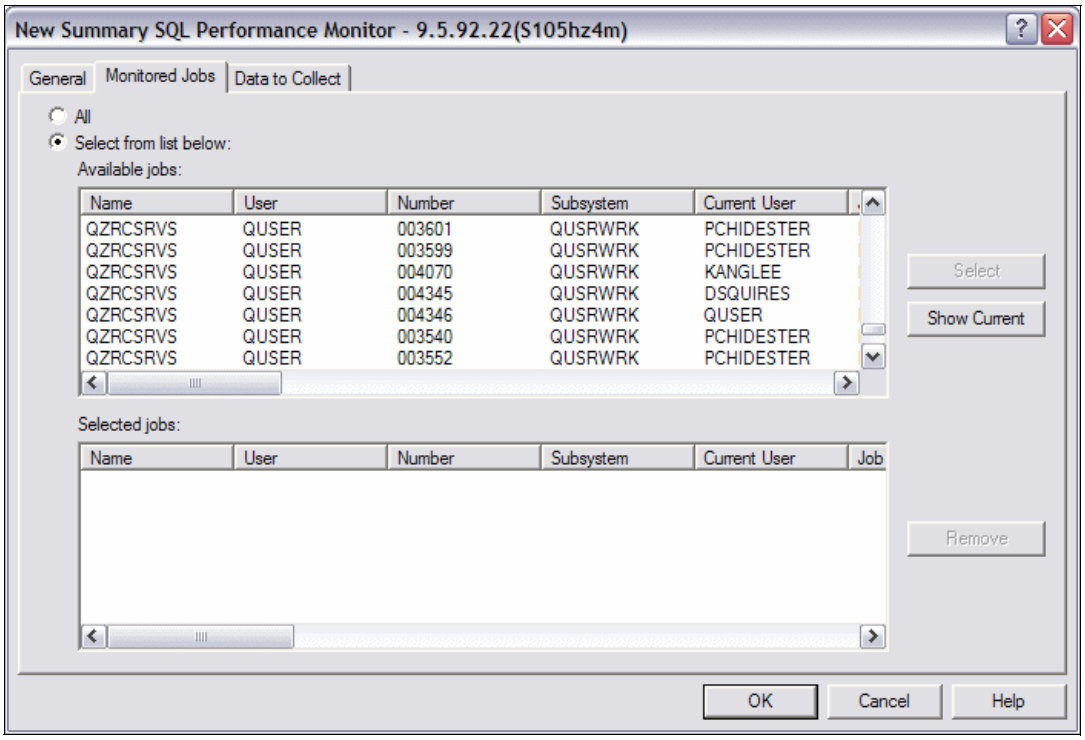


Figure 4-5 Monitored Jobs tab

You can have multiple instances of monitors running on you system at one time. However, only one monitor instance can monitor all jobs. Additionally, you cannot have two monitors monitoring the same job. When collecting information for all jobs, the monitor collects 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.

In the Data to Collect tab (Figure 4-6), select the options for the kind of data that you want to collect.

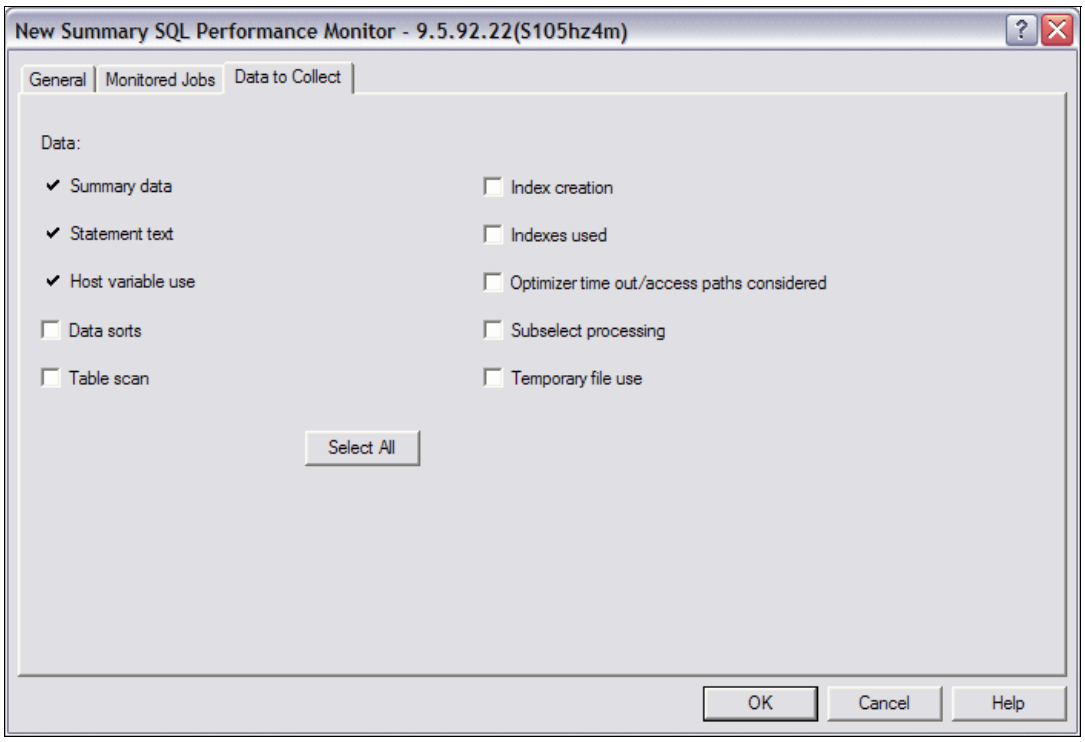


Figure 4-6 Options for Summary Monitor

The options on the Data to Collect tab collect the following information:

- **Summary data**: Contains resource and other general information about monitored jobs; option available only on this page.
- **Statement text**: Contains the SQL text that monitored jobs call
- **Host variable use**: Contains the values of host variables that monitored jobs use
- **Data sorts**: Contains details of data sorts that monitored jobs perform
- **Table scan**: Contains the table scan data for the monitored jobs
- **Index creation**: Contains details of the creation of indexes by monitored jobs
- **Indexes used**: Contains details of how indexes are used by monitored jobs
- **Optimizer timeout/access paths considered**: Contains details about any occurrences of timeouts of monitored jobs
- **Subselect processing**: Contains information about each subselect in an SQL statement
- **Temporary file use**: Contains details about temporary files that monitored jobs created

Finally, you click **OK** to start the Summary Database Monitor. The new monitor should be listed in the right pane of the iSeries Navigator window with the type *Summary* and a status of *Started*.

You can pause or end a Summary Monitor, or you can continue a monitor that was previously paused. However, you cannot continue a monitor that was previously ended. When the monitor is paused or ended, the monitored data that resides in memory is written into several database tables that can be queried and analyzed using the predefined reports that come with the tool.

After the data is collected, you can look at the properties of the monitor. In the SQL Performance Monitor window, you right-click the monitor in question and click **Properties**.

Figure 4-7 shows the properties of a collected Summary Monitor. Notice the File column, in which the files or tables reside in the schema that was specified when the monitor was started. On this page, you also see the file names and the collection period of the data for which the monitor ran. If you are viewing an imported monitor, the statement "`Information Not Available`" is displayed in the Collection period field.

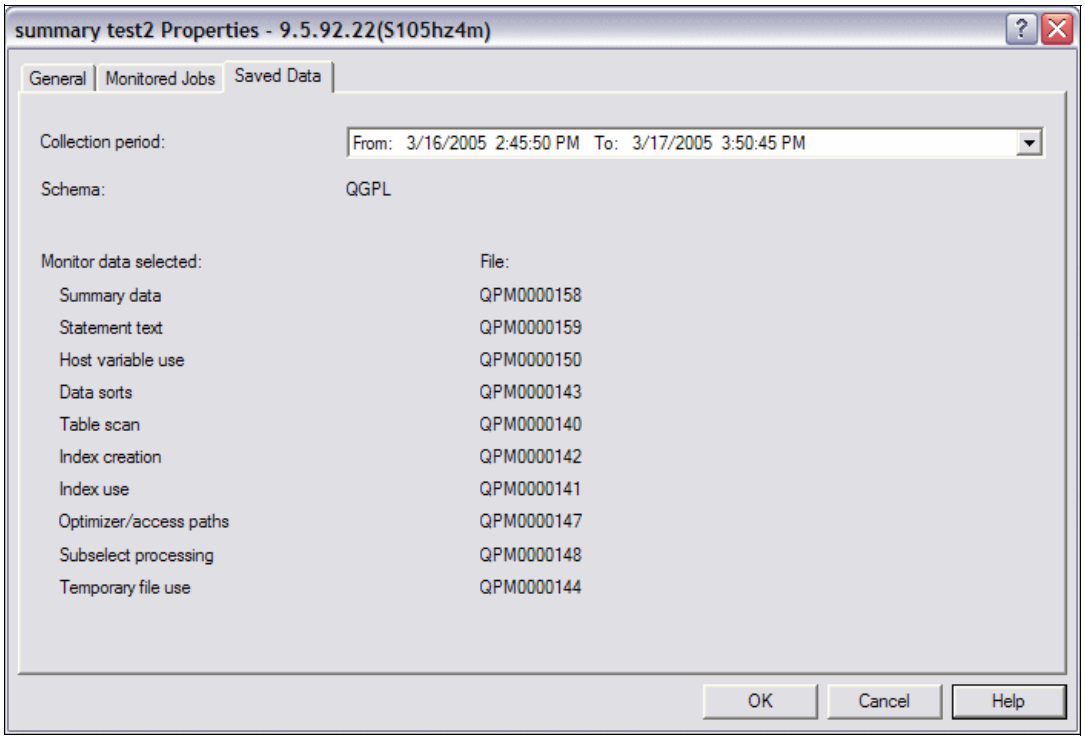


Figure 4-7 Summary Saved Data tab

External table description

The following externally described file definitions apply to those shown in Figure 4-7. For your reference, each file includes the file name for clarification.

Tip: The file names as they are shown in Figure 4-7 are specific to this iSeries server. They will change with each collection.

- QAQQQRYI Query (SQL) information (QPM0000158)

  Displays the file that contains the summary data
- QAQQTEXT SQL statement text (QPM0000159)

  Displays the file that contains the statement text
- QAQQ3000 Table scan (QPM0000140)

  Displays the file that contains the table scan
- QAQQ3001 Index used (QPM0000141)

  Displays the file that contains index use

- QAQQ3002 Index created (QPM0000142)

  Displays the file that contains index creation
- QAQQ3003 Sort (QPM0000143)

  Displays the file that contains the data sorts
- QAQQ3004 Temporary table (QPM0000144)
- Displays the file that contains temporary file use
- QAQQ3007 Optimizer timeout/all indexes considered (QPM0000147)

  Displays the file that contains optimizer or access paths
- QAQQ3008 Subquery (QPM0000148)

  Displays the file that contains information about each subselect in an SQL statement
- QAQQ3010 Host variable values (QPM00000150)

  Displays the file that contains the host variable use

Note: To see the data description specification (DDS) for the file, see Chapter 12 in *DB2 Universal Database for iSeries Database Performance and Query Optimization*, which is available on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/rzajq/rzajqmst.pdf

When to use the Summary Database Monitor

The Memory Resident or Summary Database Monitor, as already discussed, has advantages over the Detailed Database Monitor created when using the STRDBMON command. The Memory Resident Database Monitor can be more useful than the Detailed Database Monitor in the following situations:

- When you are not sure what is causing the performance issue that you might be seeing

  You can collect a lot of data over the entire system, in a short amount of time, using the Detailed Monitor. It is better to collect a Summary Monitor over the entire system and analyze that.
- When you want to monitor the system over a period of time to compare, for example, the results of one week against another

  This helps with a proactive approach to SQL performance and with trends in the way the database is performing.
- When the size of the data collected for a Summary Monitor started against the whole system is still significantly smaller

  This can help the customer provide IBM Support with initial problem determination data.

Tip: IBM Support might request a Detailed Monitor for more detailed analysis of the issues. Also the space used by the Memory Resident Database Monitor on the system is significantly less. Two monitors, side-by-side over the same SQL functions, collected only around 5% of the full Detailed Database Monitor.

Data conversion problems

You can use the Memory Resident Database Monitor to analyze possible problems with data conversion issues. Collect the Database Monitor data over the system and then use the code shown in Example 4-5 to evaluate the data.

Example 4-5 SQL to run Memory Resident Database Monitor to show data conversion issues

```
select count(*) as Conversions, a.qqdacv, b.qqsttx
from qgpl.qpm0000158 a, qgpl.qpm0000159 b
where a.qqkey = b.qqkey and a.qqdacv between '1' and '9'
group by a.qqdacv, b.qqsttx
order by 2 asc
```

A single data conversion operation is inexpensive, but repeated thousands or millions of times, it can add up. In some cases, it is a simple task to change one of the attributes so that a faster direct map can be performed. In other cases, the conversion is necessary because no exact matching data type is available. One of the advantages of the Memory Resident Database Monitor, as previously discussed, is the affect that the monitor has on the system. In this example, it is easy to see whether you have data conversion issues.

4.3.2 Starting a Detailed Database Monitor

Using the detailed SQL Performance Monitor in iSeries Navigator is equivalent to running the STRDBMON command with *DETAIL in the TYPE parameter, which is explained in 3.6.1, “Detailed Monitor” on page 44.

To start a detailed SQL Performance Monitor on the iSeries Navigator window, right-click **SQL Performance Monitors** and select **New** → **Detailed** as shown in Figure 4-8.

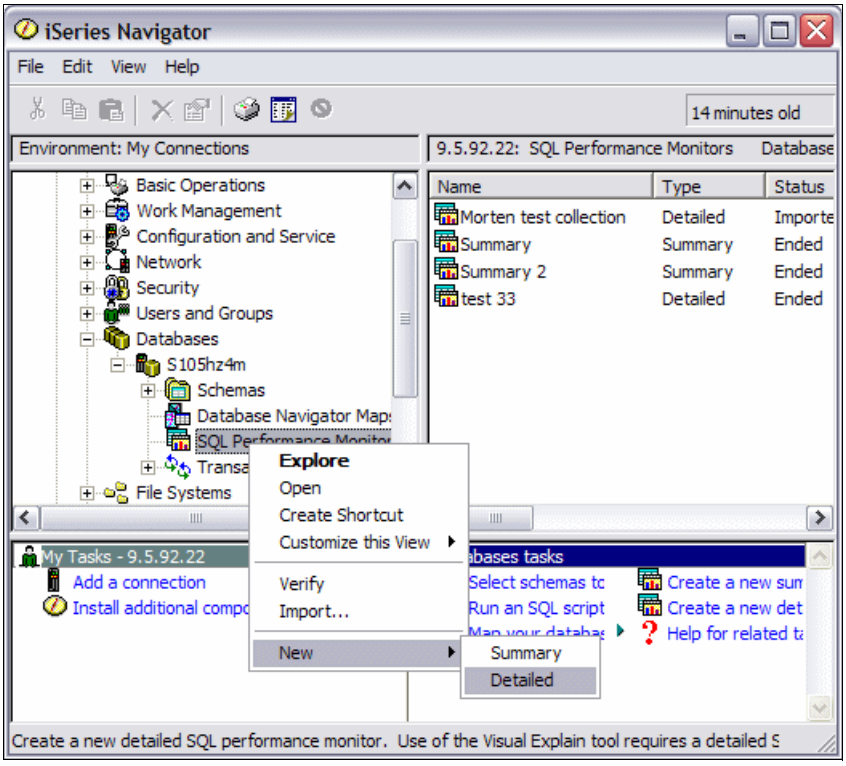


Figure 4-8 Starting the Detailed Database Monitor on iSeries Navigator

In the New Detailed SQL Performance Monitor window (Figure 4-9) that opens, under the **General** tab, specify the name of the monitor and the schema to be saved.

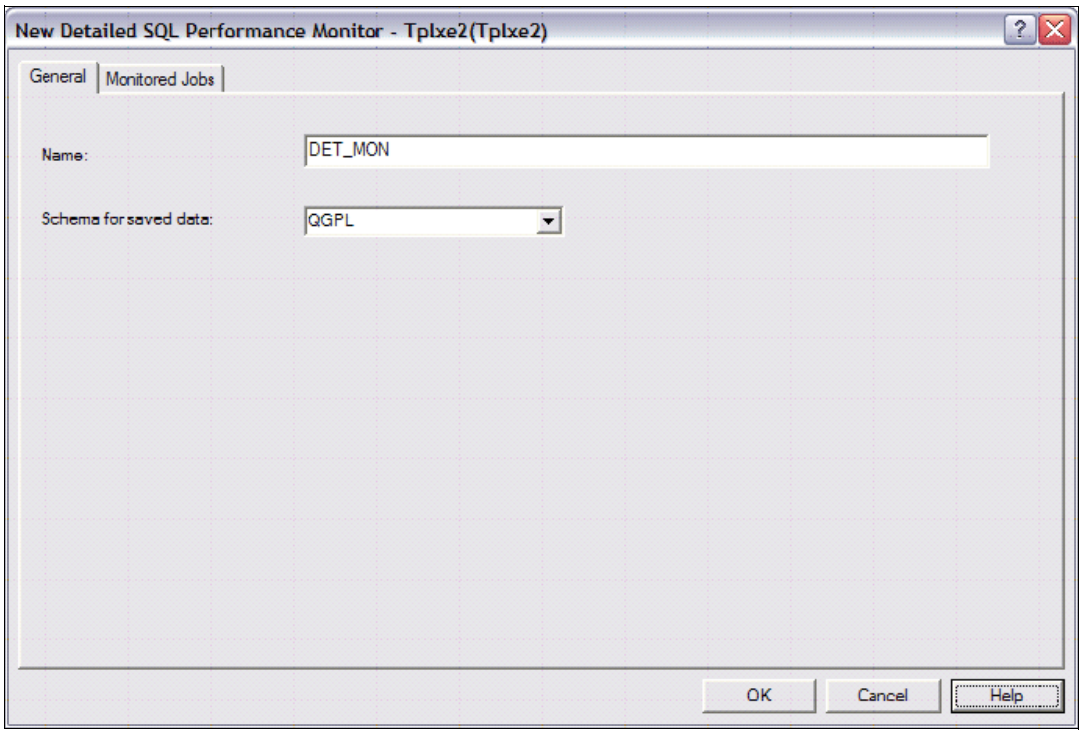


Figure 4-9 General tab of the Detailed SQL Performance Monitor window

In the **Monitored Jobs** tab, select the jobs that you are interested in monitoring. You can specify all jobs or select from the list of jobs as shown in Figure 4-10.

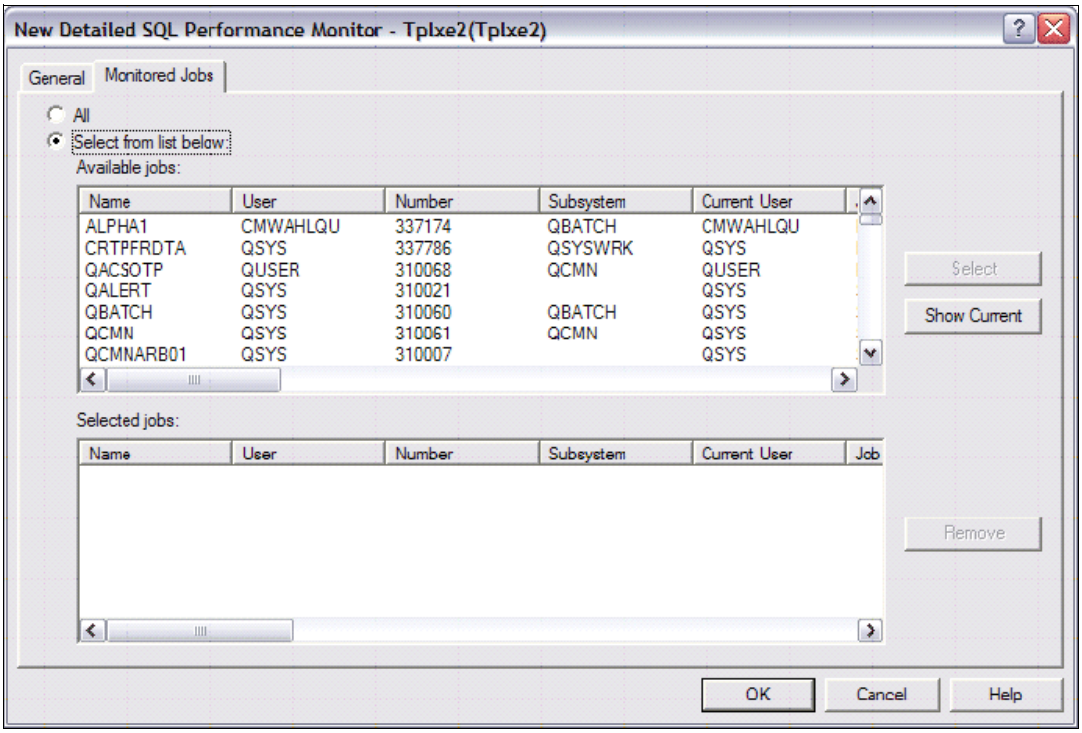


Figure 4-10 Monitored Jobs in the Detailed SQL Performance Monitor window

Selecting All (for all jobs) might cause a significant overhead on the system, particularly if it is a detailed SQL Performance Monitor.

Finally, click **OK** and the Detailed SQL Performance Monitor begins. You should see the new monitor listed in the right pane of iSeries Navigator, with a Type of *Detailed* and a Status of *Started*, as shown in Figure 4-11.

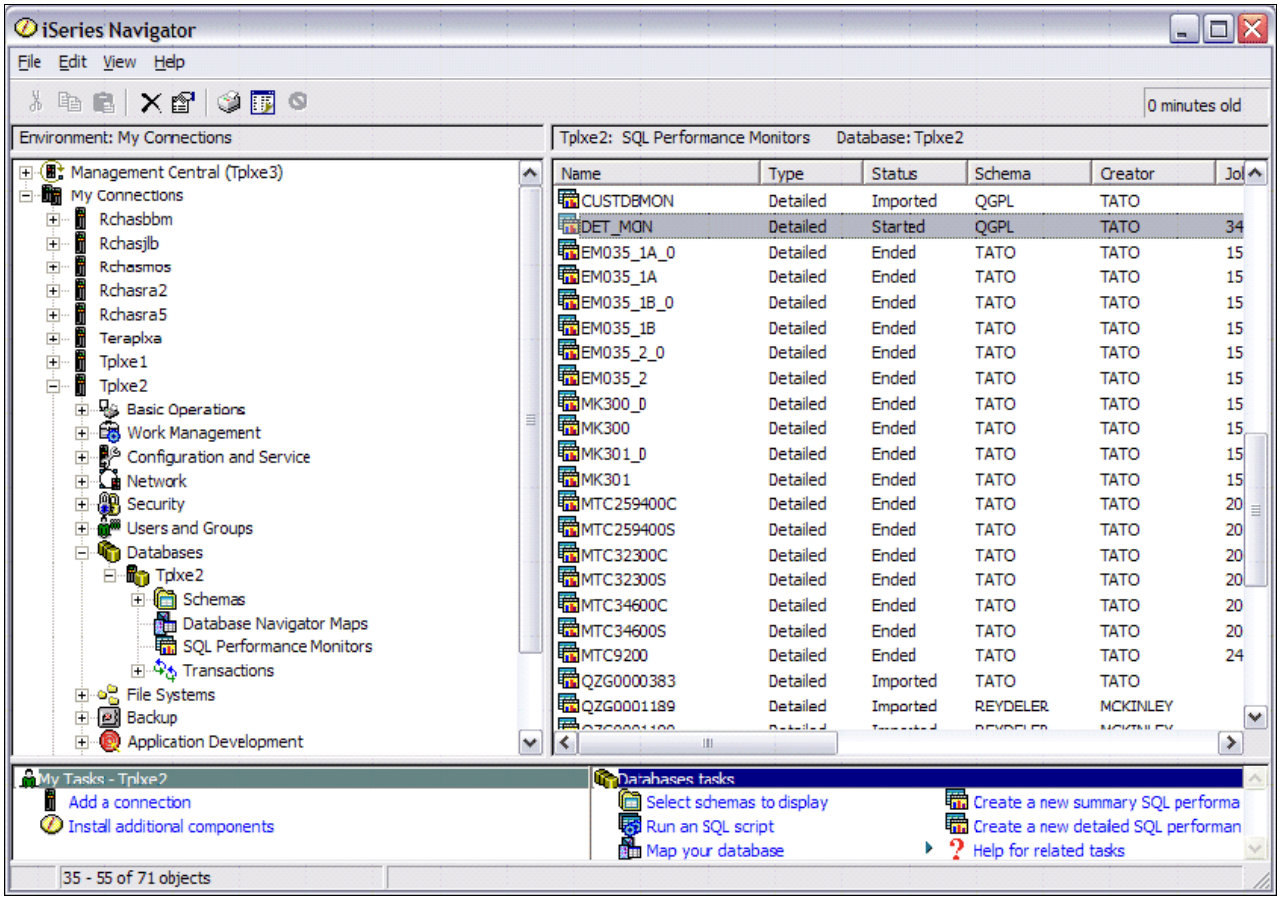


Figure 4-11 SQL Performance Monitor status pane

The monitoring of that job will continue for as long as it is on the system or until you end the monitor. To end a monitor, right-click the monitor and click the **End** option (see Figure 4-12). Then you can use this data to further analyze your query with the tools described in this redbook.

Unlike the Summary SQL Performance Monitor, you cannot pause a Detailed SQL Performance Monitor. However, you do not need to end the monitor to start analyzing the results. You can use the predefined reports to analyze the data as described in Chapter 5, “Analyzing database performance data using iSeries Navigator” on page 93. Or you can write your own queries to query the monitored data as described in Chapter 6, “Querying the performance data of the Database Monitor” on page 133.

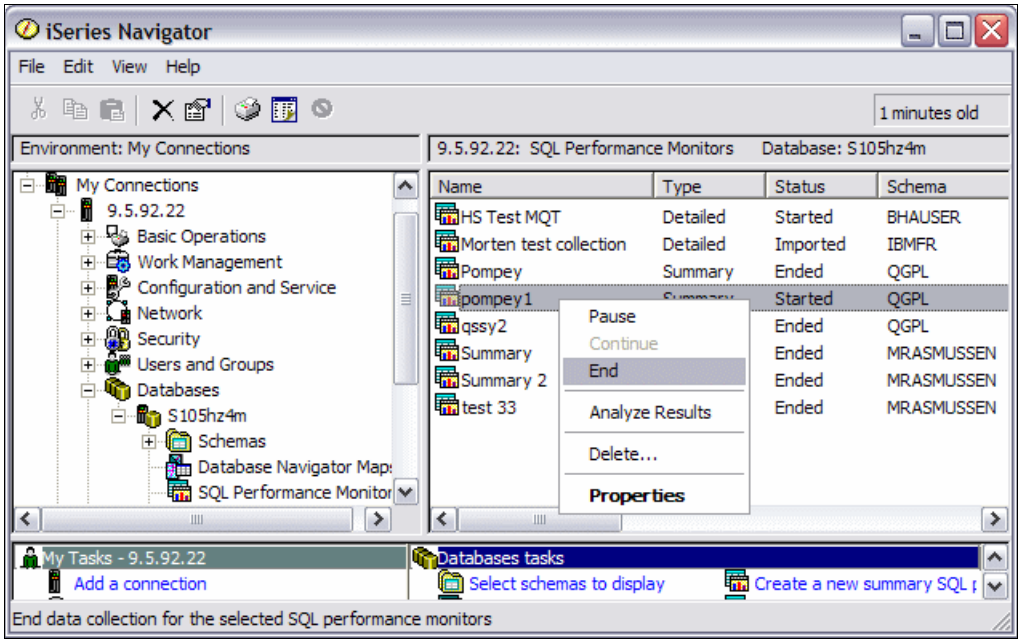


Figure 4-12 Ending an SQL Performance Monitor

Alternatively, you can start a Detailed SQL Performance Monitor from a Run SQL Script window in iSeries Navigator. In this case, the monitor is started only for the job associated with the Run SQL Script window. You can perform other functions from a Run SQL Script window, including to end, analyze, and delete the monitor, as well as to look at the monitor's properties as shown in Figure 4-13.

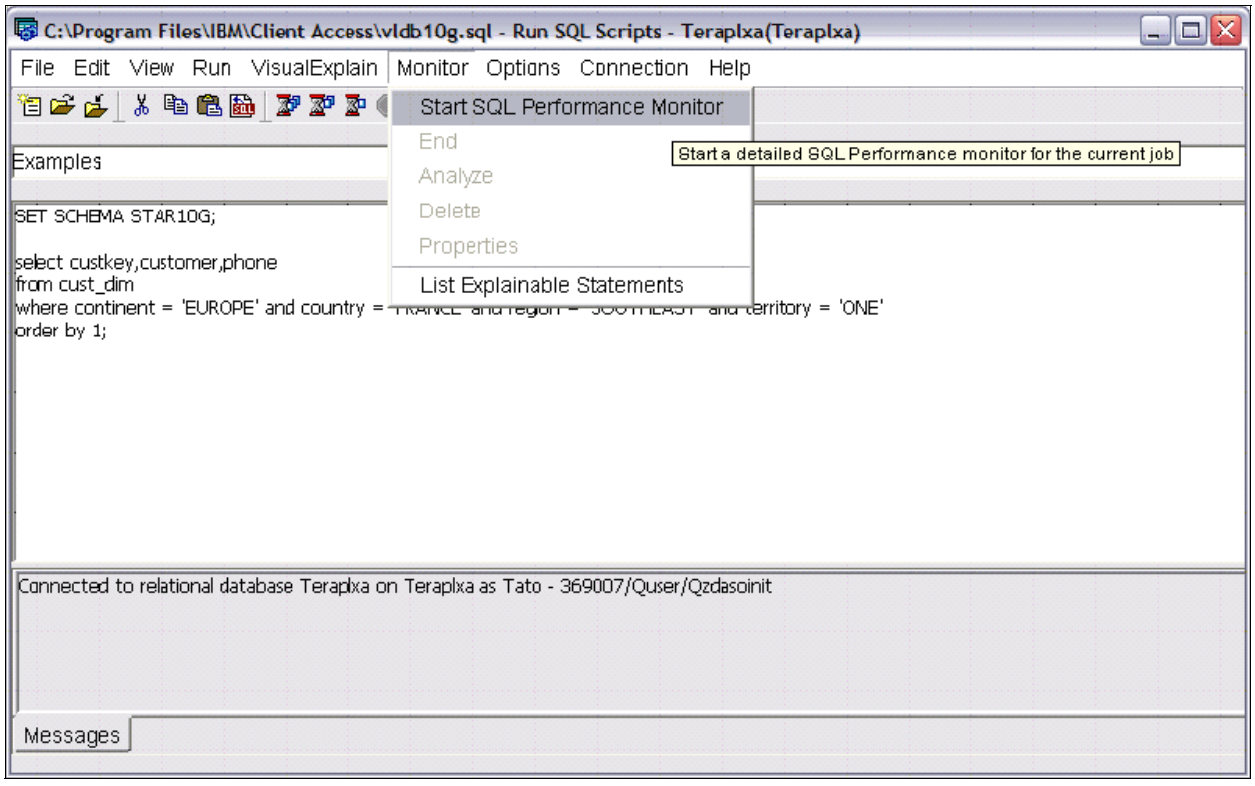


Figure 4-13 Start SQL Performance Monitor from the Run SQL Scripts window

When you end an SQL Performance Monitor, you cannot start it again, but you can continue with an SQL Performance Monitor (Summary) that was previously paused.

4.3.3 Importing Database Monitors into iSeries Navigator

In the Import SQL Performance Monitor Files window, you can incorporate data for SQL Performance Monitors that were started without using iSeries Navigator, or data that was collected on another system. To access this window, from the main iSeries Navigator window, right-click **SQL Performance Monitors** and select **Import**.

In the Import SQL Performance Monitor Files window (Figure 4-14), if you are importing data collected by a Memory Resident Database Monitor, for Type of monitor, select **Summary**. You must only specify the name of any one of the data files. Click **OK** to import the monitor.

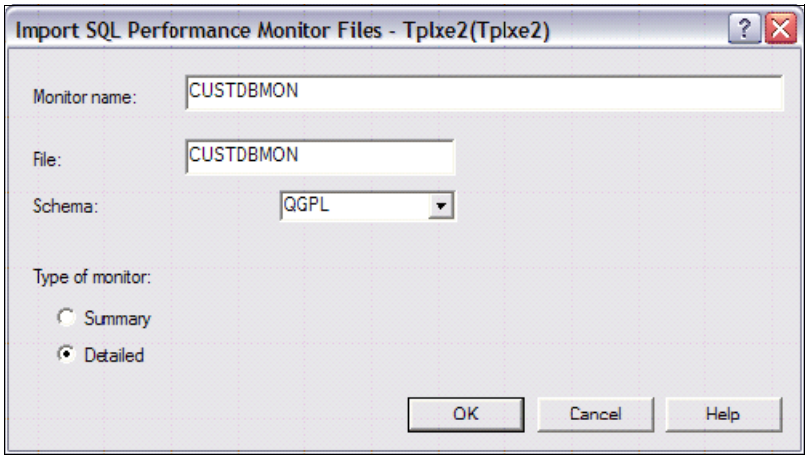


Figure 4-14 Import SQL Performance Monitor window

You should see the imported monitor in the list of monitors in the main iSeries Navigator window, with the status of *Imported*, as shown in Figure 4-15.

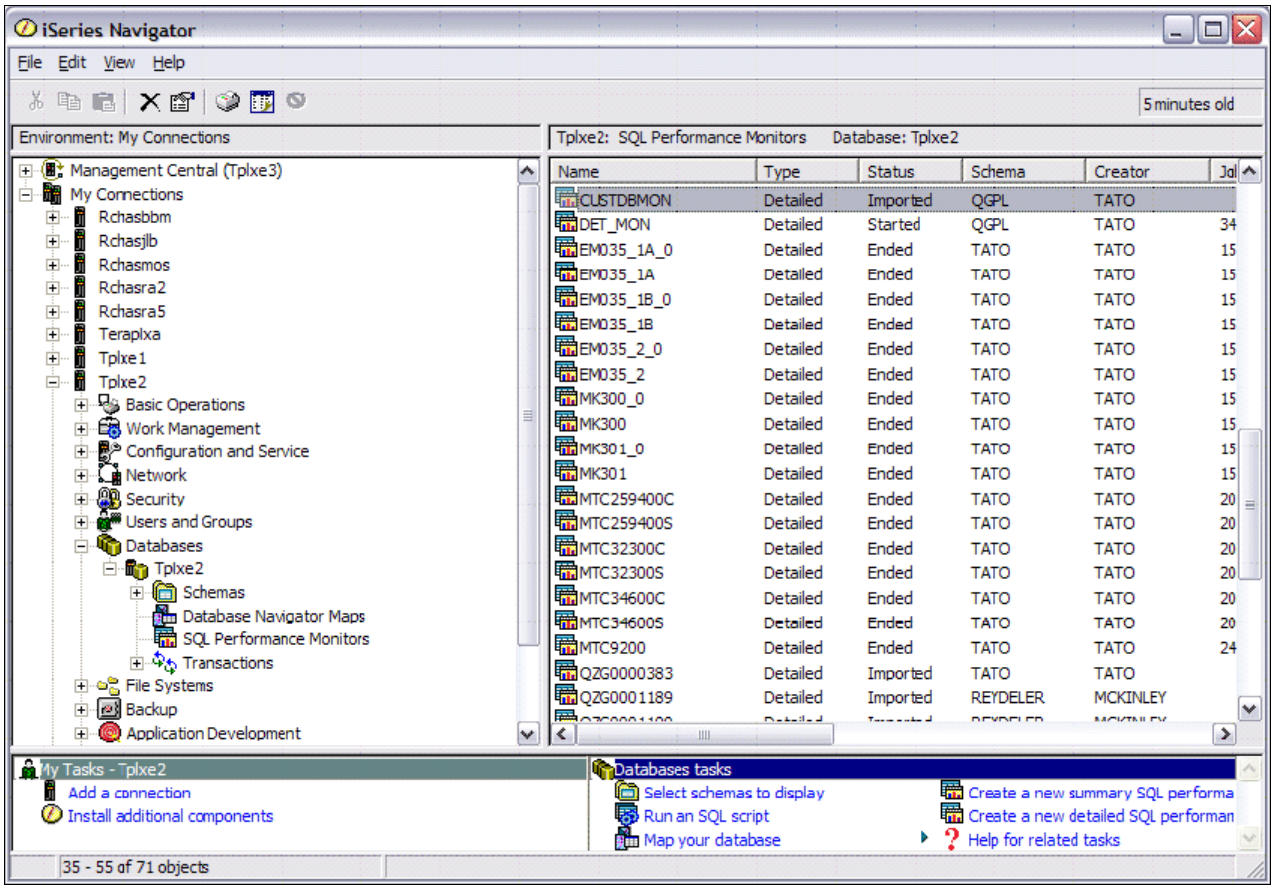


Figure 4-15 Imported monitors

4.4 SQL Performance Monitors properties

At any time, you can display the properties of a monitor in iSeries Navigator. Right-click the monitor for which you want to display the properties and select **Properties** as shown in Figure 4-16.

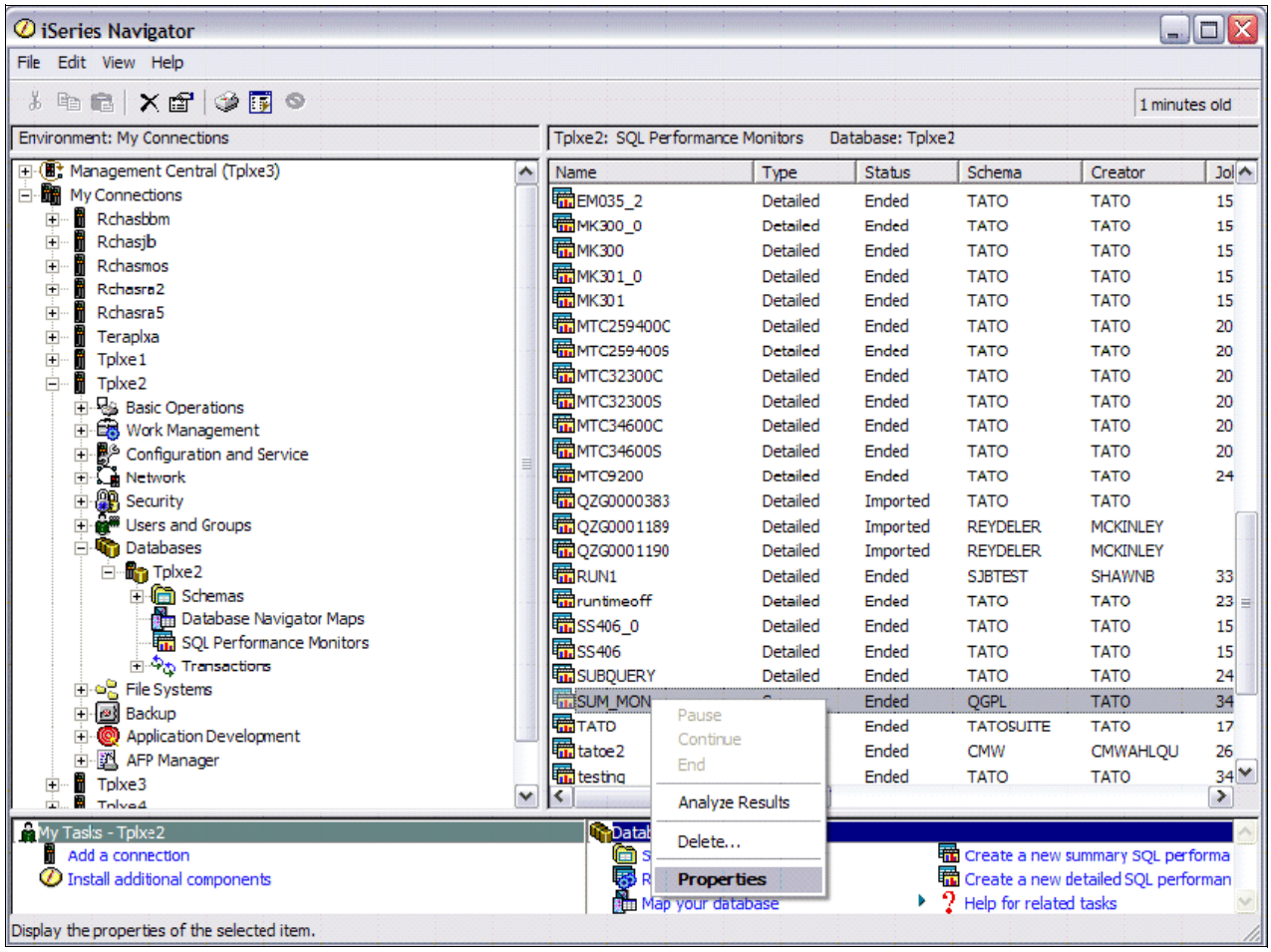


Figure 4-16 Selecting to view the properties of an SQL Performance Monitor

In the Properties window, click **General** tab (Figure 4-17) to view general information such as the monitor name, the schema to be saved, and the storage limit for the monitor. On this page, you also see the type of monitor (Summary or Detailed), the status, and the user profile that created the monitor.

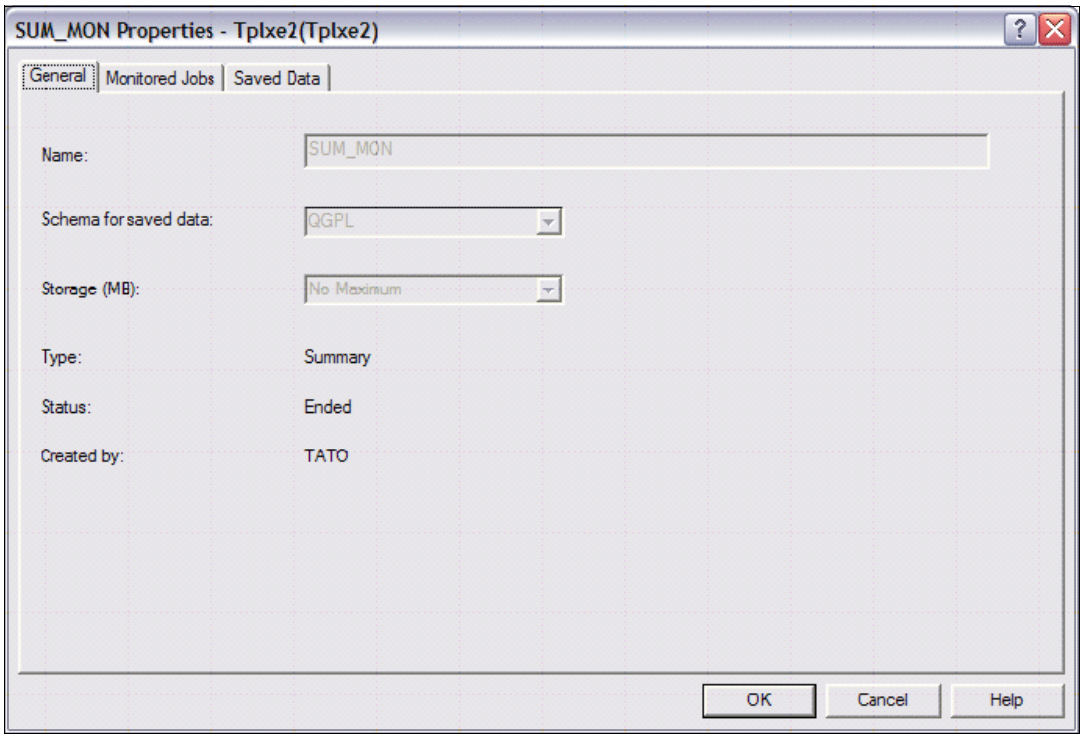


Figure 4-17 General tab of the Properties window

Click the **Monitored Jobs** tab (Figure 4-18), in the Properties window, to see the list of monitored jobs.

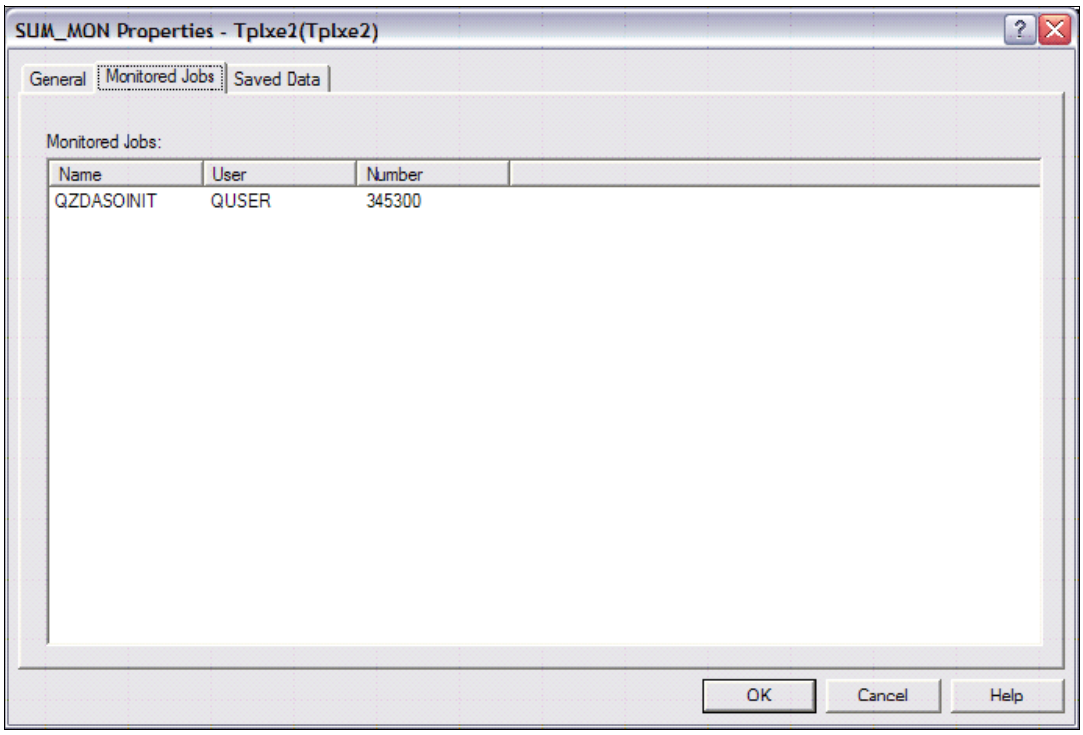


Figure 4-18 Monitored Jobs tab of the Properties window

Click the **Saved Data** tab (Figure 4-19) to see information based on the type of monitor:

- **Collection period**: Specifies the monitoring period for which you want to view data

  If an imported monitor is viewed, you see the statement "`Information Not Available`" in the Collection period box. The monitoring period can only be changed on the Properties window of the Summary Monitor.
- **Schema**: The schema where the monitor tables reside
- **Monitor data selected**: For a detailed SQL Performance Monitor, this section shows the name of the tables, for each one of the options selected when the monitor was started. The name of the tables start with the prefix QPM, followed by a sequential number. The tables have the same format as the Memory Resident Database Monitor tables (see "External table description" on page 63).

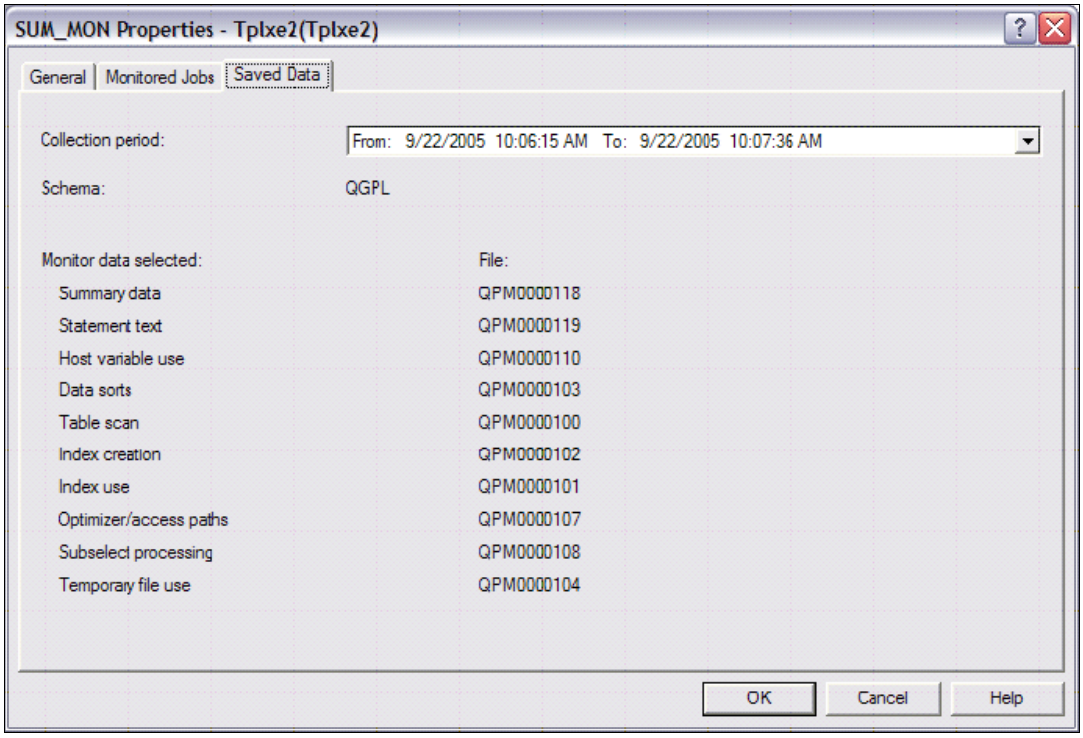


Figure 4-19 Saved Data tab of the Properties window for a Summary Monitor

For a detailed SQL Performance Monitor, you see the name of the table where the data is stored (Figure 4-20). The name of the tables starts with the prefix QZG, followed by a number assigned by the system. The table has the same format as when created with STRDBMON.

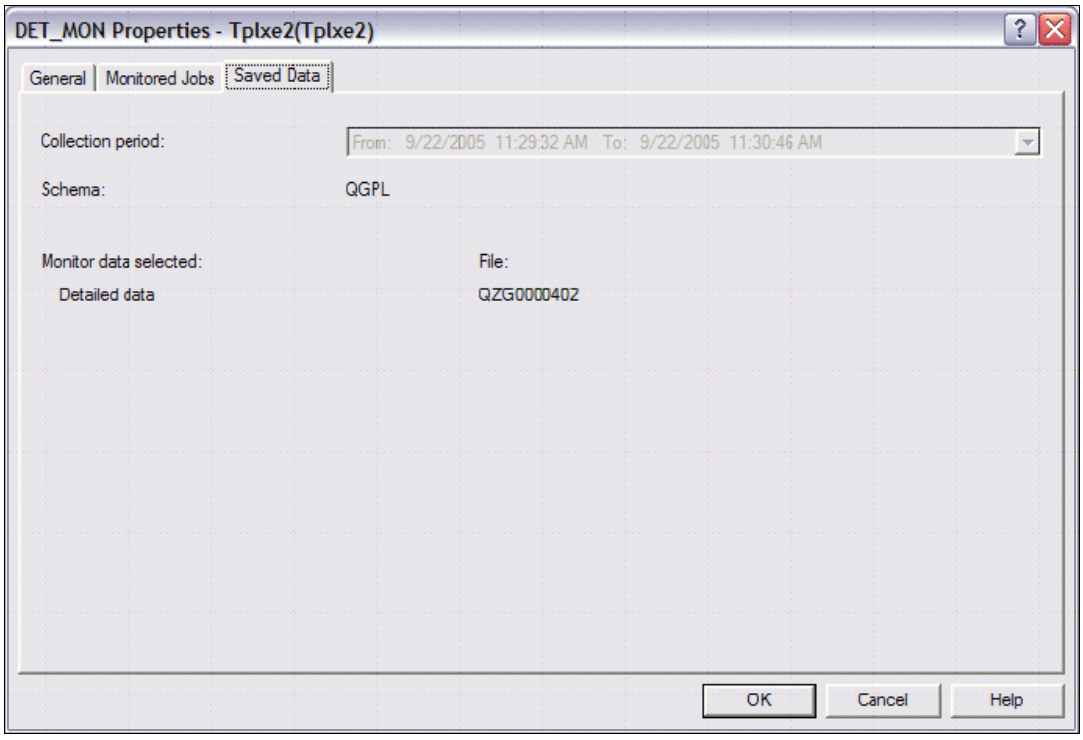


Figure 4-20 Saved Data tab for a Detailed Database Monitor

For monitors that were imported into iSeries Navigator, you see the name of the original monitor table that was used as the source table when the import took place (see Figure 4-21).

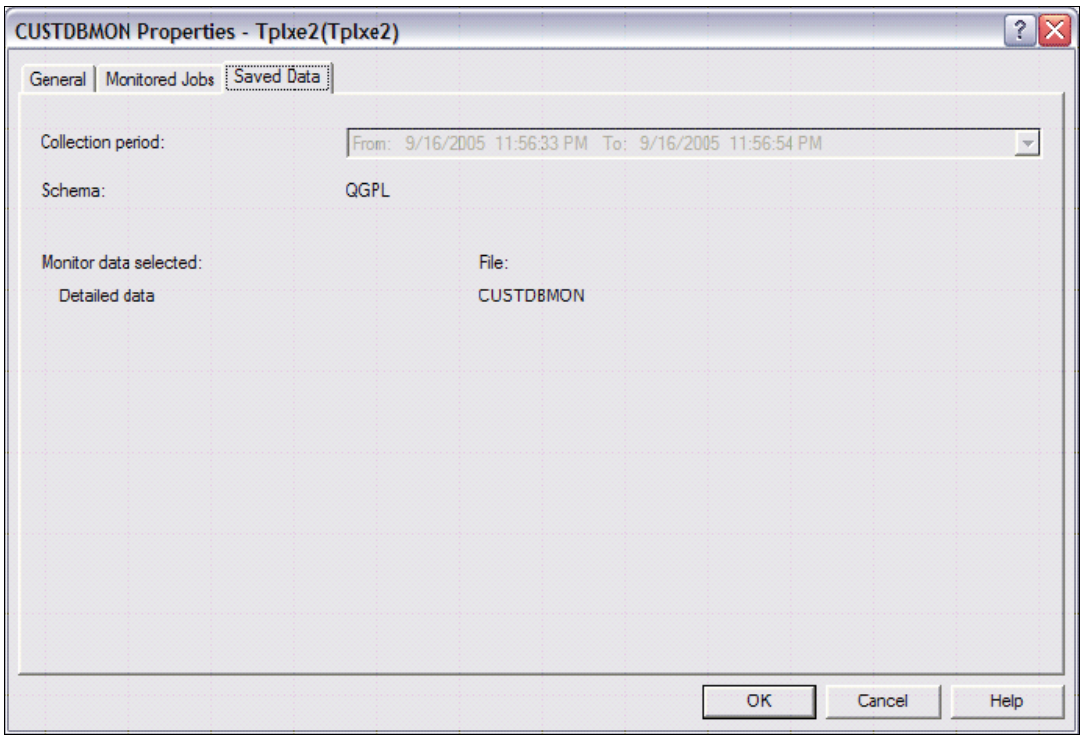


Figure 4-21 Saved Data tab of an Imported monitor

4.4.1 Considerations for the SQL Performance Monitors in iSeries Navigator

SQL Performance Monitors in iSeries Navigator only display the monitors that were created using iSeries Navigator (Summary and Detailed) or monitors that were imported. You can import monitor files that reside in different libraries on your system and that were created using other methods. You can use iSeries Navigator as the central application to work with all your monitors on the system. Moreover, you can take advantage of predefined reports to analyze the monitor data even for monitors that were imported into iSeries Navigator.

4.5 Summary or Detailed Database Monitor

There are circumstances in which it is better to collect a Summary SQL Performance Monitor than to collect a Detailed SQL Performance Monitor or vice versa. In this section, we explain when one is more useful than the other.

The Memory Resident Database Monitor has advantages over the Detailed Database Monitor created when using the STRDBMON command. The Memory Resident Database Monitor can be more useful than the Detailed Database Monitor in the following situations among others:

- When you are not sure what is causing the performance issue that you might be seeing

  You can collect a lot of data over the entire system in a short period of time using the Detailed Database Monitor, which causes a big overhead. It is better to collect a Summary Monitor over the entire system and analyze that.

- When you want to monitor the system over a set period of time to compare the results of one week against another, for example

  This helps with a proactive approach to SQL performance and with trends in the way that the database is performing.
- When the size of the data collected for a Summary Monitor started against the whole system is still significantly smaller than the data collected for a Detailed Monitor

  This can help the customer provide IBM Support with initial problem determination data.

A Detailed Database Monitor can be more useful when:

- You need more detailed information about your queries and not just summary data.
- You need to analyze non-SQL queries and see information through the 3019 detailed record.
- You want to see the implementation of your SQL queries and use tools, such as Visual Explain in iSeries Navigator, to explain the queries.
- You want to check for the generation of SQLCODE and SQLSTATE errors on your queries.

Important: The data collection in a Database Monitor is done inline within the job instead of a separate job.

Optimization records and data do not appear for queries that are already in reusable open data path (ODP) mode when the monitor is started. To ensure the capture of this data for a batch job, start the monitor before the job starts and collect it over the entire length of the job or as much as needed.

4.6 The Database Monitor record types

The Detailed Database Monitor collects different data and stores records in a single table in the order of occurrence. Within the Database Monitor table, each record contains a record type column. The Database Monitor uses the QQRID column to describe the type of information gathered in the particular record. Figure 4-22 shows the Database Monitor record types that are most often used for performance analysis.

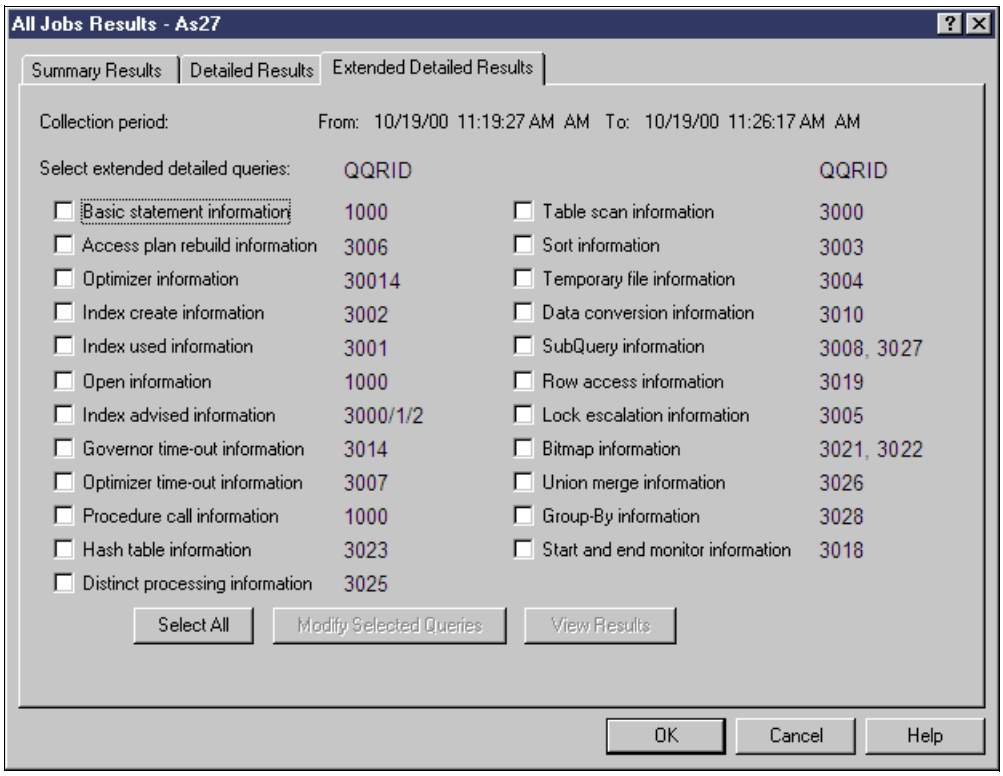


Figure 4-22 Detailed and Extended Detailed Results and their main source of information

Note: In the sections that follow, we identify the *columns* for each record type. The term "column" is also known as *field*.

4.6.1 Database Monitor record types

In this section, we list the Database Monitor record types that are most often used for performance analysis, as well as other record types. We also list the Global Database Monitor data columns and other columns that identify the tables or indexes used.

Record types most often used (QQRID value)

The following record types are most often used:

- **1000 Record**: SQL statement summary (see page 80)
- **3000 Record**: Arrival sequence (table scan; see page 82)
- **3001 Record**: Using existing index (see page 83)
- **3002 Record**: Temporary index created (page 85)
- **3003 Record**: Query sort (see page 86)
- **3004 Record**: Temporary file (see page 87)
- **3006 Record**: Access plan rebuild (see page 87)

- **3007 Record**: Summary optimization data (also known as *index optimization data*; see page 88)
- **3010 Record**: Host variable and ODP implementation (see page 89)
- **3021 Record**: Bitmap created
- **3022 Record**: Bitmap merge
- **3023 Record**: Temporal hash table created
- **3026 Record**: Union merge
- **3027 Record**: Subquery merge
- **3028 Record**: Grouping
- **3029 Record**: Index ordering

Database Monitor record types: Other record types

The following record types are in the Database Monitor table but are not used in all the records:

- **3005 Record**: Table locked
- **3008 Record**: Subquery processing
- **3014 Record**: Generic query information
- **3018 Record**: STRDBMON and ENDDBMON data
- **3019 Record**: Retrieved detail (only with *DETAIL)
- **3025 Record**: DISTINCT processing
- **3030 Record**: Query step processing
- **5002 Record**: SQL request executed by SQL Query Engine (SQE)

Record types 3000 to 3008, 3014, and 3021 to 3029 occur during a full open and can be referred to as *optimization records*. Optimization records are much like debug messages. These records are necessary to determine the access plan for any given query in the Database Monitor data.

Global Database Monitor data columns

The following data columns are common to all record types:

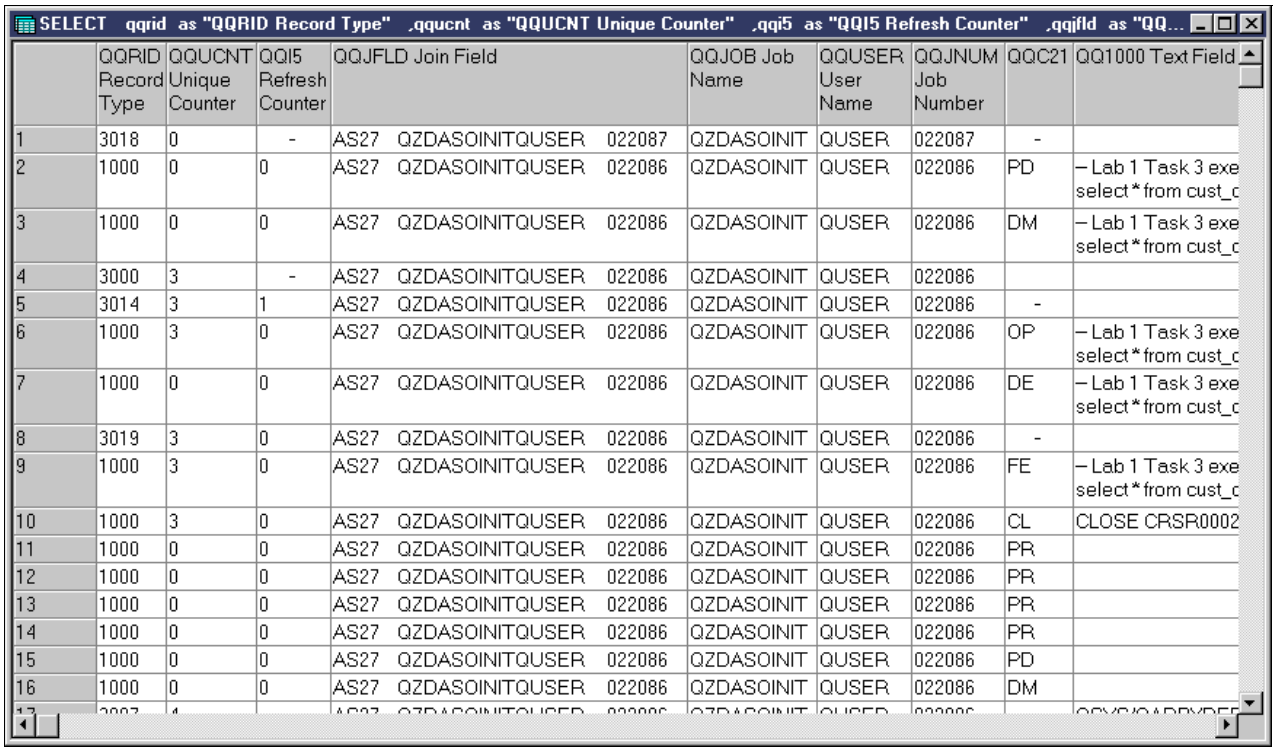
- **QQJOB** (job name)
- **QQUSER** (job user name)
- **QQJNUM** (job number): The job number is useful when multiple jobs are collected in one DB Monitor file.
- **QQJFLD** (join field): This column contains information that uniquely identifies a job and includes job name, job user name, and job number.
- **QQTIME** (time at which the record was created): The time record can be useful when trying to determine which queries were running in a given time period.
- **QQUCNT** (unique number given for each query within a job): QQUCNT links together all Database Monitor records associated with all instances of a unique query within a job. The QQUCNT value assigned at full open time stays constant for all subsequent instances of that query. Non-ODP SQL operations (prepare, describe, commit) have QQUCNT = 0 and cannot be linked to a query. But the QQ1000 column in the prepare or describe 1000 record contains the prepared SQL text.

  This data column is not set for optimization records.
- **QQI5** (refresh counter): This record specifies the instance number for a unique query. It is used in conjunction with the QQUCNT value to look at a specific instance of a query and is only valid on 3010 and 1000 SQL summary records.

  Non-ODP 1000 records (commit, prepare, and so on) have QQI5 = 0.
- **QQRID** (record identifier): This column identifies the type of record.

- **QVC102**: This column refers to the CURRENT job user name.
- **QQ19**: (thread identifier): This column might be useful for multithreaded applications.

Figure 4-23 shows some of the Global Database Monitor columns.



SELECT qqrid as "QQRID Record Type" ,qqucnt as "QQUCNT Unique Counter" ,qqi5 as "QQI5 Refresh Counter" ,qqjfld as "QQ...

	QQRID Record Type	QQUCNT Unique Counter	QQI5 Refresh Counter	QQJFLD Join Field	QQJOB Job Name	QQUSER User Name	QQJNUM Job Number	QQC21	QQ1000 Text Field
1	3018	0	-	AS27 QZDASOINITQUSER 022087	QZDASOINIT	QUSER	022087	-	
2	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	PD	-- Lab 1 Task 3 exe select * from cust_c
3	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	DM	-- Lab 1 Task 3 exe select * from cust_c
4	3000	3	-	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086		
5	3014	3	1	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	-	
6	1000	3	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	OP	-- Lab 1 Task 3 exe select * from cust_c
7	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	DE	-- Lab 1 Task 3 exe select * from cust_c
8	3019	3	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	-	
9	1000	3	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	FE	-- Lab 1 Task 3 exe select * from cust_c
10	1000	3	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	CL	CLOSE CRSR0002
11	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	PR	
12	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	PR	
13	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	PR	
14	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	PR	
15	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	PD	
16	1000	0	0	AS27 QZDASOINITQUSER 022086	QZDASOINIT	QUSER	022086	DM	

Figure 4-23 Global Database Monitor columns

Tip: If you are going to use the Run SQL Script in iSeries Navigator, change the JDBC setup to force translation for CCSID 65535, because the QQJFLD has been defined as FOR BIT DATA. Otherwise this column is shown in hexadecimal.

Reset the default value to use Visual Explain; otherwise it will fail.

Other columns that identify tables or indexes used

The following columns can also identify the tables or indexes that are used:

- **QQTLN**: Library of the table queried
- **QQTFN**: Name of the table queried
- **QQPTLN**: Base library
- **QQPTFN**: Base table
- **QQILNM**: Library of the index used
- **QQIFNM**: Name of the index used; *N when it is a temporary index
- **QVQTBL**: Queried table long name
- **QVQLIB**: Queried library long name
- **QVPTBL**: Base table long name
- **QVPLIB**: Base table library long name
- **QVINAM**: Index used long name
- **QVILIB**: Index library long name

How the data is organized in the Database Monitor table

The first occurrence of a unique query within the job always results in full open. A “unique” query is one that requires a new ODP. SQL has determined that there is no existing ODP that can be used.

The presence of optimization records indicates a full open for an Open, Select into, Update, Delete, or Insert operation. Optimization records are immediately followed by SQL summary records (QQRID=1000) for that operation.

Subsequent occurrences of this query within the same job either run in reusable ODP or nonreusable ODP mode. Nonreusable mode is indicated by the presence of optimization records each time a particular query is run (full open). Reusable ODP mode is indicated by 3010 and 1000 records each time the given query is run (no optimization records or full open).

Linking query instances in the Database Monitor data

The data in the Database Monitor file is arranged chronologically. This organization can make it difficult to find all instances of a unique query. Use the QQJFLD, QUCNT, and QQI5 columns to view specific query instances. Be aware of the following situations:

- QUCNT is assigned during a full open and is constant for all subsequent instances of a query.
- Non-ODP SQL operations (prepare, describe, commit) have QQUCNT = 0 and, therefore, cannot be linked to a query. If this is the case, you can use the QQ1000 column, which in the Prepare or Describe operation, contains the prepared SQL text.
- Non-ODP 1000 records (commit, prepare, and so on) have QQI5 = 0.

A full open occurs when there is an SQL update, insert, delete, or open operation and the QQI5 record is 0.

4.6.2 The 1000 Record: SQL statement summary

The 1000 record is the basic record type for any SQL query analysis. One record exists for each SQL operation (open, update, close, commit, and so on). The following columns are those that are most commonly used:

- **QQ1000**: Prepared text of SQL statement literals in the original SQL text might be replaced by parameter markers in prepared text if SQL was able to convert them during preparation (desired). For the original SQL text, use literal values from the matching 3010 record in the place of parameter markers or obtain the text from the step mode file using the QQSTIM time stamp from this record.
- **QQC21**: This column indicates the type of SQL operation (OP, FE, CL, UP, IN, DL, and so on). A value of MT in this column indicates the continuation of a record for SQL statements that exceed 1000 characters. A value of FE indicates a Fetch summary record and *not* the actual number of fetch operations.

  The ODP-related operation types are OP, IN, UP, DL, SI, and SK.
- **QQI2**: Number of rows updated, inserted, or deleted
- **QQI3**: Number of rows fetched (only on FE rows)

  This column indicates the actual number of rows fetched, and not the number of fetched attempts.

- **QQI6**: Elapsed time for this operation in microseconds

  The time to fetch all rows might not be included in with Open and Select operations; you must look at the time on Fetch operation rows.

Access plan information

The following columns indicate information about the access plan:

- **QQC103** and **QQC104**: Package or program name and library
- **QVC18**: Dynamic SQL statement type

  A value of E represents “extended dynamic”, a value of S represents “system wide” statement cache, and a value of L represents a prepared statement.
- **8** and **QVC22**: Access plan rebuild code and subcode

  This subcode is useful for IBM debug purposes.
- **QVC24**: Access plan save status

  A value of Ax means that the access plan could not be saved. Values of Bx or a blank mean that the access plan was saved successfully.

ODP information

The following columns provide information about the ODP:

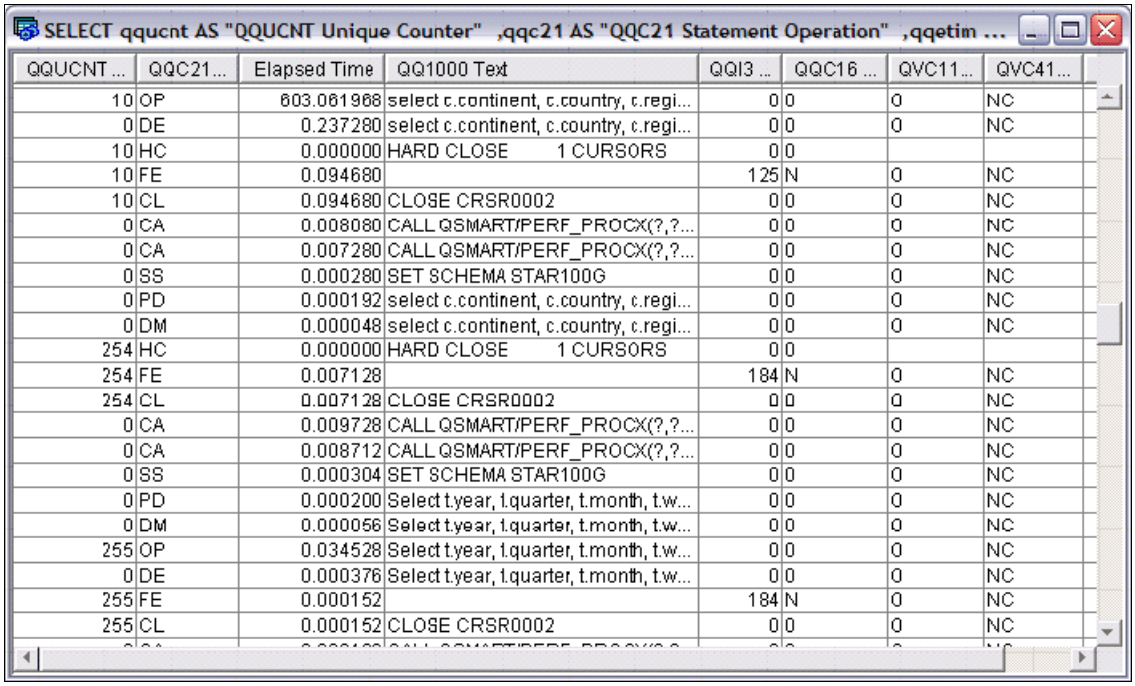
- **QQI5**: Query instance counter, where a value of 0 means that a full open occurred
- **QQC15**: Hard close reason code (for an HC operation type)
- **QVC12**: Pseudo open indicator
- **QVC13**: Pseudo close indicator
- **QQC181** and **QQC182**: Cursor and statement name

Example 4-6 shows a query with some of the most commonly used columns in the 1000 record.

Example 4-6 Common columns in the 1000 record

```
SELECT qqucnt AS "QQUCNT Unique Counter"
,qqc21 AS "QQC21 Statement Operation"
,qqetim - qqstim AS "Elapsed Time"
,qq1000 AS "QQ1000 Text"
,qqi3 AS "QQI3 Fetched Rows"
,qqc16 AS "QQC16 Data Conv."
,qvc11 AS "QVC11 ALWCPYDTA"
,QVC41 AS "QVC41 Cmt Control Lvl"
,rrn(a)
FROM vldbmon a
WHERE qqrid = 1000
ORDER BY rrn(a)
OPTIMIZE FOR ALL ROWS;
```

Figure 4-24 shows the result of the query in Example 4-6. The first row shows that the Open operation took 603 seconds and only 0.09 seconds to fetch 125 rows. The query ran with ALWCPYDTA *OPTIMIZE and with commitment control set to *NONE. Also, there were no conversion problems when the query was run.


SELECT qqucnt AS "QQUCNT Unique Counter" ,qqc21 AS "QQC21 Statement Operation" ,qqetim ...

QQUCNT ...	QQC21...	Elapsed Time	QQ1000 Text	QQI3 ...	QQC16 ...	QVC11...	QVC41...
10	OP	603.061968	select c.continent, c.country, c.regi...	0	0	0	NC
0	DE	0.237280	select c.continent, c.country, c.regi...	0	0	0	NC
10	HC	0.000000	HARD CLOSE 1 CURSORS	0	0		
10	FE	0.094680		125	N	0	NC
10	CL	0.094680	CLOSE CRSR0002	0	0	0	NC
0	CA	0.008080	CALL QSMART/PERF_PROCX(?,?...	0	0	0	NC
0	CA	0.007280	CALL QSMART/PERF_PROCX(?,?...	0	0	0	NC
0	SS	0.000280	SET SCHEMA STAR100G	0	0	0	NC
0	PD	0.000192	select c.continent, c.country, c.regi...	0	0	0	NC
0	DM	0.000048	select c.continent, c.country, c.regi...	0	0	0	NC
254	HC	0.000000	HARD CLOSE 1 CURSORS	0	0		
254	FE	0.007128		184	N	0	NC
254	CL	0.007128	CLOSE CRSR0002	0	0	0	NC
0	CA	0.009728	CALL QSMART/PERF_PROCX(?,?...	0	0	0	NC
0	CA	0.008712	CALL QSMART/PERF_PROCX(?,?...	0	0	0	NC
0	SS	0.000304	SET SCHEMA STAR100G	0	0	0	NC
0	PD	0.000200	Select t.year, t.quarter, t.month, t.w...	0	0	0	NC
0	DM	0.000056	Select t.year, t.quarter, t.month, t.w...	0	0	0	NC
255	OP	0.034528	Select t.year, t.quarter, t.month, t.w...	0	0	0	NC
0	DE	0.000376	Select t.year, t.quarter, t.month, t.w...	0	0	0	NC
255	FE	0.000152		184	N	0	NC
255	CL	0.000152	CLOSE CRSR0002	0	0	0	NC

Figure 4-24 Common columns in the 1000 record

4.6.3 The 3000 Record: Arrival sequence (table scan)

The 3000 record points outs queries in which the entire table is scanned. A table scan is generally acceptable in cases where a large portion of the table will be selected or the table has a small number of rows.

Table information

Table information is provided by the following columns:

- **QVQTBL** and **QVQLIB**: Table name and schema name respectively
- **QQPTFN** and **QQPTLN**: Short system table name and library name respectively
- **QQTOTR**: Number of rows in table

Query optimization details

The following columns provide details about query optimization:

- **QQRCOD**: Reason code, why table scan chosen
- **QQIDXA**: Index advised (Y or N)

  If the value is N, QQI2 and QQIDXD will not contain data.

- **QQI2**: Number of primary (key positioning) keys in QQIDXD column
- **QQIDXD**: Suggested keys for index (selection only)

  This column can contain both primary and secondary keys. Starting from the left, QQI2 indicates the number of keys that are considered primary.

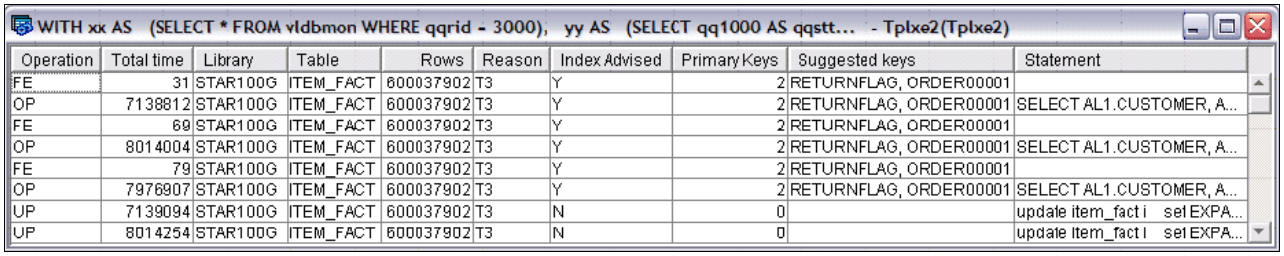
The query shown in Example 4-7 illustrates some of the most commonly used columns in the 3000 record.

Example 4-7 Common columns in the 3000 record

```
WITH xx AS  (SELECT * FROM vldbmon WHERE qqrid = 3000),
yy AS
(SELECT qq1000 AS qqsttx
,qqjfld ,qqucnt ,qqc21 as qqop ,qqi4 as qqtt FROM vldbmon
WHERE qqrid = 1000 AND qqc21 <> 'MT'
AND (qvc1c = 'Y' OR (qqc21 IN('DL', 'UP')
AND qqc181 <= ' ') OR qqc21 IN ('IN', 'IC', 'SK', 'SI') OR qqc21 LIKE 'O%'))

SELECT qqop as "Operation" ,qqtt as "Total time" ,qqptln as "Library"  ,qqptfn as "Table"
,qqtotr as "Rows"  ,qqrcod as "Reason"  ,qqidxa as "Index Advised"  ,qqi2  as "Primary
Keys"  ,qqidxd as "Suggested keys" ,qqsttx as "Statement" FROM xx a  LEFT JOIN yy b  ON
a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt ORDER BY qqidxa DESC;
```

Figure 4-25 shows the result of the query in Example 4-7. The information from the 3000 record is joined with information from the 1000 record, to determine which queries did a table scan. Therefore, in a case where an index is not recommended, we can still look at the selection in the SQL text to see if a good index can be created.


WITH xx AS (SELECT * FROM vldbmon WHERE qqrid = 3000), yy AS (SELECT qq1000 AS qqstt... - Tplxe2(Tplxe2)

Operation	Total time	Library	Table	Rows	Reason	Index Advised	Primary Keys	Suggested keys	Statement
FE	31	STAR100G	ITEM_FACT	600037902	T3	Y	2	RETURNFLAG, ORDER00001	
OP	7138812	STAR100G	ITEM_FACT	600037902	T3	Y	2	RETURNFLAG, ORDER00001	SELECT AL1.CUSTOMER, A...
FE	69	STAR100G	ITEM_FACT	600037902	T3	Y	2	RETURNFLAG, ORDER00001	
OP	8014004	STAR100G	ITEM_FACT	600037902	T3	Y	2	RETURNFLAG, ORDER00001	SELECT AL1.CUSTOMER, A...
FE	79	STAR100G	ITEM_FACT	600037902	T3	Y	2	RETURNFLAG, ORDER00001	
OP	7976907	STAR100G	ITEM_FACT	600037902	T3	Y	2	RETURNFLAG, ORDER00001	SELECT AL1.CUSTOMER, A...
UP	7139094	STAR100G	ITEM_FACT	600037902	T3	N	0		update item_fact i set EXPA...
UP	8014254	STAR100G	ITEM_FACT	600037902	T3	N	0		update item_fact i set EXPA...

Figure 4-25 Common columns in the 3000 record

4.6.4 The 3001 Record: Using an existing index

The 3001 record shows the index that will be used to access the table and why it was chosen. If the index was chosen for a join operation, additional information is given to help determine how the table “fits” in the join. The order of the 3001 records indicates the join order chosen by the optimizer.

Index and table information

The following columns provide index and table information:

- **QVINAM** and **QVILIB**: Name of the chosen index and library
- **QVQTBL** and **QVQLIB**: Name of the associated table and library

Query optimization details

The following columns indicate details about query optimization:

- **QQRCOD***:* Reason the index was selected

  I1 Selection only
  I2 Ordering or grouping
  I3 Selection and ordering or grouping
  I4 Nested loop join
  I5 Record selection using bitmap

- **QQIDXA**: Index advised (Y or N)
- **QQI2**: Number of primary (key positioning) keys in QQIDXD column
- **QQIDXD**: Suggested keys for index (selection only)
- **QVC14**: Index only access indicator (Y or N)
- **QQIA**: Index page size

Note: A combination of the 3000 and 3001 records for a table indicates bitmap processing.

Example 4-8 shows a query with some of the most commonly used column in the 3001 record.

Example 4-8 Common columns in the 3001 record

```
SELECT qqilnm as "Library"  ,qqifnm as "Table"  ,qqtotr as "Rows"  ,qqrcod as "Reason"
,qqidxa as "Index Advised"  ,qqi2  as "Primary Keys"  ,qqidxd as "Suggested keys"
,qvc14 as "Index Only"
,qqia as "Index page size"
FROM vldbmon a
WHERE qqrid = 3001
order by qqidxa desc;
```

Figure 4-26 shows the result of the query in Example 4-8. It shows the following information:

- ITEM_00003 was used for ordering and grouping (reason I2). However, the optimizer is advising an index with two primary keys most likely for selection.
- Some indexes are temporary and were created to satisfy the join in the query (reason I4).
- Some indexes were used for row selection (reason I1) or row selection and ordering or grouping (reason I3).
- For some indexes, the data was retrieved from the index, without accessing the table (QVC14 column).

SELECT qqilnm as "Library" ,qqifnm as "Table" ,qqtotr as "Rows" ,qqrcod as "Reason" ... - Tplxe2(Tplxe2)

Library	Table	Rows	Reason	Index Advised	Primary Keys	Suggested keys	Index Only	Index page size
STAR100G	ITEM_00003	600037902	I2	Y	2	SHIPDATE, RETURNFLAG	N	65536
STAR100G	ITEM_00003	600037902	I2	Y	2	SHIPDATE, RETURNFLAG	N	65536
STAR100G	CUST_00001	15000000	I4	N	0		N	65536
*N	*TEMPX0001	15000000	I4	N	0		N	65536
*N	*TEMPX0001	1450	I4	N	0		N	65536
STAR100G	CUST_00001	15000000	I4	N	0		N	65536
STAR100G	CUST_00003	15000000	I3	N	0		Y	65536
STAR100G	CUST_00003	15000000	I3	N	0		Y	65536
STAR100G	CUST_00001	15000000	I4	N	0		N	65536
*N	*TEMPX0001	1450	I4	N	0		N	65536
STAR100G	CUST_00001	15000000	I4	N	0		N	65536
*N	*TEMPX0001	15000000	I4	N	0		N	65536
*N	*TEMPX0001	1450	I4	N	0		N	65536
STAR100G	CUST_00001	15000000	I4	N	0		N	65536
*N	*TEMPX0001	15000000	I4	N	0		N	65536
STAR100G	TIME_00002	1450	I1	N	0		Y	65536
STAR100G	TIME_00002	1450	I1	N	0		Y	65536
STAR100G	TIME_00002	1450	I1	N	0		N	65536
STAR100G	ITEM_00003	600037902	I1	N	0		N	65536
STAR100G	TIME_00002	1450	I1	N	0		Y	65536
STAR100G	TIME_00002	1450	I1	N	0		Y	65536
STAR100G	TIME_00002	1450	I1	N	0		Y	65536
STAR100G	TIME_00002	1450	I1	N	0		N	65536
STAR100G	ITEM_00003	600037902	I1	N	0		N	65536

Figure 4-26 Common columns from the 3001 record

4.6.5 The 3002 Record: Temporary index created

The 3002 record shows instances in which the database optimizer decided that existing indexes are too costly or do not have the right key order for join, group by, or order by clauses. For this reason, the optimizer might decide to create a temporary index (Classic Query Engine (CQE)).

Index and table information

The following columns provide information about the index and table:

- **QVQTBL** and **QVQLIB**: Table name for which the index is built
- **QQRCOD**: Reason why the index build was done

  I2 Ordering or grouping
  I3 Selection and ordering or grouping
  I4 Nested loop join

- **QQTOTR**: Number of rows in the table
- **QQRIDX**: Number of entries in the temporary index
- **QQSTIM**: Time stamp for the start of the index build
- **QQETIM**: Time stamp for the end of the index build
- **QQ1000**: Name of the columns used for the index keys

  Column names are the “short” column names.

Example 4-9 shows a query with some of the most commonly used columns in the 3002 record.

Example 4-9 Common columns in the 3002 record

```
SELECT qqptln as "QQPTLN Library"
,qqptfn as "QQPTFN Table"
,qqrcod as "QQRCOD Reason"
,qqtotr as "QQTOTR Rows"
,qqridx as "QQRIDX Entries in Temp Idx"
,qqetim - qqstim as "Idx Build Time"
,qq1000 as "QQ1000 Index Created Key"
,qqidxa as "QQIDXA Index Advised"
,qqi2 as "QQI2 Nbr of Primary Key"
,qqidxd as "QQIDXD Index Advised"
,qvc16
FROM vldbmon
WHERE QQRID = 3002
ORDER BY "Idx Build Time" desc;
```

Figure 4-27 shows the result of the query in Example 4-9. It shows the following information:

- In the data sampled, we found some indexes that were created for a join were recognized by reason code (QQRCOD) I4. Others were created for ordering or grouping (reason code I2).
- No indexes were advised.
- Some rows show an index from an index. Notice the presence of information in columns QQILNM and QQIFNM and the value on QVC16 (index from an index).
- Three rows show an index with a mapped key.

SELECT qqptln as "QQPTLN Library" ,qqptfn as "QQPTFN Table" ,qqilnm as "QQILNM Lib for Idx... - Tplxe2(Tplxe2)

QQPTLN...	QQPTFN...	QQILNM...	QQIFNM ...	QQRCOD...	QQTOTR...	QQRIDX...	Idx Build Time	QQ1000 Index Created Key	QQIDXA Index Advised	QQI2...	QQIDXD...	QVC16
STAR10G	CUST_DIM			I4	1500000	1500000	4.083728	CUSTKEY ASCEND	N	0		N
STAR10G	CUST_DIM			I4	1500000	1500000	4.073400	CUSTKEY ASCEND	N	0		N
*N	*N			I2	15551	15551	0.247272	*MAP ASCEND	N	0		N
STAR100G	TIME_DIM	STAR100G	TIME_00003	I4	1450	31	0.162944	DATEKEY ASCEND	N	0		Y
*N	*N			I2	1183	1183	0.106112	*MAP ASCEND	N	0		N
STAR100G	CUST_DIM	STAR100G	CUST_00002	I4	15000000	0	0.071312	CUSTKEY ASCEND	N	0		Y
STAR100G	CUST_DIM	STAR100G	CUST_00002	I4	15000000	0	0.068192	CUSTKEY ASCEND	N	0		Y
STAR100G	TIME_DIM	STAR100G	TIME_00003	I4	1450	31	0.066568	DATEKEY ASCEND	N	0		Y
*N	*N			I2	1183	1183	0.041592	*MAP ASCEND	N	0		N
STAR100G	CUST_DIM	STAR100G	CUST_00002	I4	15000000	0	0.035016	CUSTKEY ASCEND	N	0		Y
STAR100G	TIME_DIM	STAR100G	TIME_00003	I4	1450	31	0.027088	DATEKEY ASCEND	N	0		Y

Figure 4-27 Common columns in the 3002 record

4.6.6 The 3003 Record: Query sort

The 3003 record shows that the database optimizer has decided to put selected rows into a temporary space and sort them. This is either cheaper than alternative indexing methods or an operation is forcing the optimizer to do so. An example is a UNION or an ORDER BY from the columns of several tables.

The following columns are the most commonly used:

- **QQSTIM**: Time stamp for the start of the refill and sort
- **QQETIM**: Time stamp for the end of the refill and sort
- **QQRCOD** and **QQI7**: Reason why a sort technique was chosen
- **QQRSS**: The number of rows in a sort space

Keep in mind that sorting might increase the open time and cost since sorting is often performed at open time. If a number rows sorted is small, then adding the right index might improve performance. Indexes can still be used to select or join records before the sort occurs. This does not indicate that the ODP is nonreusable.

The 1000 SQL summary record for the open might have a high elapsed time (QQI6 or QQI4). Sort buffers are refilled and sorted at open time, even in reusable ODP mode. However, high elapsed times might indicate a large answer set. In this case, the sort outperforms index usage (the situation in most cases).

4.6.7 The 3004 Record: Temporary file

The 3004 record shows that the database optimizer is forced to store intermediate results and rows in a temporary file because of the nature of the query. Examples are group by columns from more than one file or materializing view results.

The following columns are the most commonly used:

- **QQSTIM**: Time stamp for the start of a fill temporary results table
- **QQETIM**: Time stamp for the end of a fill temporary results table
- **QQTMPR**: Number of rows in a temporary table
- **QQRCOD**: Reason for building temporary index

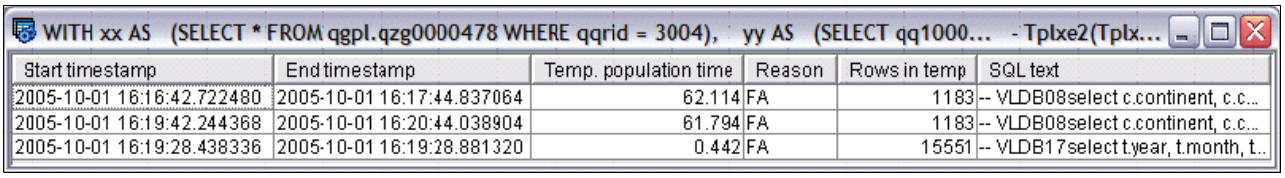
Example 4-10 shows a query with some of the most commonly used columns in the 3004 record.

Example 4-10 Common columns in the 3004 record

```
WITH xx AS
(SELECT * FROM qgpl.qzg0000478 WHERE qqrid = 3004),
yy AS
(SELECT qq1000 AS qqsttx, qqjfld, qqucnt FROM qgpl.qzg0000478  WHERE qqrid = 1000 AND qqc21
<> 'MT' AND (qvc1c = 'Y' OR (qqc21 IN('DL', 'UP')
AND qqc181 <= ' ') OR qqc21 IN ('IN', 'IC', 'SK', 'SI') OR qqc21 LIKE 'O%'))

SELECT  qqstim as "Start timestamp" ,qqetim as "End timestamp",
DECIMAL((DAY(qqetim-qqstim)*24*3600)+
(HOUR(qqetim-qqstim)*3600)+(MINUTE(qqetim-qqstim)*60)+
(SECOND(qqetim-qqstim))+(MICROSECOND(qqetim-qqstim)*.000001),18,3) AS "Temp. population
time" ,qqrcod as "Reason" ,qqtmpr as "Rows in temp" ,qqsttx as "SQL text" FROM xx a  LEFT
JOIN yy b  ON a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt  ORDER BY "Temp. population time"
DESC;
```

Figure 4-28 shows the result of the query in Example 4-10. The information from the 3004 record is joined with information from the 1000 record so we know which queries were implemented using a temporary result file. The reason in the example is that the query contains a join condition that requires a temporary table (reason FA).



Start timestamp	End timestamp	Temp. population time	Reason	Rows in temp	SQL text
2005-10-01 16:16:42.722480	2005-10-01 16:17:44.837064	62.114	FA	1183	-- VLDB08select c.continent, c.c...
2005-10-01 16:19:42.244368	2005-10-01 16:20:44.038904	61.794	FA	1183	-- VLDB08select c.continent, c.c...
2005-10-01 16:19:28.438336	2005-10-01 16:19:28.881320	0.442	FA	15551	-- VLDB17select t.year, t.month, t...

Figure 4-28 Common columns in the 3004 record

4.6.8 The 3006 Record: Access plan rebuild

The 3006 record is *not* present on every full open. It is generated only when an access plan previously existed and has to be rebuilt. Also, the 3006 record is not generated when SQE switches between cached access plans (up to three) in the SQE plan cache for an SQL statement.

The following columns are the most commonly used:

- **QQRCOD**: Rebuild reason code
- **QQC21**: Rebuild reason code subtype (IBM debug purposes)
- **QVC22**: Previous rebuild code

- **QVC23**: Previous rebuild code subtype
- **QQC11**: Plan required optimization

  Y Plan had to be re optimized & rebuilt
  N QAQQINI Re optimize option prevent access plan rebuild

  The 1000 row contains an indicator regarding whether the rebuilt access plan can be saved (QVC24).

Example 4-11 shows a query with some of the most commonly used columns in the 3006 record.

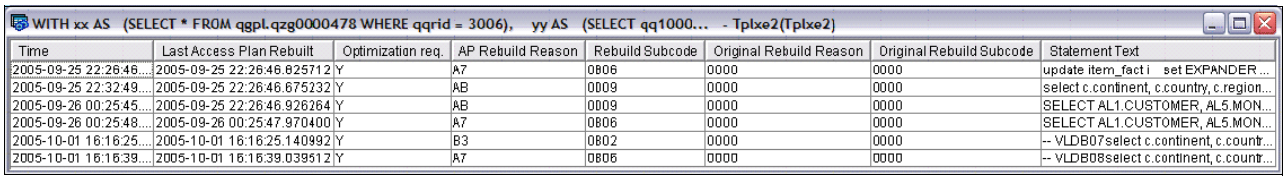
Example 4-11 Common columns in the 3006 record

```
WITH xx AS
(SELECT * FROM qgpl.qzg0000478 WHERE qqrid = 3006),
yy AS
(SELECT qq1000 AS qqsttx, qqstim, qqetim, qqjfld, qqucnt  FROM qgpl.qzg0000478 WHERE qqrid
= 1000 AND qqc21 <> 'MT'
AND (qvc1c = 'Y' OR (qqc21 IN('DL', 'UP')
AND qqc181 <= ' ') OR qqc21 IN ('IN', 'IC', 'SK', 'SI') OR qqc21 LIKE 'O%'))

SELECT  qqtime AS "Time", qqtim1 AS "Last Access Plan Rebuilt", qqc11 AS "Optimization
req.",
qqrcod AS "AP Rebuild Reason", HEX(qqc21) AS "Rebuild Subcode", HEX(qvc22) AS "Original
Rebuild Reason", HEX(qvc23) AS "Original Rebuild Subcode", varchar(qqsttx,20000) AS
"Statement Text"
FROM xx a LEFT JOIN yy b  ON a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt  WHERE (a.qqtime
BETWEEN b.qqstim AND b.qqetim OR b.qqstim IS NULL)  ORDER BY "Time";
```

Figure 4-29 show the result of the query in Example 4-11. The information from the 3006 record is joined with information from the 1000 record, so we know which queries had access plans rebuilt. The example shows three reasons:

- The access plan was rebuilt because of system programming changes (reason A7).
- The storage pool changed, or the DEGREE parameter of CHGQRYA command changed (reason AB).
- This is the first run of the query after a Prepare operation. That is, it is the first run with real actual parameter marker values.


WITH xx AS (SELECT * FROM qgpl.qzg0000478 WHERE qqrid = 3006), yy AS (SELECT qq1000... - Tplxe2(Tplxe2)

Time	Last Access Plan Rebuilt	Optimization req.	AP Rebuild Reason	Rebuild Subcode	Original Rebuild Reason	Original Rebuild Subcode	Statement Text
2005-09-25 22:26:46....	2005-09-25 22:26:46.825712	Y	A7	0B06	0000	0000	update item_fact i set EXPANDER ...
2005-09-25 22:32:49....	2005-09-25 22:26:46.675232	Y	AB	0D09	0000	0000	select c.continent, c.country, c.region...
2005-09-26 00:25:45....	2005-09-25 22:26:46.926264	Y	AB	0D09	0000	0000	SELECT AL1.CUSTOMER, AL5.MON...
2005-09-26 00:25:48....	2005-09-26 00:25:47.970400	Y	A7	0B06	0000	0000	SELECT AL1.CUSTOMER, AL5.MON...
2005-10-01 16:16:25....	2005-10-01 16:16:25.140992	Y	B3	0B02	0000	0000	-- VLDB07select c.continent, c.countr...
2005-10-01 16:16:39....	2005-10-01 16:16:39.039512	Y	A7	0B06	0000	0000	-- VLDB08select c.continent, c.countr...

Figure 4-29 Common columns in the 3006 record

4.6.9 The 3007 Record: Index evaluation

The 3007 record shows all indexes that are evaluated for a given table, including which (if any) were selected for use in this query and which were not and why. Reason codes are listed next to each index. A reason code of 0 indicates that the index was selected.

This record also indicates whether the optimizer timed out while evaluating the indexes. Indexes are evaluated in order from newest to oldest, in the same order as shown by the DSPDBR CL command, excluding the views. To ensure that an index is evaluated, you can delete and recreate it. This way it becomes first in the list.

The following columns are the most commonly used:

- **QVQTBL**: Table name
- **QVQLIB**: Table library name
- **QQC11**: Optimizer timed out (Y or N)
- **QQ1000**: Contains library qualified index names, each with a reason code
  - Reason code 0 indicates that an index was selected.
  - Other codes are displayed in the second level text of CPI432C and CPI432D messages.
  - Documentation and iSeries Navigator reports classify this as the “Optimizer timed out”; the 3007 row will be generated even when the optimizer does not time out.

Example 4-12 shows a query with some of the most commonly used columns in the 3007 record.

Example 4-12 Common columns in the 3007 record

```
WITH xx AS
(SELECT * FROM qgpl.qzg0000478 WHERE qqrid = 3007),

yy AS
SELECT qq1000 AS qqsttx, qqjfld, qqucnt
FROM qgpl.qzg0000478
WHERE qqrid = 1000 AND qqc21 <> 'MT'
AND (qvc1c = 'Y' OR (qqc21 IN('DL', 'UP')
AND qqc181 <= ' ') OR qqc21 IN ('IN', 'IC', 'SK', 'SI') OR qqc21 LIKE 'O%'))

SELECT qq1000,
varchar(qqsttx,20000) AS "Statement Text"
FROM xx a LEFT JOIN yy b
ON a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt
```

Figure 4-30 shows the result of the query in Example 4-12. This is an example with information from the qq1000 column in the 3007 record. The information is joined with information from the 1000 record so we know the index evaluation for a given query. The first row in the example shows that four indexes were evaluated and the last one (CUST_00003) was used to implement the query.

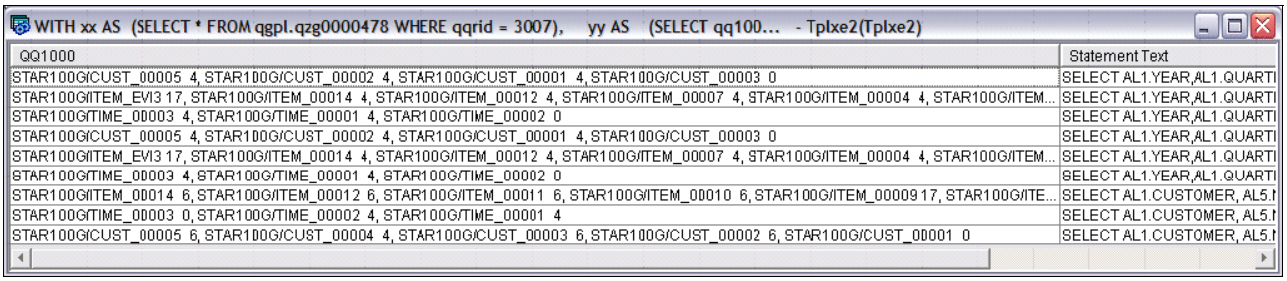


Figure 4-30 Common columns from the 3007 record

4.6.10 The 3010 Record: Host variables

The 3010 record shows substitution values for host variables or parameter markers in the query text (refer to the QQ1000 column in the 1000 record). This record appears just prior to each instance of an open, update, delete, or insert with subselect. This record is not displayed for insert with values. Data might not match exactly for updates with parameter markers in the SET clause.

- **QQ1000**: This column contains substitution values for host variables or parameter marker. The values (separated by commas) correspond left to right, with host variables and parameter markers. All values are displayed as a character, with no special indication of type. A floating point value is displayed as *F.
- **QQUCNT** and **QQI5**: These columns must be used to determine to which exact query the substitution values belong. The 3010 row type is not generated for the INSERT with Values statement.

4.6.11 The 3014 Record: General query optimization information

The 3014 record is displayed with full open optimization records. In most cases, one 3014 record is displayed per full open. You might see multiple 3014 records if the query consists of multiple separately run queries, for example, a subquery with grouping functions or views that need results materialized for use in the outer query. Values in this column help to identify the type of query that this record represents and the amount of time it took to open the cursor for this query.

This record also has summary information for the query or subquery. In most cases, there is one 3014 row per full open. Subselects and materialized views can cause multiple 3014 rows. It contains values for most of the settings that impact the query optimizer.

- **QQC102**: This column contains a library for the QAQQINI file if one is used. A value of *N indicates that no QAQQINI file was used.

  Many of the QAQQINI settings are found in the 3014 row; a couple are found in the 1000 row type.
- **QVP154**: Memory pool size
- **QVP155**: Memory pool ID
- **QQC16**: This column contains a Y when SQE is used to process the SQL statement (CQE = N).

4.6.12 The 3015 Record: SQE statistics advised

The 3015 record is generated by SQE when it determines that a column statistic needs to be collected or refreshed.

The following columns are commonly used:

- **QVQTBL**: Table name
- **QVQLIB**: Table library name
- **QQC11**: Statistics request type

  N No statistic existed for the column
  S Column statistic was stale
- **QQ1000**: Name of the column identified in Statistic Advice

Remember: Column statistics are created in the background automatically by default for all Statistics Advised. Use QQUCNT to analyze the SQL request to determine if an index is better than a column statistic.

4.6.13 The 3019 Record: Rows retrieved detail

The 3019 record shows a summary of the Fetch or Retrieve operations performed by DB2. For non-SQL interfaces, such as OPNQRYF, it is the only way to determine number of rows

returned and the amount of time to retrieve those rows. This record can also be used to analyze SQL requests.

The following columns are the most commonly used:

- **QQI1**: CPU time to return all rows, in milliseconds
- **QQI2**: Clock time to return all rows, in milliseconds
- **QQI3**: Number of synchronous database reads
- **QQI4**: Number of synchronous database writes
- **QQI5**: Number of asynchronous database reads
- **QQI6**: Number of asynchronous database writes
- **QQI7**: Number of rows returned
- **QQI8**: Number of calls to retrieve rows returned

4.6.14 Record information for SQL statements involving joins

SQL statements with a JOIN are among the most sensitive areas in tuning SQL queries. Unfortunately, there is not a specific record type for JOIN information. The information is contained in the access method records used for the tables that are used in the query. To see the join information, you need to look for the following, commonly used columns in records 3000 to 3004, 3007, 3021 to 3023, 3027 to 3029, among others:

- **QQJNP**: Join position
- **QQC21**: Join method, where NL indicates a nested loop, MF indicates a nested loop with selection, and HJ indicates a hash join

  Currently Database Monitor can only capture data related to a nested loop and nested loop with selection.
- **QQC22**: Join type, where IN indicates an inner join, PO indicates a left partial outer join, and EX indicates an exception join; also present in the 3014 record
- **QQC23**: Join operator

  When it uses the nested loop with selection, it shows a Cartesian product, even when it is not a Cartesian product.
- **QVJFANO**: Join fan out (Normal, Distinct fanout, or Unique fanout)

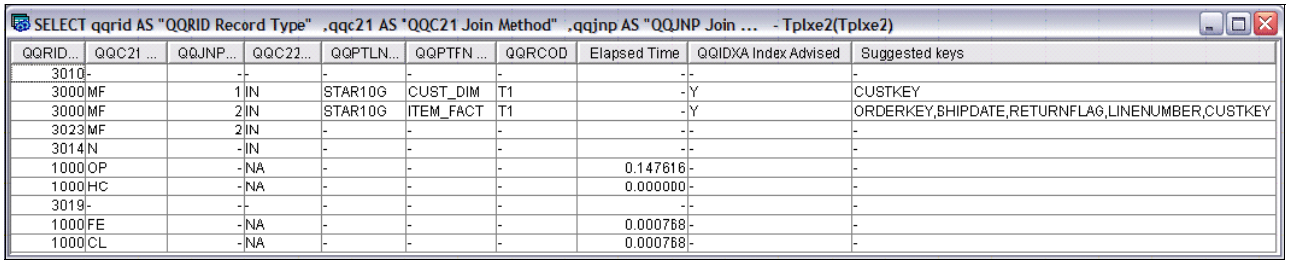
Example 4-13 shows a query to find information regarding the join operations.

Example 4-13 Information about join operations

```
SELECT qqrid AS "QQRID Record Type" ,qqc21 AS "QQC21 Join Method" ,qqjnp AS "QQJNP Join
Position" ,qqc22 AS "QQC22 Join Type" ,qqptln AS "QQPTLN Base Library" ,qqptfn AS "QQPTFN
Base Table" ,qqrcod AS "Reason"
,qqetim - qqstim AS "Elapsed Time" ,qqidxa AS "QQIDXA Index Advised" ,qqidxd AS "Suggested
keys"
FROM QZG00000479 a WHERE qqucnt = 28 AND qqjfld LIKE '%360528%' ORDER BY RRN(a)
```

Figure 4-31 shows the result of the query in Example 4-13. From the example, we can see:

- The operation joins two tables, CUST_DIM as the primary table and ITEM_FACT as the secondary table.
- The type of join is an inner join (IN in the QQC22 column).
- The implementation for both tables was a table scan (QQRID is 3000).
- The reason for the table scan is because no indexes exist (T1 in the QQRCOD column).
- An index is advised on both tables (Y in the QQIDXA column).
- QQIDXD lists the suggested keys for the advised indexes.



SELECT qqrid AS "QQRID Record Type" ,qqc21 AS "QQC21 Join Method" ,qqjnp AS "QQJNP Join ... - Tplxe2(Tplxe2)

QQRID...	QQC21 ...	QQJNP...	QQC22...	QQPTLN...	QQPTFN ...	QQRCOD	Elapsed Time	QQIDXA Index Advised	Suggested keys
3010	-	-	-	-	-	-	-	-	-
3000	MF	1	IN	STAR10G	CUST_DIM	T1	-	Y	CUSTKEY
3000	MF	2	IN	STAR10G	ITEM_FACT	T1	-	Y	ORDERKEY,SHIPDATE,RETURNFLAG,LINENUMBER,CUSTKEY
3023	MF	2	IN	-	-	-	-	-	-
3014	N	-	IN	-	-	-	-	-	-
1000	OP	-	NA	-	-	-	0.147616	-	-
1000	HC	-	NA	-	-	-	0.000000	-	-
3019	-	-	-	-	-	-	-	-	-
1000	FE	-	NA	-	-	-	0.000768	-	-
1000	CL	-	NA	-	-	-	0.000768	-	-

Figure 4-31 Information about Join operations

4.6.15 New MQT record types

Enhancements have been added to the Database Monitor data to indicate usage of materialized query tables (MQT).

Note: The query optimizer support for recognizing and using MQTs is available with V5R3 i5/OS PTF SI17164 and DB group PTF SF99503 level 4.

Although an MQT can contain almost any query, the optimizer only supports a limited set of query functions when matching MQTs to user-specified queries. The user-specified query and the MQT must both use the SQE. The optimizer only uses one MQT per query.

- **1000/3006 Record**: QQC22 has a new reason code of B5, which indicates that the access plan needed to be rebuilt because the MQT was no longer eligible to be used. The reasons might be that:
  - The MQT no longer exists
  - A new MQT was found
  - The enable or disable query optimization changed
  - Time since the last REFRESH TABLE exceeds the MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option
  - Other QAQQINI options no longer match
- **3014 Record**: This record shows the new QAQQINI file values:
  - Logical field name QQMQTR, physical field name QQI7, contains the MATERIALIZED_QUERY_TABLE_REFRESH_AGE duration. If the QAQQINI parameter value is set to *ANY, the timestamp duration will be 99999999999999.
  - Logical field name QQMQTU, physical field name QVC42. The first byte of QVC24 contains the MATERIALIZED_QUERY_TABLE_USAGE. Supported values are:

    N *NONE, no materialized query tables are used in query optimization and implementation.

    A User-maintained refresh-deferred query tables might be used.

    U Only user-maintained materialized query tables might be used.
- **3000, 3001, 3002 Records**: New columns have been added to the 3000, 3001, and 3002 records to indicate that a table or tables were replaced with an MQT. The logical field name QQMQT, physical field name QQC13, is either Y or N, indicating that this is an MQT, which replaced a table or tables in the query.
- **3030 Record**: The new 3030 record contains information about the MQTs that are examined. This record is only written if MQTs are enabled and MQTs exist over the tables specified in the query.

5

Analyzing database performance data using iSeries Navigator

In Chapter 4, “Gathering database SQL performance data” on page 51, you learned about the different Database Monitors and how to collect performance data. After you collect the performance data, you must analyze it.

You can analyze the Database Monitor data by using predefined queries that come with iSeries Navigator. In this chapter, we explain the analysis tool that is included with iSeries Navigator and the predefined queries.

5.1 Considerations before analyzing Database Monitor data

You can collect Database Monitor data in several ways as explained in Chapter 4, “Gathering database SQL performance data” on page 51. In some cases, you must import the data into a graphical interface to take advantage of the reporting capabilities of the tool. This section addresses some considerations for analyzing the Database Monitor data.

Note: Keep in mind that performance data can become quite large in size and could consume a lot of disk space. The more data that you have, the more time it takes to analyze it.

5.1.1 Importing the Database Monitor data

One way to collect Database Monitor data is to use the Start Database Monitor (STRDBMON) CL command on a green screen. But to use the reporting capability of iSeries Navigator, you must import the Database Monitor data. For ease of use, the structure and layout of the performance data are identical to the data created by the SQL Performance Monitor tool. This enables you to use the predefined reports in iSeries Navigator to analyze performance data gathered by Database Monitor.

To analyze the data using the predefined reports, import the performance data collection from the Database Monitor table into the SQL Performance Monitor. Select **Database** → **SQL Performance Monitors** → **Import** as shown in Figure 5-1.

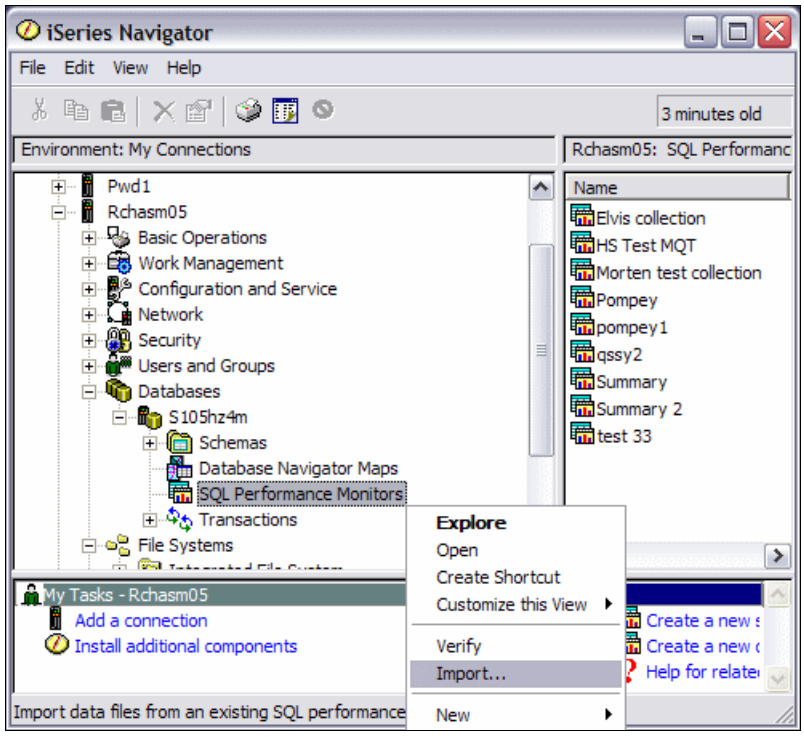


Figure 5-1 Selecting the options to import performance data from the Database Monitor file

In the Import SQL Performance Monitor Files window (Figure 5-2), you import your Database Monitor performance data. Type the monitor name that you want to specify, select the schema where you want the monitor to reside, and choose the type of monitor. Then click **OK**.

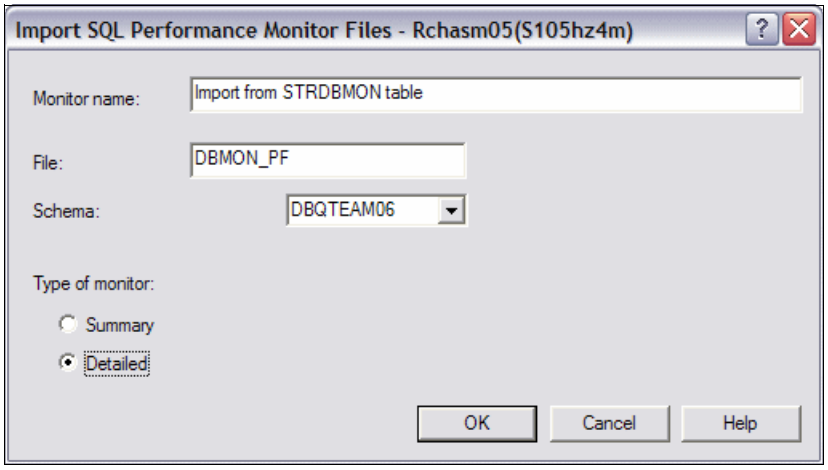


Figure 5-2 Import SQL Performance Monitor Files window

As shown in the example in Figure 5-3, the newly created SQL Performance Monitor now reflects a status of *Imported*.

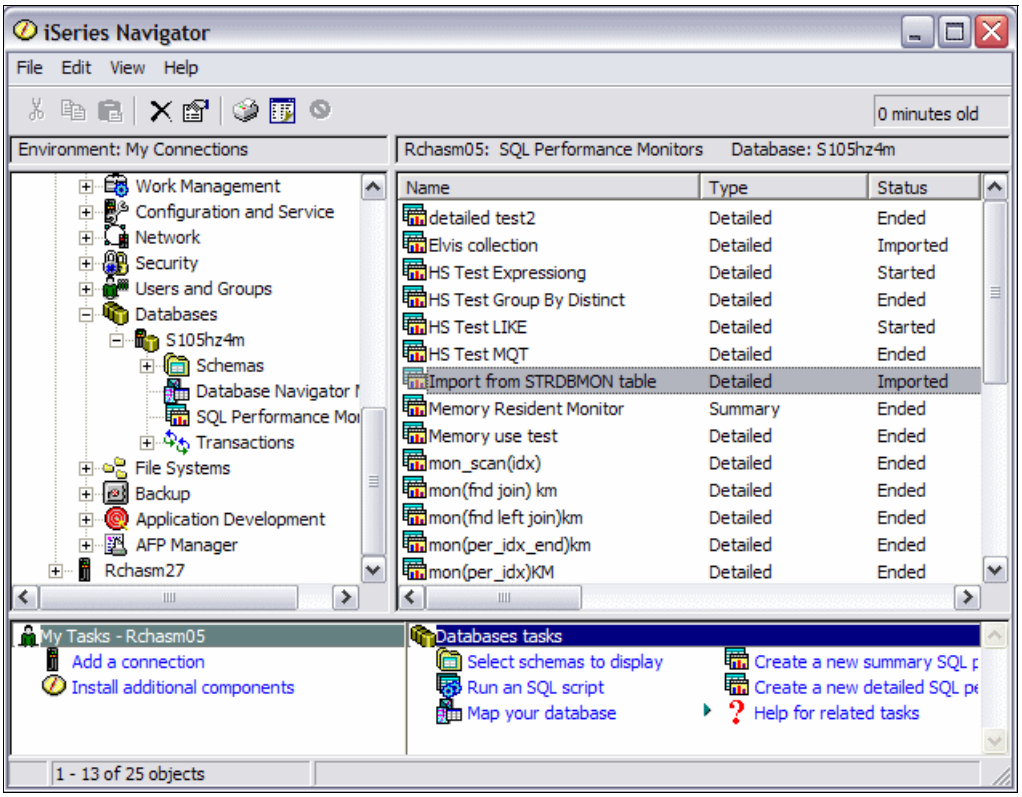


Figure 5-3 Imported SQL Performance Monitors

Note: It is possible to run the Database Monitor on one iSeries server and import the collection data into a separate system to perform the analysis.

5.1.2 Reducing the analysis time

If you are collecting Performance Monitor information for all jobs or all queries, the Database Monitors can generate a rather large file in the case of using a detailed SQL Performance Monitor. Since the large monitor database files can cause slow response time during your analysis, for faster analysis, you can select only those jobs in which you are interested by using one of the following options:

- CRTDUPOBJ command and SQL INSERT with a subselect specifying the QQJNUM value or values
- CREATE TABLE small as (SELECT...WHERE QQJNUM=*xyz* with DATA, where *xyz* is the job number in which you are interested

Because the monitor files created from the Detailed SQL Performance Monitor have generic names with sequence numbers as suffixes, you can find the monitor file name on the Saved Data page of the Properties panel (Figure 5-4).

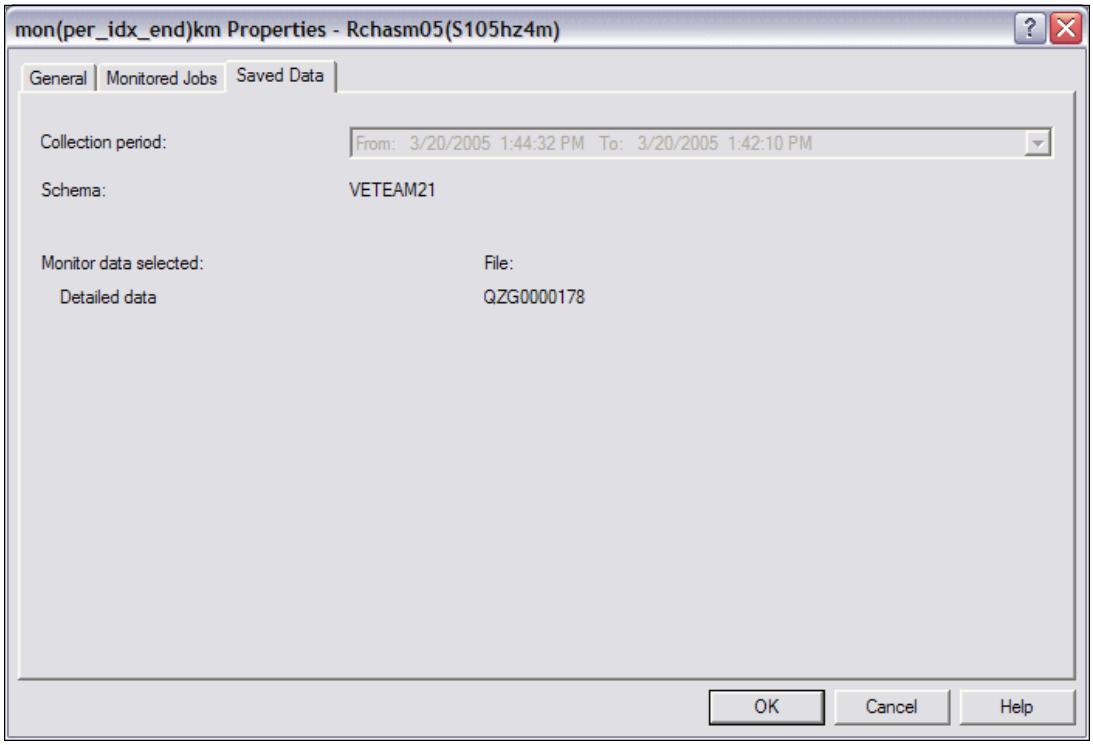


Figure 5-4 Saved Data properties of the Detailed SQL Performance Monitor

Note: After you copy the rows into a separate file, re-import the file again into the SQL Performance Monitor.

You can also create indexes on the monitor file over common selection and grouping or order by columns. For more information about index creation over commonly used keys, refer to 6.2.4, “Creating additional indexes over the Database Monitor table” on page 136.

5.2 Predefined database performance reports

Information gathered by the database monitoring tools provides the most complete pictures about a query's execution. You can analyze this information by using predefined result reports in SQL Performance Monitor.

In general, there are two categories of performance data:

- **Summary SQL Performance Monitoring**: Summary SQL performance data is collected via a Memory Resident Database Monitoring capability that can be managed with application program interfaces (APIs) or with iSeries Navigator. When the monitor is paused or ended, the data is written to disk so it can be analyzed. Because the information is stored only in memory while it is collected, the performance impact on the system is minimal, but the trade-off is that only summary data is gathered.
- **Detailed SQL Performance Monitoring**: Detailed SQL performance data is collected in real time and recorded immediately to a database table. The monitor is not required to be paused or ended to begin analysis of the results. Since this monitor saves the data in real time, it might have a greater impact on overall system performance, depending on the number of jobs that are monitored.

Note: The Memory Resident Monitor statistics are kept in main storage for fast recording, but must be written to database files to use the iSeries Navigator interface to review the results. To write the statistics to database files, either pause or end the started SQL Performance Monitor.

5.2.1 Accessing the collected performance data

To review the SQL Performance Monitor results, in the right pane, right-click the active SQL Performance Monitor files. In the selection window that is displayed, you see a variety of monitor actions, such as Pause, Continue, and End, as shown in Figure 5-5.

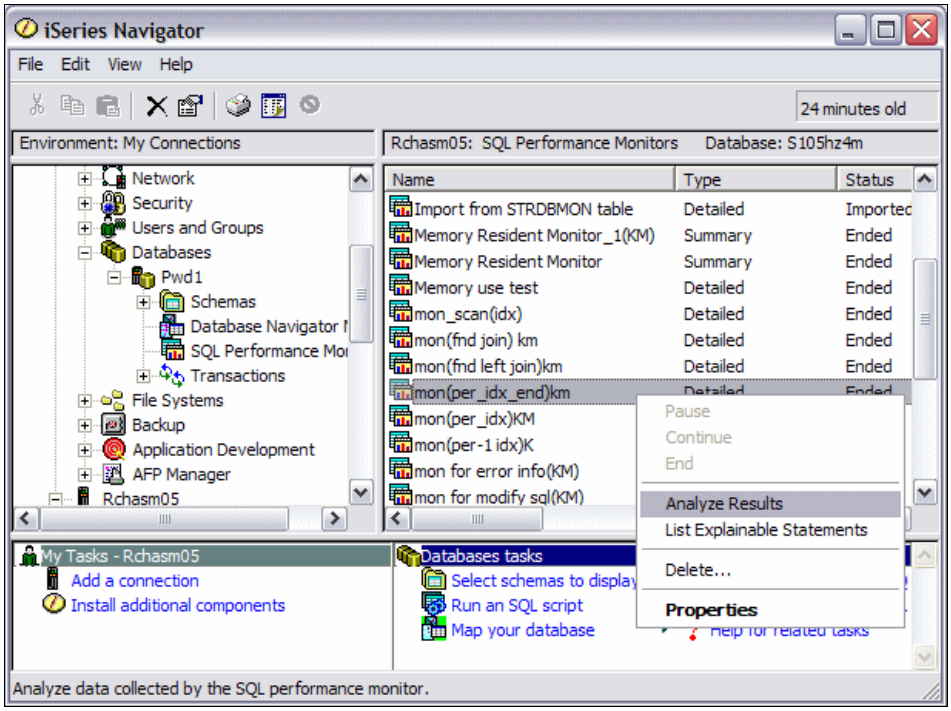


Figure 5-5 Managing the SQL Performance Monitor

From this list, you can choose from the following functions:

- **Pause**: This function stops the current collection of statistics and writes the current statistics into several database files or tables that can be queried by selecting the Analyze Results action. The monitor remains ready to collect more statistics, but requires the Continue action to restart collection.
- **Continue**: This function restarts the collection of statistics for a monitor that is currently paused.

> **Note:** The Pause and Continue functions are activated only after a Summary type SQL Performance Monitor is started because Detailed SQL Performance Monitor does not need to be paused to view results.

- **End**: This function stops and ends the monitor. If you started a Summary type SQL Performance Monitor, then it writes the current collection of statistics to the database files or tables.
- **Analyze Results**: This function opens a window with three tabs for selecting ways to view the collected statistics in the database files or tables:
  - Memory Resident SQL Performance Monitor
    - Summary Results
    - Detailed Results
    - Composite View
  - Detailed SQL Performance Monitor
    - Summary Results
    - Detailed Results
    - Extended Detailed Results
- **List Explainable Statements**: This function opens a window that lists the SQL statements for which the Detailed SQL Performance Monitor has collected data and for which a Visual Explain diagram can be produced. See 5.4.1, “List Explainable Statements” on page 130, for an example.
- **Properties**: This function opens a window with three tabs that represent the original monitor definition:
  - General
  - Monitored Jobs
  - Saved Data

iSeries Navigator provides many predefined queries to view the recorded statistics. You can select these queries by checking the various query types on the Analyze Results panels.

Memory Resident SQL Performance Monitoring

To begin viewing the results for the Memory-based Database Monitor, right-click the paused or ended monitor and then select **Analyze Results**.

Figure 5-6 shows the first results panel that groups queries according to three tabs:

- Summary Results
- Detailed Results
- Composite View

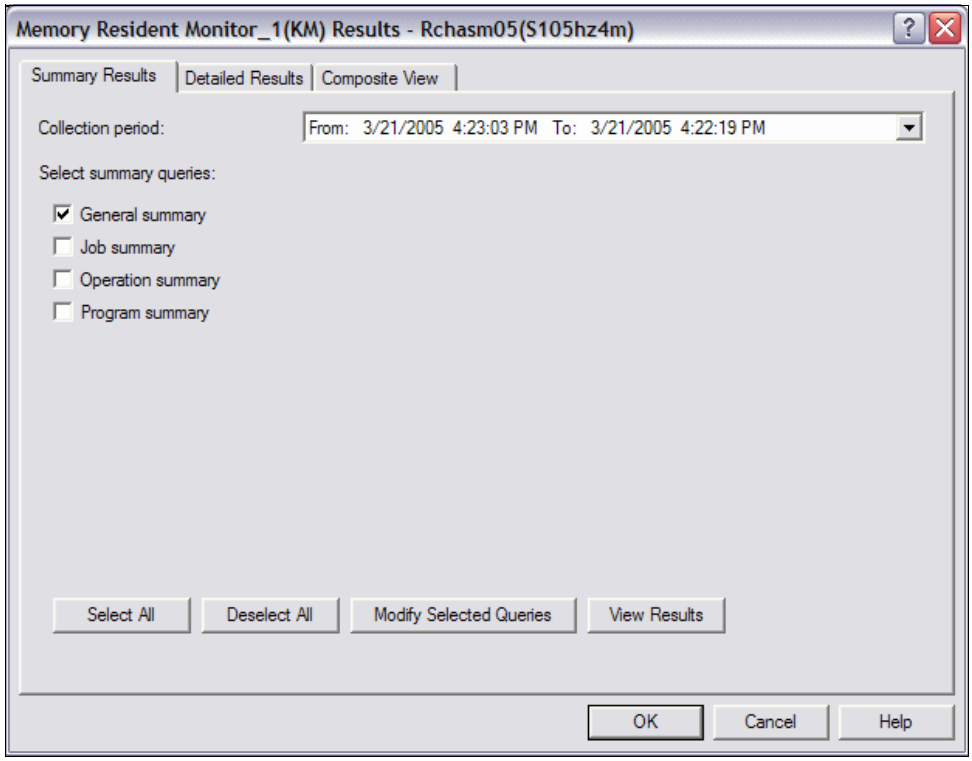


Figure 5-6 Access SQL Performance Monitor results

On the Summary Results page, you can select individual queries or use the Select All button. After you select the queries that you want to run, click the **View Results** button.

You can also choose to modify the predefined queries and run the new queries by clicking the Modify Selected Queries button. For more information about this function, refer to 5.3, “Modifying a predefined query report” on page 119.

The Memory Resident Database Monitor uses its own set of tables instead of using the single table with logical files that the Detailed SQL Performance Monitor uses. The Memory-Resident tables closely match the suggested logical files of the Detailed SQL Performance Monitor.

You can see the set of tables that correspond to the data to be collected by the Database Monitor in the Saved Data property panel of Database Monitor as shown in Figure 5-7.

The monitor files have generic names with sequence numbers as suffixes. You can also see the corresponding External Tables in Figure 5-7. For more information about the External Table description for the Memory Resident Database Monitor, refer to Chapter 2, “Database Monitor DDS” in *DB2 Universal Database for iSeries Database Performance and Query Optimization*, which is available in the iSeries Information Center on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzajq/rzajqmst.htm

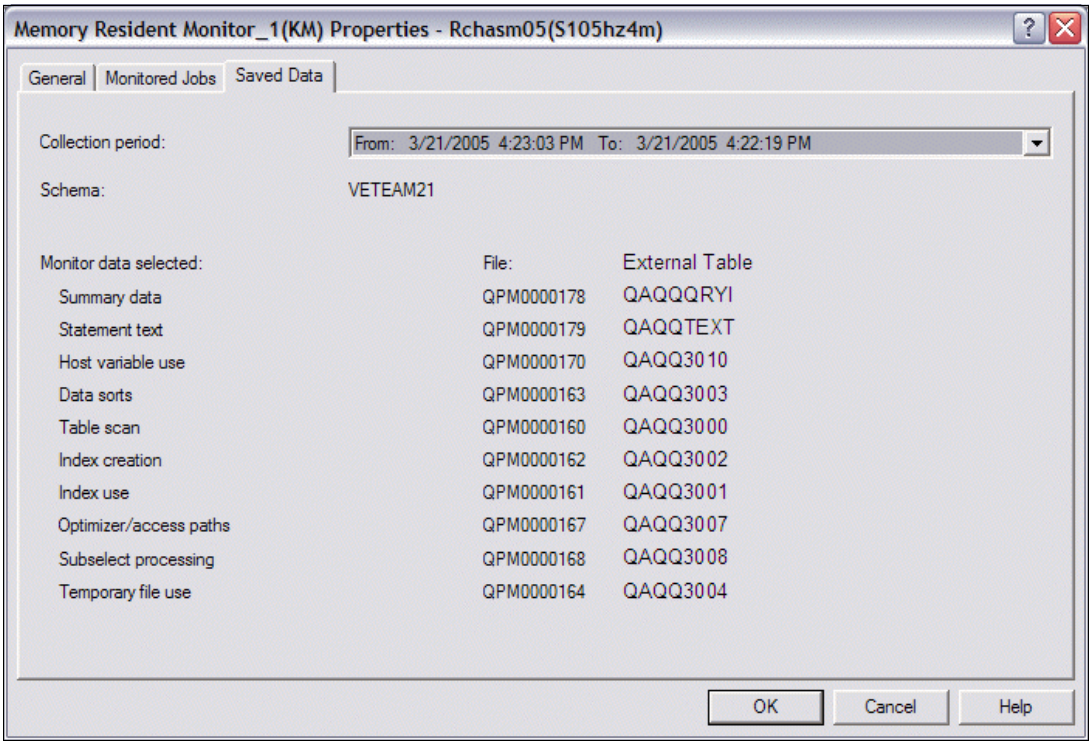


Figure 5-7 Saved Data Property of Memory Resident Performance Monitor Results

Detailed SQL Performance Monitor

To access the collected Detailed Performance Monitor data, select **File** → **Analyze Results** as shown in Figure 5-5. Notice that the Detailed SQL Performance Monitor is not required to be paused or ended.

Figure 5-8 shows the first results window that groups queries according to three tabs:

- Summary Results
- Detailed Results
- Extended Detailed Results

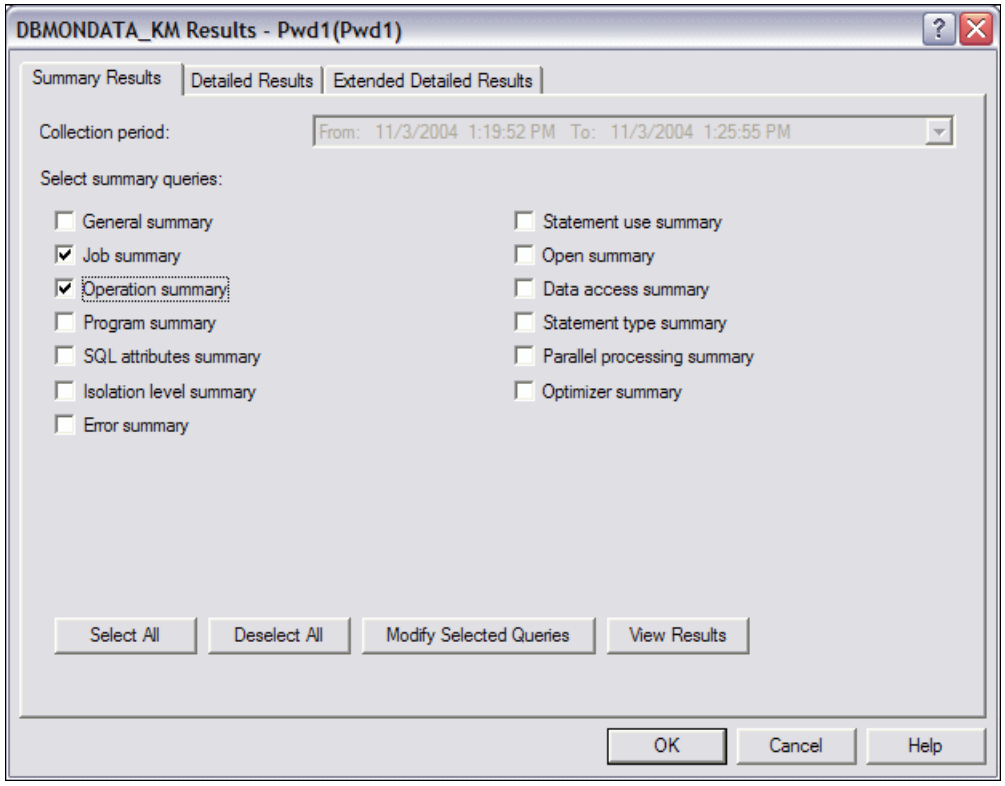


Figure 5-8 Detailed SQL Performance Monitor results

You can select from a variety of predefined reports to view performance-related information about your queries and choose reports that you want to analyze. Figure 5-9 shows an example of a General Summary predefined report with summary information. For example, it might indicate the number of times that tables are scanned and the number of times that the optimizer recommends to create a permanent index.

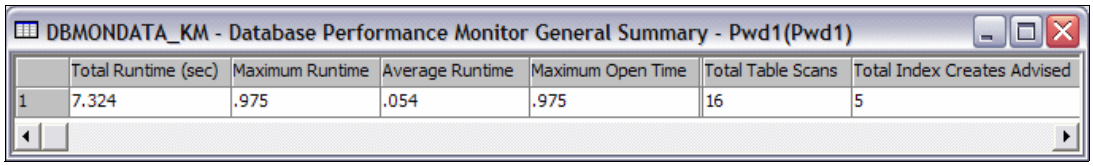


Figure 5-9 General Summary report example

By analyzing the contents of these reports, you can identify problem areas in your queries and take the appropriate steps to remedy such problems. The details of predefined reports are described in the following section.

5.2.2 SQL performance report information from Summary reports

To begin viewing the summary reports from Memory Resident Database Monitor and the Detailed SQL Performance Monitor results, click **View Results** in the Summary Results tab.

Because all reports from the summary results are summarized at a Database Monitor level, you can get a high-level perspective of what was happening during the monitoring such as:

- Analysis of the type of SQL operation
- Identification of the jobs that consume the most DB2 resources

Summary reports of the Memory Resident Monitor

The Memory Resident Database Monitor collects much of the same information as the Detailed SQL Performance 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.

Table 5-1 describes the summary reports that are available for the Memory Resident Monitor.

Table 5-1 Types of summary reports in the SQL Performance Monitor

Reports	Description
General Summary	This report contains information that summarizes all SQL activity, such as the amount of SQL used in the application, whether the SQL statements are mainly short-running or long running, and whether the number of results returned is small or large.
Job Summary	This report contains a row of information for each job. The information can be used to tell which jobs on the system are the heaviest users of SQL, and therefore, which ones are perhaps candidates for performance tuning.
Operation Summary	This report contains a row of summary information for each type of SQL operation. For example, if a large amount of INSERT activity is occurring, then using an OVRDBF command to increase the blocking factor or perhaps use of the QDBENCWT API is appropriate.
Program Summary	This report contains a row of information for each program that performed SQL operations. This information can be used to identify which programs use the most or most expensive SQL statements. Note that a program name is available only if the SQL statements are embedded inside a compiled program. SQL statements that are issued through Open Database Connectivity (ODBC), Java Database Connectivity (JDBC), or OLE DB have a blank program name unless they result from a procedure, function, or trigger.

You can find the most time consuming job from a Job Summary report (see Figure 5-10). This information is quite useful because a predefined Job Summary report can be a starting point for further analysis.

Memory Resident Monitor_1(KM) - Database Performance Monitor Job Summary - Rchasm05(S105hz4m)

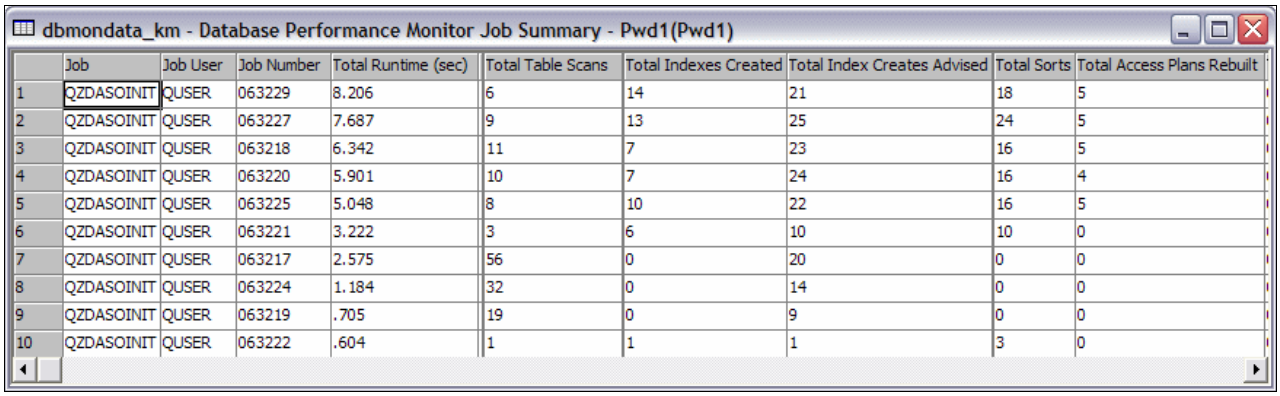
Job	Job User	Job Number	Total Runtime (sec)	Total Indexes Created	Total Index Creates Advised	Total Temporary Tables	Total Access Plans Rebuilt
QZDASOINIT	QUSER	004127	12.698	1	1	1	13
QZDASOINIT	QUSER	004485	7.757	0	0	0	0

Figure 5-10 Memory-Resident Job Summary report

Summary reports of the Detailed Performance Monitor

The Memory-Resident Monitor is not meant to replace the Detailed SQL Performance Monitor. There are circumstances where the loss of detail in the Memory-Resident Monitor is insufficient to fully analyze an SQL statement. In these cases, you should still use the Detailed SQL Performance Monitor.

As shown for the Memory-Resident Monitor Summary report, in the summary report for the Detailed Monitor, you can also identify the jobs that consume the most run time by viewing a job summary report like the example in Figure 5-11.

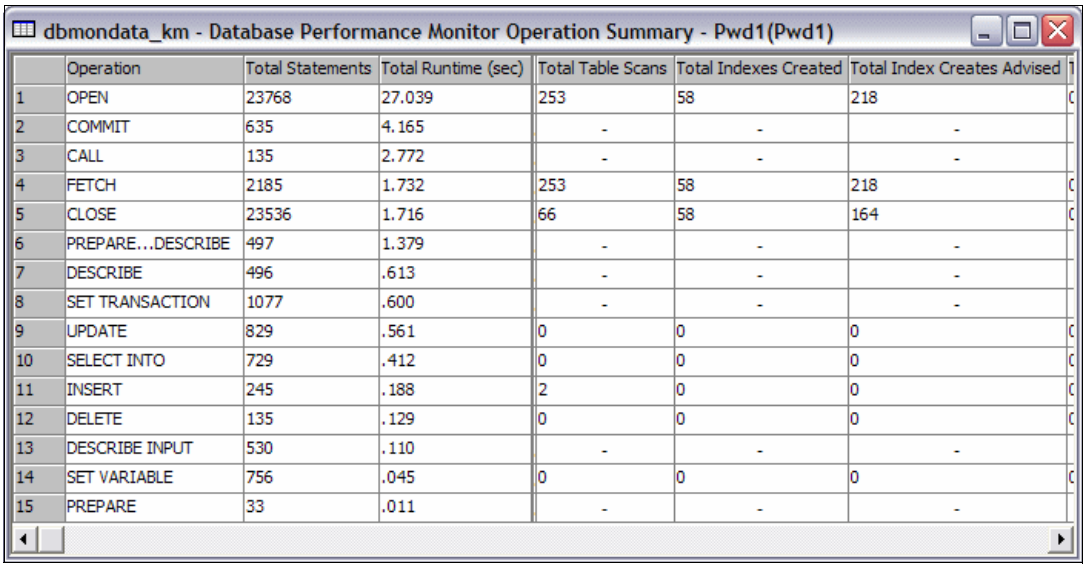


dbmondata_km - Database Performance Monitor Job Summary - Pwd1(Pwd1)

	Job	Job User	Job Number	Total Runtime (sec)	Total Table Scans	Total Indexes Created	Total Index Creates Advised	Total Sorts	Total Access Plans Rebuilt
1	QZDASOINIT	QUSER	063229	8.206	6	14	21	18	5
2	QZDASOINIT	QUSER	063227	7.687	9	13	25	24	5
3	QZDASOINIT	QUSER	063218	6.342	11	7	23	16	5
4	QZDASOINIT	QUSER	063220	5.901	10	7	24	16	4
5	QZDASOINIT	QUSER	063225	5.048	8	10	22	16	5
6	QZDASOINIT	QUSER	063221	3.222	3	6	10	10	0
7	QZDASOINIT	QUSER	063217	2.575	56	0	20	0	0
8	QZDASOINIT	QUSER	063224	1.184	32	0	14	0	0
9	QZDASOINIT	QUSER	063219	.705	19	0	9	0	0
10	QZDASOINIT	QUSER	063222	.604	1	1	1	3	0

Figure 5-11 Job Summary report in the Detailed Performance Monitor

From this report, you can see that the most time consuming job also has the highest value of temporary index creation. You can also analyze the type of SQL operation from the Operation Summary report as shown in Figure 5-12.



dbmondata_km - Database Performance Monitor Operation Summary - Pwd1(Pwd1)

	Operation	Total Statements	Total Runtime (sec)	Total Table Scans	Total Indexes Created	Total Index Creates Advised
1	OPEN	23768	27.039	253	58	218
2	COMMIT	635	4.165	-	-	-
3	CALL	135	2.772	-	-	-
4	FETCH	2185	1.732	253	58	218
5	CLOSE	23536	1.716	66	58	164
6	PREPARE...DESCRIBE	497	1.379	-	-	-
7	DESCRIBE	496	.613	-	-	-
8	SET TRANSACTION	1077	.600	-	-	-
9	UPDATE	829	.561	0	0	0
10	SELECT INTO	729	.412	0	0	0
11	INSERT	245	.188	2	0	0
12	DELETE	135	.129	0	0	0
13	DESCRIBE INPUT	530	.110	-	-	-
14	SET VARIABLE	756	.045	0	0	0
15	PREPARE	33	.011	-	-	-

Figure 5-12 Operation Summary result in the Detailed Performance Monitor

In Figure 5-12, you can see that the most time-consuming SQL operation is OPEN. The number of total table scans is 253, and the number of temporary index creation is 58. From this report, you can see that the creation of perfect indexes can be one of many good candidates for reducing OPEN time and the total run time.

Note: These reports are a good starting point, but they do not identify the cause of the performance problem.

Table 5-2 describes additional reports of Detailed Performance Monitor results.

Table 5-2 Summary reports for the Detailed Performance Monitor

Reports	Description
SQL Attributes Summary	This report contains a summary of the optimization attributes, which can help identify attributes that potentially are more costly than others. For example, in some cases ALWCPYDTA(*YES) can allow the optimizer to run a query faster if live data is not required by the application. Also, *ENDMOD and *ENDPGM are much more expensive than *ENDJOB or *ENDACTGRP.
Isolation Level Summary	This report contains a summary of the number of statements that were run under each isolation level. The higher the isolation level is, the higher the chance of contention is between users. For example, a high level of Repeatable Read or Read Stability use is likely to produce a high level of contention. You should always use the lowest level isolation level that still satisfies the application design requirement.
Error Summary	This report contains a summary of any SQL statement error messages or warnings that were captured by the monitor.
Statement Use Summary	This report contains a summary of the number of statements that are executed and the number of times they are executed during the collection period of the Performance Monitor. This information provides the user with a high-level indication of how often the same SQL statements are used multiple times. If the same SQL statement is used more than one time, it might be cached. Subsequent uses of the same statement are less expensive. It is more important to tune an SQL statement that is executed many times than an SQL statement that is only run one time.
Open Summary	This report contains a summary of the number of statements that perform an open and the number of times they are executed during the Performance Monitor collection period. The first open of a query in a job is a full open. After this, the open data path (ODP) might be pseudo-closed and then reused. An open of a pseudo-closed ODP is far less expensive than a full open. The user can control when an ODP is pseudo-closed and the number of pseudo-closed ODPs are allowed in a job by using the Change Query Attributes action in the Database Folder of iSeries Navigator. In rare cases, an ODP is not reusable. High usage of nonreusable ODPs might indicate that the SQL statements causing the nonreusable ODPs should be rewritten.
Data Access Summary	This report contains a summary of the number of SQL statements that are read-only versus those that modify data. This information provides the user with a less detailed view of the type of SQL statements used than that available through the Operation Summary. This information can then be used to try specific performance tuning techniques. For example, if a large amount of INSERT activity is occurring, it might be appropriate to use the Override with Database File (OVRDBF) command, to increase the blocking factor, or to use of the QDBENCWT API.
Statement Type Summary	This report contains a summary of whether SQL statements are in extended dynamic packages, in a system-wide statement cache, or in regular dynamic or static SQL statements. This information provides the user with a high level indication of the number of SQL statements that were fully parsed and optimized (dynamic). The information also indicates whether the SQL statements and access plans were stored statically in a program, procedure, function, package, or trigger. An SQL statement that must be fully parsed and optimized is more expensive than the same statement that is static, extended dynamic, or cached in the system-wide statement cache.
Parallel Processing Summary	This report contains a summary of the parallel processing techniques that were used. This information provides the user with a high-level indication of whether one of the many parallel processing techniques were used to execute the SQL statements. Most parallel processing techniques are available only if the Symmetric Processing for iSeries is installed. When the option is installed, the user must specify the degree of parallelism through the Change Query Attributes action in the Database Folder of iSeries Navigator, the Change Query Attributes (CHGQRYA) CL command, or the QQRYDEGREE system value.

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

For more information about SQL Summary Performance Monitor Summary results, refer to the *Preparing for and Tuning the V5R2 SQL Query Engine on DB2 Universal Database for iSeries*, SG24-6598.

5.2.3 SQL performance report information from Extended Detailed reports

The Extended Detailed results are most useful if the user has a basic knowledge of IBM System i5™ or iSeries query optimization techniques. The most useful function from the Extended Retailed reports is that you can analyze each SQL request.

Since each SQL statement can be identified by the monitor by according to job number, statement number, and unique count, these reports give a DB2 trace at each SQL statement level. But, because statements are not sorted or grouped by job, you must determine how to address this issue as explained in the following sections.

Basic Statement Information report

From the Job Summary report shown in Figure 5-11 on page 103, where you can identify the most time consuming job, we copied all data for a single job (job number 063229) into its own collection to reduce analysis time. For more information about faster analysis, refer to 5.1.2, “Reducing the analysis time” on page 96.

The most useful information is which query is the most time consuming. You can see that by looking at the Basic Statement Information report (Figure 5-13).

DBmon for specific job - Database Performance Monitor Basic Statement Information - Pwd1(Pwd1)

	Total Runtime	Maximum Runtime	Statement Usage Count	Statement Text
1	.974844	.214408	18	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE, T1.
2	.338544	.191256	6	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE, T1.
3	.693808	.182808	16	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE, T1.
4	.313208	.161968	8	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE, T1.
5	.145176	.138624	4	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE, T1.

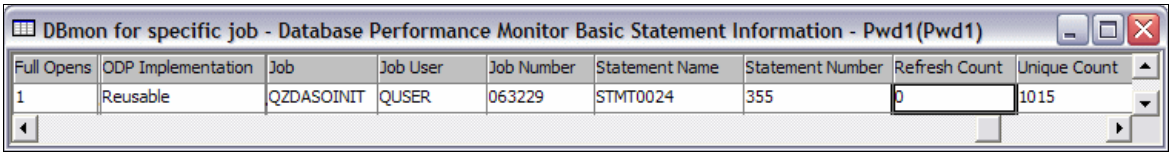
Figure 5-13 The Basic Statement Information report

The Basic Statement Information report provides the user with basic information about each SQL statement. The most expensive SQL statement is presented first in the list; at a glance, the user can see which statements (if any) were long running.

Literals in the original SQL text might be replaced by parameter markers in prepared text if SQL was able to convert them during preparation (desired). For the original SQL text, you can

use literal values from the matching Host Variable Values column in the place of parameter markers.

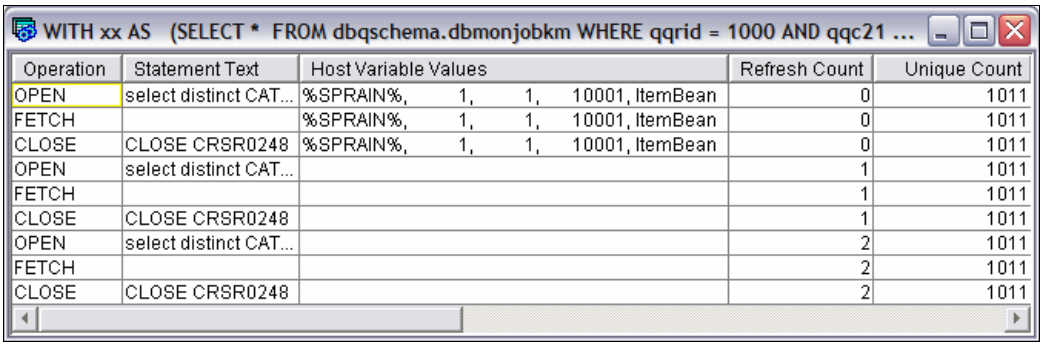
The other important information from Basic Statement Information is *Unique Count*. In case each query statement in the result report is sorted by only time stamp, you can identify a specific query by seeking a Unique Count. The example in Figure 5-14 shows the Unique Count of the most time consuming SQL statement.


DBmon for specific job - Database Performance Monitor Basic Statement Information - Pwd1(Pwd1)

Full Opens	ODP Implementation	Job	Job User	Job Number	Statement Name	Statement Number	Refresh Count	Unique Count
1	Reusable	QZDASOINIT	QUSER	063229	STMT0024	355	0	1015

Figure 5-14 The information of specific SQL statements

Since the Unique Count number does not increase any more if ODPs are reusable, then you must also consider looking at the Refresh Count number to identify unique SQL statements like the example in Figure 5-15.


WITH xx AS (SELECT * FROM dbqschema.dbmonjobkm WHERE qqrid = 1000 AND qqc21 ...

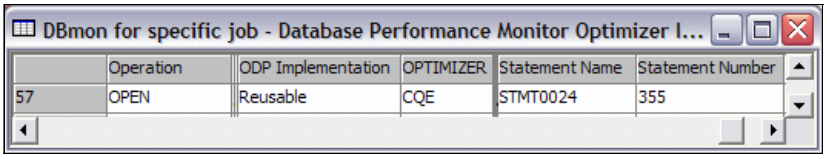
Operation	Statement Text	Host Variable Values	Refresh Count	Unique Count
OPEN	select distinct CAT...	%SPRAIN%, 1, 1, 10001, ItemBean	0	1011
FETCH		%SPRAIN%, 1, 1, 10001, ItemBean	0	1011
CLOSE	CLOSE CRSR0248	%SPRAIN%, 1, 1, 10001, ItemBean	0	1011
OPEN	select distinct CAT...		1	1011
FETCH			1	1011
CLOSE	CLOSE CRSR0248		1	1011
OPEN	select distinct CAT...		2	1011
FETCH			2	1011
CLOSE	CLOSE CRSR0248		2	1011

Figure 5-15 Refresh Count with reusable ODPs

In our example, since ODP is reusable during repeated execution of same statement, Refresh Count increases from 0 to 2, while Unique Count is fixed to 1011.

Optimizer Information report

From the previous information, you can see the Optimizer Information report (Figure 5-16). This report provides you with basic optimizer information about SQL statements that involve data manipulation (select, open, update and so on).


DBmon for specific job - Database Performance Monitor Optimizer I...

	Operation	ODP Implementation	OPTIMIZER	Statement Name	Statement Number
57	OPEN	Reusable	CQE	STMT0024	355

Figure 5-16 The Optimizer Information report

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

In our example, you can see that the optimizer chose to use Classic Query Engine (CQE), and ODP is reusable for this specific SQL statement. To retrieve a report of the specific SQL

statement more easily, you can also make the same report by clicking Modify Selected Query and modifying the predefined query statement. In our example, a new selection clause is inserted in the predefined query to retrieve a result of statement number 355 and Unique Count 1015 as shown in Figure 5-17.

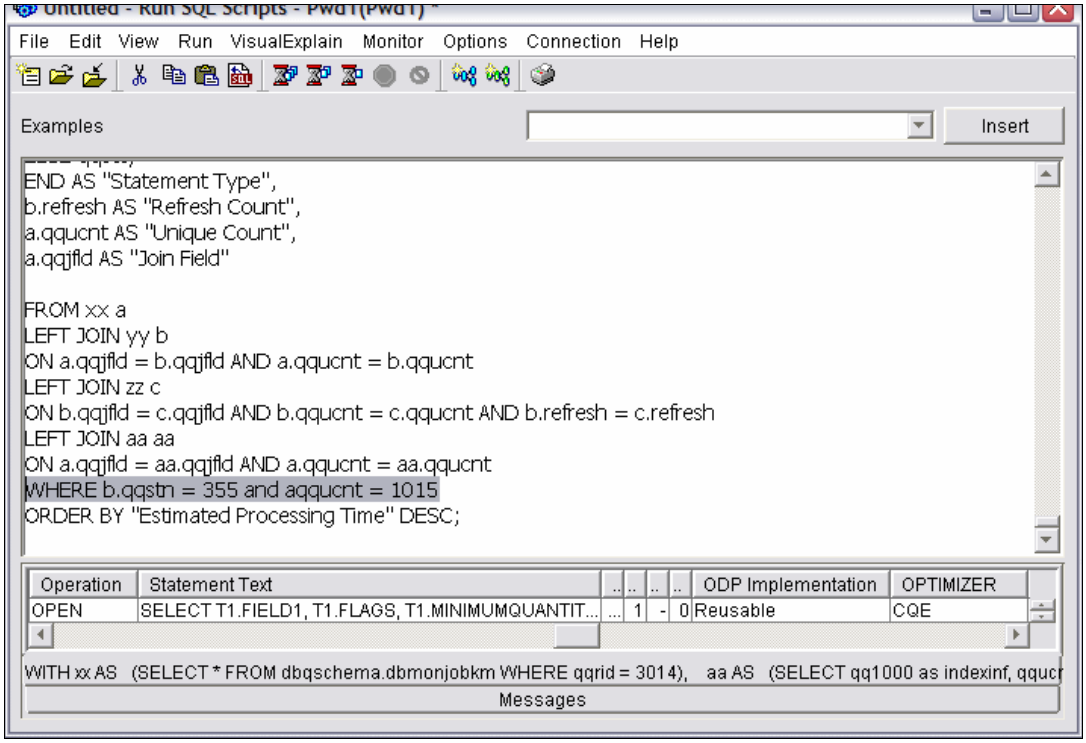


Figure 5-17 The report from Run SQL Script window

The same result is retrieved and displayed in the iSeries Navigator Run SQL Script window, from where you can modify and execute it. For more information about modifying a predefined query, see 5.3, “Modifying a predefined query report” on page 119.

Table Scan Information report

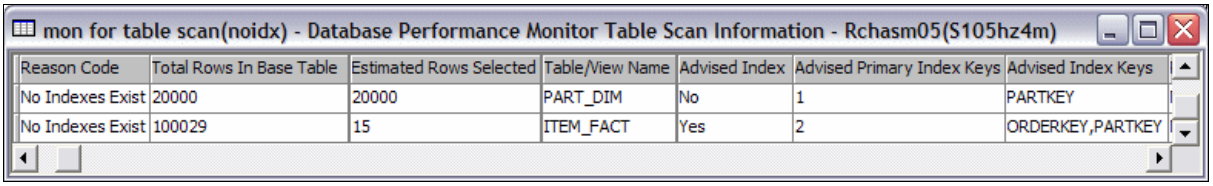
The Table Scan Information report indicates which entire table is scanned in arrival sequence order without using an index. A table scan is generally acceptable in cases where a large portion of the table is selected or the selected table contains a small number of records. Otherwise, a table scan of large tables can be time consuming. Therefore, if you investigate a long running SQL statement, using an index usually provides better performance.

To analyze and show table scan information, we run the sample query joining two tables as shown in Example 5-1.

Example 5-1 Sample query joining two tables

```
select  a.orderkey, a. partkey, b.part, a.quantity
 from   veteam21.item_fact a, veteam21.part_dim b
 where  a.partkey = b.partkey
  and   a.orderkey = '896';
```

We started SQL Performance Monitor in the Run SQL Script window using the JDBC interface. The Table Scan Information report shows that the optimizer uses table scan for both tables as shown in Figure 5-18.


mon for table scan(noidx) - Database Performance Monitor Table Scan Information - Rchasm05(S105hz4m)

Reason Code	Total Rows In Base Table	Estimated Rows Selected	Table/View Name	Advised Index	Advised Primary Index Keys	Advised Index Keys
No Indexes Exist	20000	20000	PART_DIM	No	1	PARTKEY
No Indexes Exist	100029	15	ITEM_FACT	Yes	2	ORDERKEY,PARTKEY

Figure 5-18 Table Scan Information report

In the Table Scan Information report, the following information is the most commonly used:

- **Reason Code**: This column shows why the arrival sequence was chosen by the optimizer. Our example in Figure 5-18 indicates `No Index Exists` in both of tables.
- **Total rows in Base Table**: This column indicates the number of rows in a table.
- **Estimated Rows Selected**: From this value, you can determine if the table scan was done for a significant number of rows.
- **Advised Index**: This column indicates whether an index was advised (Yes or No).
- **Advised Index Keys**: This column suggests keys for an index. In our example, since both tables do not have any indexes, the index creation can be one of good candidates for providing better performance. Especially, since the table scan of ITEM_FACT table selects only small portion of all rows, the index creation on ITEM_FACT is required more.

  Note: The Advised Index column is set to “Y” only if the local selection key columns exist on the table.

  The Advised Index Keys column can contain both primary and secondary keys. The Advised Primary Index keys column indicates the number of keys that are considered primary. The other keys are considered less selective (secondary).

  In case of ITEM_FACT table, two index key columns are advised. The first key column (ORDERKEY) is for local selection, and the other (PARTKEY) is for a nested loop join with the PART_DIM table. For the PART_DIM table, you can see that the Advised Index Key column (PARTKEY) is for a nested loop join, although the Advised Index column is set to “No”.

You can see the same results in Visual Explain diagram like the example shown in Figure 5-19. This diagram is displayed by executing List Explainable Statements. For more information about List Explainable Statements, refer to 5.4.1, “List Explainable Statements” on page 130.

An insert with a subselect also has table scan information for the file inserted into, but this is not a performance problem on its own. Nor does it indicate that the ODP is nonreusable. The record data might contain useful Index Advisor data.

From this report, you can see that the perfect index creation is a key element for providing a better access method with optimizer. Therefore, we investigate more information related to the index.

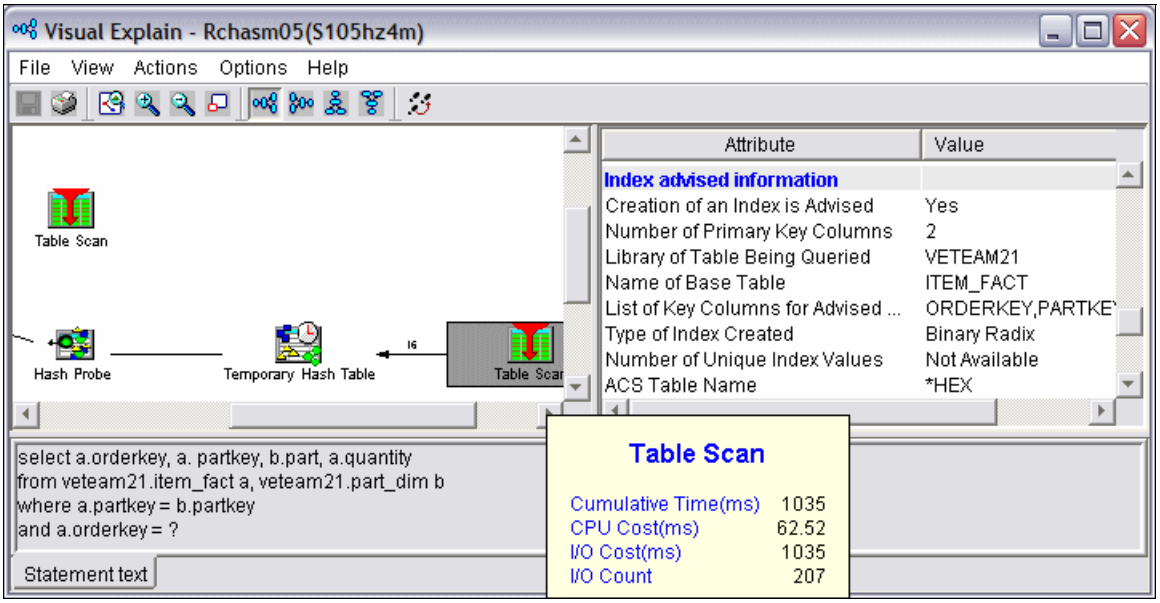


Figure 5-19 Table Scan information from visual Explain

Open Information report

The Open Information report contains a row of information, for each open activity, for each SQL statement. If an open operation occurs for the first time for a specific statement in a job, it is a *full open*.

A full open creates an ODP that is used to fetch, update, delete, or insert rows. Since there are typically 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 the same processing is not done on each subsequent I/O operation. An ODP might be cached at close time so that, if the SQL statement is run again during the job, the ODP is reused. Such an open is called a *pseudo open*, which is less expensive than a full open as shown in Figure 5-20.

Note: At least, the same statement must be executed more than two times to create a reusable ODP in the job.

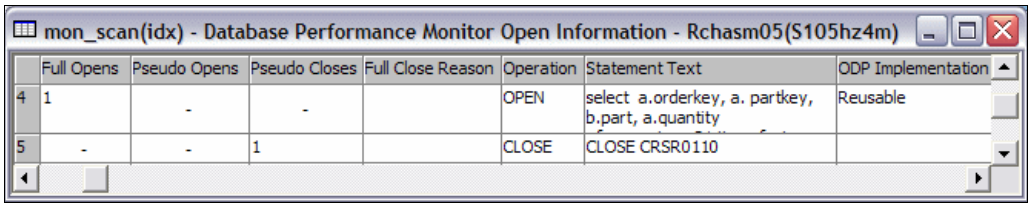


Figure 5-20 Open Information report

Analyzing full opens is one of the most important steps because an ODP creation process is expensive for query processing. The key point is determining why an SQL statement has an excessive number of full opens.

In our example, since the sample query is executed only one time, you can see that the open operation uses a full open that creates an ODP and the Full Opens column is set to “1”.

Otherwise, the Pseudo Opens column is set to “1”. The ODP Implementation column shows that an ODP will be reusable if same SQL statement is executed more than two times.

If an ODP is not reusable, the Operation column is set to “CLOSE(Hard)” instead of “CLOSE”. Since there are many reasons that the cursor is hard closed, you must look more carefully at the Full Close Reason column.

Access Plan Rebuild Information report

The Access Plan Rebuild Information report contains a row of information for each SQL statement that required the access plan to be rebuilt. Reoptimization is occasionally necessary for one of several reasons such as a new index being created or dropped, applying a program temporary fix (PTF), and so on. However, excessive access plan rebuilds might indicate a problem.

In our example, you can see the reason why the access plan was rebuild as shown in Figure 5-21.

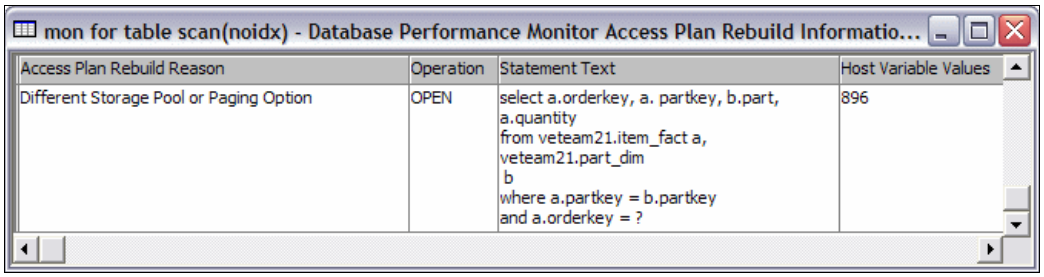


Figure 5-21 Access Plan Rebuild Information report

From this report, the most important information is in the Access Plan Rebuild Reason column. The following reasons are possible:

- Different File Or Member
- Reusable Plan to Non-Reusable Plan Change
- Non-Reusable To Reusable Plan Change
- More Than Ten Percent Change In Number Of Rows
- New Access Path Found
- Access Path No Longer Found Or Valid
- System Programming Change
- Different CCSID
- Different Date Or Time Format
- Different Sort Sequence Table
- Different Storage Pool or Paging Option
- Symmetric Multi Processing Change
- Different DEGREE
- Open View or View Materialization
- UDF or UDT Change
- QAQQINI Change
- Different Isolation Level or Scroll Option
- Different FETCH FIRST *n* ROWS
- First Run™ With Variable Values
- Different Constraints
- New Release

Index Advised Information report

Although it is possible to see the Advised Index keys from the Table Scan Information report, you can see the same information from the Index Advised Information report as shown in Figure 5-22.

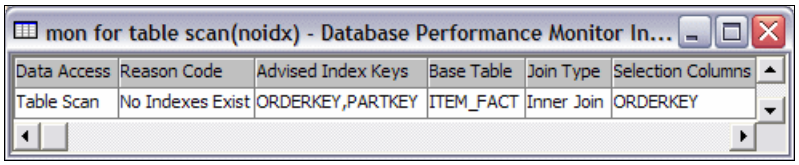


Figure 5-22 Index Advised Information report

This report show that since there are no indexes on the ITEM_FACT table, the optimizer chooses to use the Table Scan Data Access method. Notice that you can't see index advised information about PART_DIM table because the records included in this report are selected only if the Advise Index column is set to "Y" in the Table Scan information report.

Visual Explain can offer more accurate information about an advised index. In our SQL performance data, although you can't see index advised information about PART_DIM table, Visual Explain shows the Advised Index Key column information as shown in Figure 5-23.

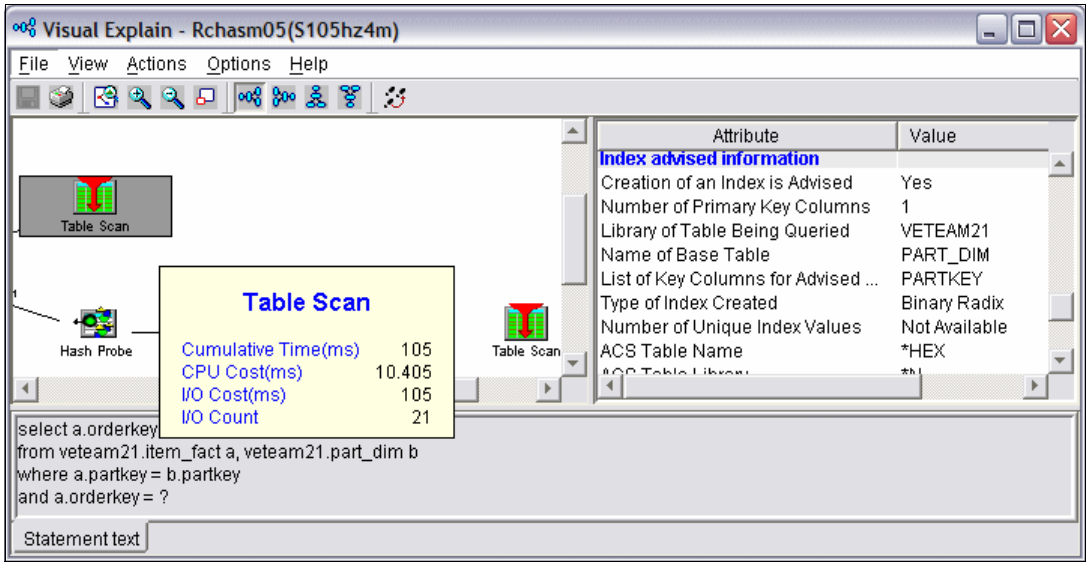


Figure 5-23 Advised index key on table without selection key columns

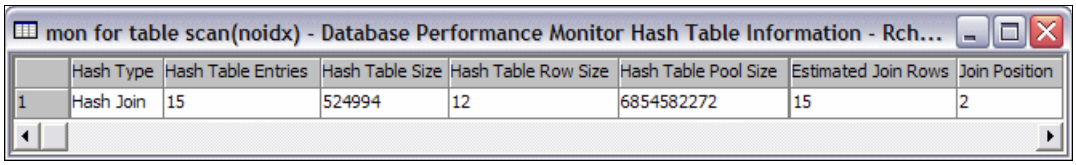
For more information about Visual Explain, refer to Chapter 8, "Analyzing database performance data with Visual Explain" on page 197.

Hash Table Information report

From the example in Figure 5-19 on page 109, you can see that the temporary hash table is built on the ITEM_FACT table for hash prove. To analyze the cause of the hash table creation, you can review the Hash Table Information report (see Figure 5-24).

This report shows information about any temporary hash tables that were used. The option is available only when you use a detailed SQL Performance Monitor. Hash tables are used in DB2 Universal Database for iSeries in two ways:

- For grouping
- For implementing the hash join algorithm



	Hash Type	Hash Table Entries	Hash Table Size	Hash Table Row Size	Hash Table Pool Size	Estimated Join Rows	Join Position
1	Hash Join	15	524994	12	6854582272	15	2

Figure 5-24 Hash Table Information report

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

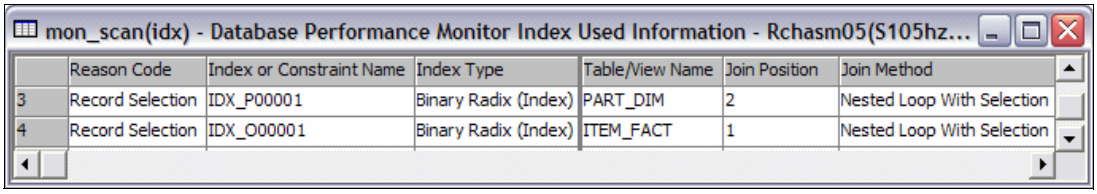
Some useful columns are:

- Hash table Entries
- Hash Table Size
- Hash table Row Size
- Hash Table Pool Size

Figure 5-24 shows that the temporary hash table is created for the hash join.

Index Used Information report

According to the information from the previous reports, perfect indexes were created on both tables. After creating new indexes, we ran the query again to see the effects of the indexes in the query. From the newly created SQL Performance Monitor data, you can see that the optimizer chooses to use existing indexes from the Index Used Information report as shown in Figure 5-25.



	Reason Code	Index or Constraint Name	Index Type	Table/View Name	Join Position	Join Method
3	Record Selection	IDX_P00001	Binary Radix (Index)	PART_DIM	2	Nested Loop With Selection
4	Record Selection	IDX_O00001	Binary Radix (Index)	ITEM_FACT	1	Nested Loop With Selection

Figure 5-25 Index Used Information report

This report shows the index that will be used to access the table and why it was chosen. Since the index was chosen for a nested loop join, additional information is given to help determine how the table fits in the join in Join Position and Join Method columns.

You can see same result in the Visual Explain diagram as shown in Figure 5-26. The index probe access method was chosen by optimizer, although you cannot see the temporary hash table any more in the Visual Explain diagram. The optimizer also does not advise an additional index for better performance.

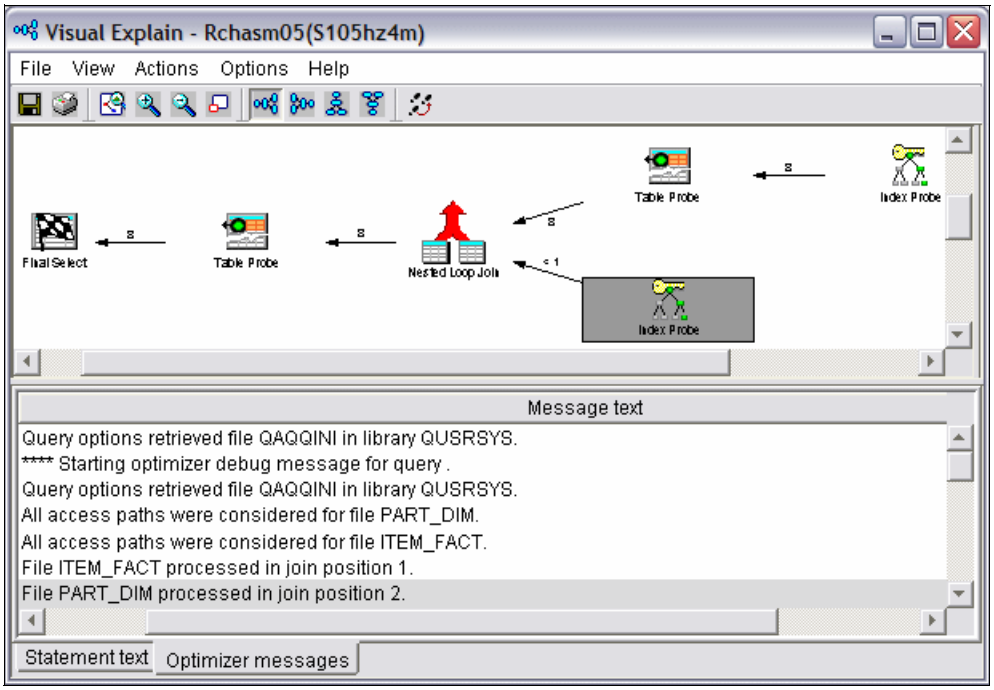


Figure 5-26 Visual Explain diagram showing effects of indexes

The following information is the most commonly used in this report:

- Reason Code: Why the optimizer chooses to use indexes, which might be for the following possible reasons:
  - Selection only
  - Ordering or grouping
  - Nested loop join
  - Record selection using bitmap
  - Bitmap selection
- Index or Constraint Name
- Index Type: Binary Radix Index or encoded-vector index (EVI)
- Join Position: Join position of each tables.

Index Create Information report

The Index Create Information report contains a row of information for each SQL statement that required an index to be created. Temporary indexes might need to be created for several reasons such as to perform a join, to support scrollable cursors, or to implement Order by or Group by. The indexes that are created might contain keys only for rows that satisfy the query (such indexes are known as *sparse indexes*). In many cases, index creation might 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 can create a permanent index to improve performance of this query. This might be true regardless of whether an index was advised.

A temporary index build does *not* always mean that the ODP is nonreusable. The database optimizer tries to reuse the temporary index for each execution of the specific query if possible.

For example, if host variable value is used to build selection into a temporary index (that is, sparse), then ODP is not reusable because temporary index selection can be different on

every execution of the query. In this case, the optimizer tends to avoid creating *sparse* indexes if the statement execution history shows it to be a frequently executed statement. But if the temporary index build is done during repeated query run times and the query goes into reusable ODP mode, then the temporary index *is* reusable.

Note: Temporary indexes are not usable by other ODPs.

If a particular query is run multiple times and a temporary index is built each time, a permanent index must be created to avoid the index build and to make the ODP reusable.

Temporary indexes are never built for selection alone. They always involve a join or a group by or order by clause. Furthermore, since no name is given to the temporary index, *TEMP is used in subsequent monitor data.

To show an example, we use another sample query because the temporary index can be created only if the optimizer chooses to use CQE for execution. Example 5-2 shows the sample SQL statement.

Example 5-2 Sample query statement using CQE

```
  select a.orderkey, max(a.quantity) as qty
    from veteam23.item_fact a, veteam23.part_dim b
   where a.partkey = b.partkey
     and a.orderkey >= '800'
     and a.orderkey <= '900'
     and shipmode like 'T%'
group by a.orderkey
order by qty;
```

From the monitor data, you can see the Index Create Information report as shown in Figure 5-27.

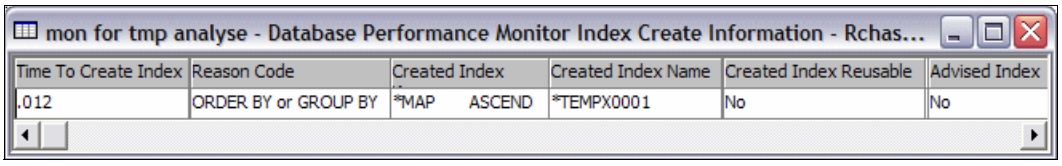


Figure 5-27 Index Create Information report

In this report, you can see that the Reason Code for index creation is the ORDER BY or GROUP BY clause in SQL statement. But you can't see any Advised Index information from this report because a temporary index was built over a temporary table. If a temporary index is built over a real table, you can see the advised index keys and create a perfect index using advised keys.

You can also see the temporary index creation process from the Visual Explain diagram as shown in Figure 5-28. In our example, the index build can't be avoided if *MAP is one of the keys listed, but this does not mean that the ODP is nonreusable. Notice that the purpose of this example is only for getting the monitoring report. To tune this sample query, you must first analyze the reason why temporary hash table was built on PART_DIM table.

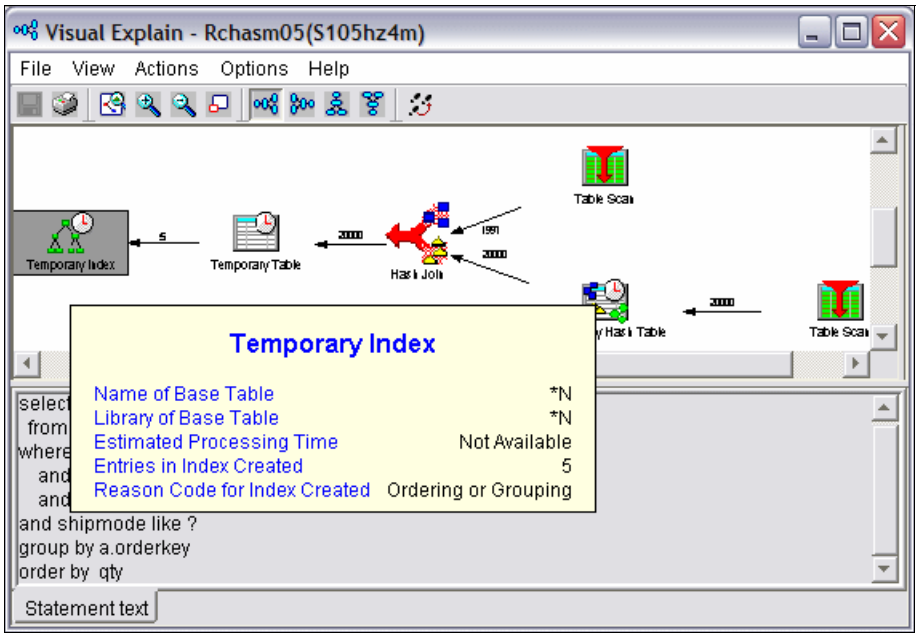


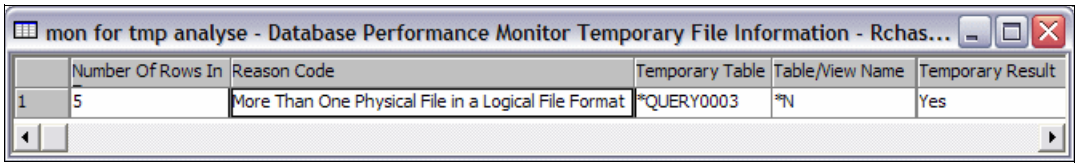
Figure 5-28 Visual Explain diagram showing Temporary Index creation

Temporary File Information report

Figure 5-28 shows that temporary index is built over a temporary table. From the monitoring data, you can see that the Temporary File Information report 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 might want to investigate why the temporary result is necessary. In some cases, you can modify the SQL statement to eliminate the need for the temporary result. For example, if a cursor has an attribute of INSENSITIVE, a temporary result is created. Eliminating the keyword INSENSITIVE usually removes the need for the temporary result, but your application sees changes as they occur in the database tables.

In our example, you can see that the temporary file was built because a join condition between two tables requires a temporary file for further processing as shown in Figure 5-29.



	Number Of Rows In	Reason Code	Temporary Table	Table/View Name	Temporary Result
1	5	More Than One Physical File in a Logical File Format	*QUERY0003	*N	Yes

Figure 5-29 Temporary File Information report

Sort Information report

The Sort Information report contains a row of information for each sort that an SQL statement performed. In our example, you can see that the optimizer chooses to use query sort processing to obtain the final data as shown in Figure 5-30.

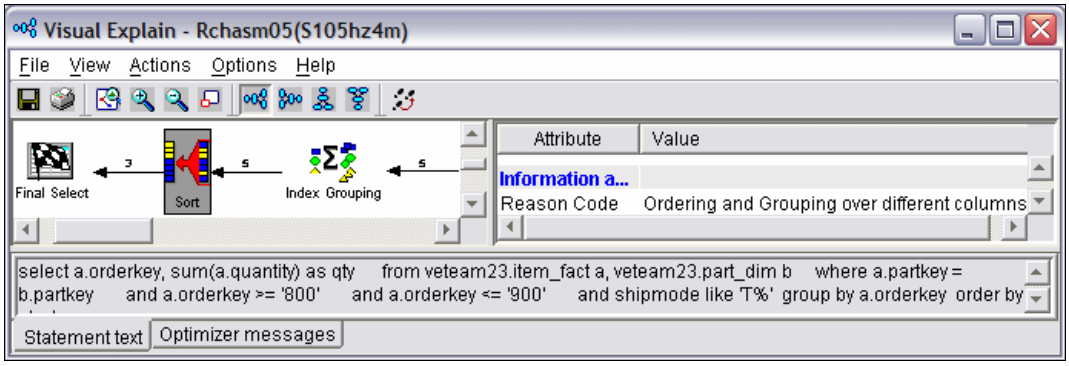


Figure 5-30 Query Sort report in the Visual Explain diagram

You can see the same results in the Sort Information report shown in Figure 5-31. Notice that the reason for Sort is GROUP BY and ORDER BY Columns Different.

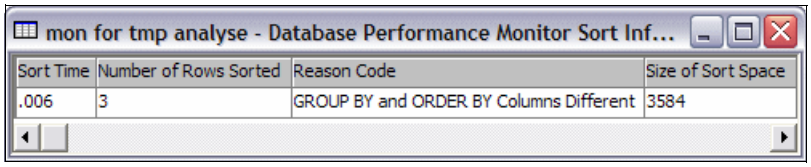


Figure 5-31 Sort Information report

Sorts of large result sets in SQL statement can be a time-consuming operation. In some cases, an index can be created that eliminates the need for a sort.

Group By Information report

From the example in Figure 5-30, you can see that Index Grouping method is chosen by the optimizer. You can see the information about the GROUP BY process in the Group By Information Report as shown in Figure 5-32.

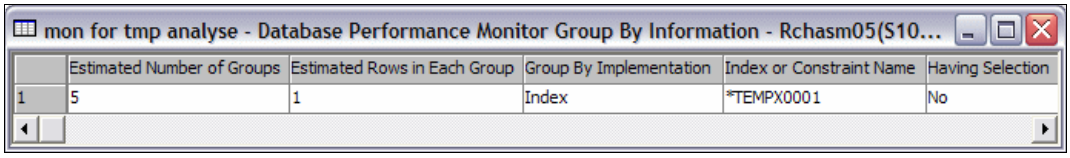


Figure 5-32 Group By Information report

This report shows that the temporary index is used for the GROUP BY process. This option is available only when you use a detailed SQL Performance Monitor.

Other reports

The additional reports in the following sections are useful for query and SQL performance analysis.

Governor Timeout Information report

The Governor Timeout Information report provides information about all optimizer timeouts. This option is available only when you use a detailed SQL Performance Monitor. You can use this report to determine how often users attempt to run queries that exceed the governor timeout value. A large number of governor timeouts might indicate that the timeout value is set too low.

Optimizer Timeout Information report

The Optimizer Timeout Information report provides information about any optimizer timeouts. This option is available only when you use a detailed SQL Performance Monitor. Choosing the best access plan for a complex query can be time consuming. As the optimizer evaluates different possible access plans, a better estimate of how long a query takes shape.

At some point, for dynamic SQL statements, the optimizer might decide that further time spent optimizing is no longer reasonable and use the best access plan up to that point. This might not be the best plan available. If the SQL statement runs only a few times or if the performance of the query is good, an optimizer timeout is not a concern. However, if the SQL statement is long running, or if it runs many times and the optimizer times out, a better plan might be possible by enabling extended dynamic package support or by using static SQL in a procedure or program. Since many dynamic SQL statements can be cached in the system-wide statement cache, optimizer timeouts are not common.

Procedure Call Information report

The Procedure Call Information report provides information about procedure call usage. This option is available only when you use a detailed SQL Performance Monitor. Performance of client/server or Web-based applications is best when the number of round trips between the client and the server is minimized, because the total communications cost is minimized. A common way to accomplish this is to call a procedure that performs a number of operations on the server before it returns results, rather than sending each individual SQL statement to the server. Figure 5-33 shows a sample report.

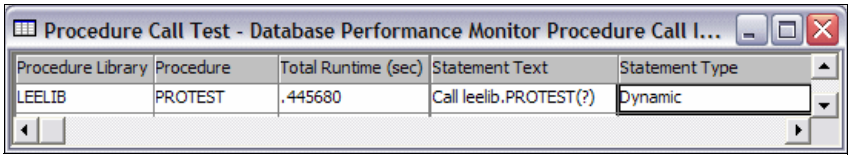


Figure 5-33 Procedure Call Information report

Distinct Processing Information report

The Distinct Processing Information report provides information about any DISTINCT processing. This option is available only when you use a detailed SQL Performance Monitor. SELECT DISTINCT in an SQL statement might be a time consuming operation because a final sort might be necessary for the result set to eliminate duplicate rows. Use DISTINCT in long running SQL statements only if it is necessary to eliminate the duplicate resulting rows.

Data Conversion Information report

The Data Conversion Information report 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 of the result is returned to DECIMAL, the data must be converted from integer to decimal. A single data conversion operation is inexpensive. However, if the

operation is repeated thousands or millions of times, it can become expensive. In some cases, it is easiest to change one attribute so a faster direct map can be performed. In other cases, conversion is necessary because no exact matching data type is available.

In our example, you can see that data conversion is caused by Different Numeric Types in the insert operation as shown in Figure 5-34.

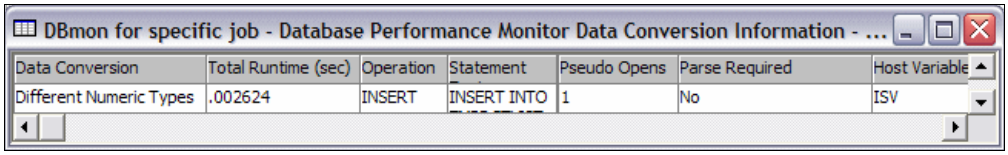


Figure 5-34 Data Conversion Information report

Subquery Information report

The Subquery Information report contains a row of subquery information. This information can indicate which subquery in a complex SQL statement is the most expensive.

Row Access Information report

The Row Access Information report contains information about the rows returned and I/Os performed. This option is available only when you use a detailed SQL Performance Monitor. This information can indicate the number of I/Os that occur for the SQL statement. A large number of physical I/Os can indicate that a larger pool is necessary or that the Set Object Access (SETOBJACC) command might be used to bring some of the data into main memory.

Lock Escalation Information report

The Lock Escalation Information report provides information about any lock escalation. This option is available only when you use a detailed SQL Performance Monitor. In a few rare cases, a lock must be escalated to the table level instead of the row level. This can cause much more contention or lock wait timeouts between a user who is modifying the table and the reader of the table. A large number of lock escalation entries might indicate a contention performance problem.

Bitmap Information report

The Bitmap Information report provides information about any bitmap creations or merges. This option is available only when you use a detailed SQL Performance Monitor. Bitmap generation is typically used when performing index ANDing or ORing. This typically is an efficient mechanism.

Union Merge Information report

The Union Merge Information report provides information about any union operations. This option is available only when you use a detailed SQL Performance Monitor. UNION is a more expensive operation than UNION ALL because duplicate rows must be eliminated. If the SQL statement is long running, make sure it is necessary that the duplicate rows be eliminated.

Error Information report

The Error Information report provides information about any SQL statement error messages and warnings that were captured by the monitor as shown in Figure 5-35.

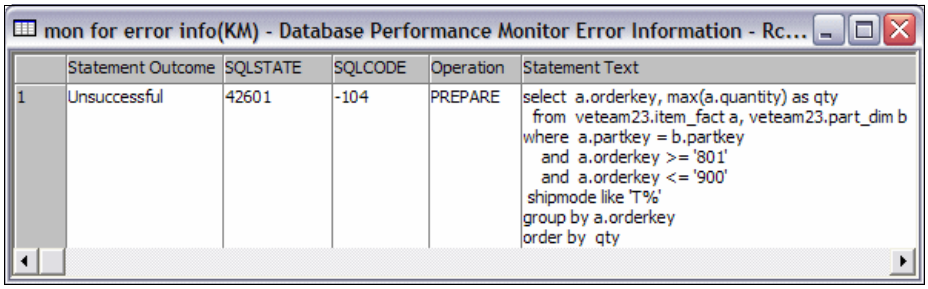


Figure 5-35 Error Information report

If you receive an error message from your application, the error information report provides SQLSTATE and SQLCODE. From these messages, you can analyze and correct errors.

Start and End Monitor Information report

The Start and End Monitor Information report provides information about any start and end Database Monitor operations. This option is available only when you use a detailed SQL Performance Monitor.

5.3 Modifying a predefined query report

The SQL Performance Monitor also lets you retrieve the SQL statements of any of these reports to use as a starting point for creating your own analysis reports. Although the predefined queries can be materialized and customized, at first glance, they can be very intimidating. Even power SQL users might be somewhat reluctant to make changes to these statements.

This section explains how you can modify predefined queries and make your own reports. Notice that the original query used by iSeries Navigator is not changed. You go step-by-step through the process to customize a predefined query in iSeries Navigator.

1. From the Analyze Results window (Figure 5-36), select a query to modify. In this example, we modify the Basic statement information query.
   a. Click the **Extended Detailed Results** tab.
   b. Under Select extended detailed queries, select the **Basic statement information** query and click the **Modify Selected Queries** button.

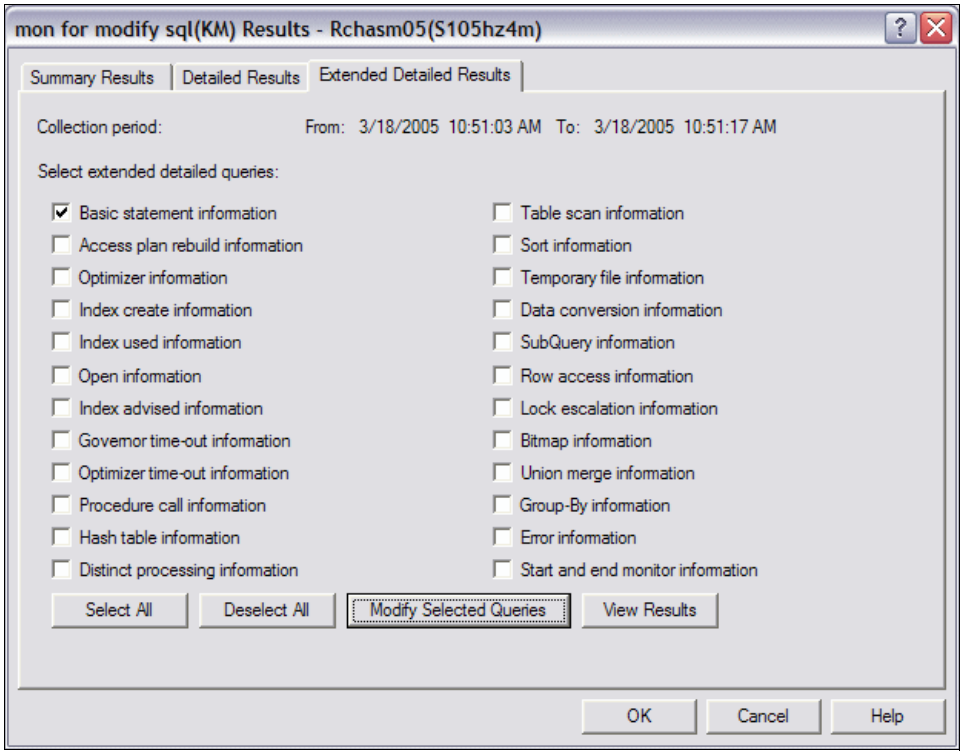


Figure 5-36 Analyze Results window for customizing query

You see the SQL source for the query in the Run SQL Script window (Figure 5-37).

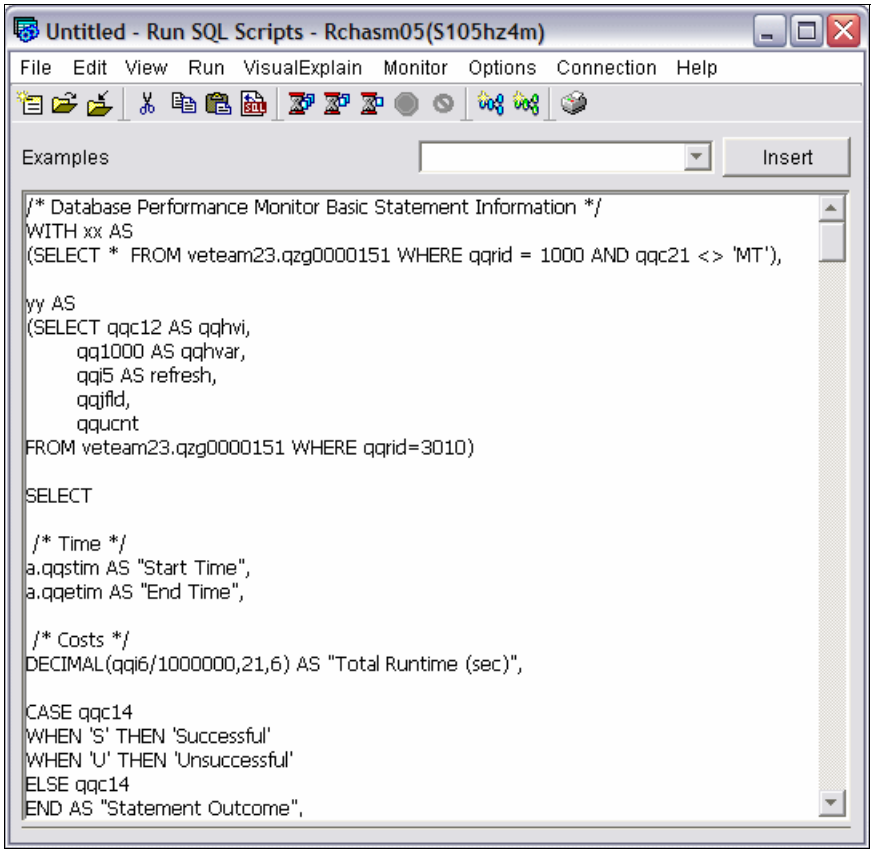


Figure 5-37 Basic statement information query on Run SQL Script window

2. In this example, we are trying to find top five most time-consuming queries from the report. In the Run SQL Script window, scroll to the end of the query and add the following line (see Figure 5-38):

   ```
   FETCH FIRST 5 ROWS ONLY;
   ```

   Click the **Run Query** icon from the toolbar (circled icon). The result appears at the lower half of panel.

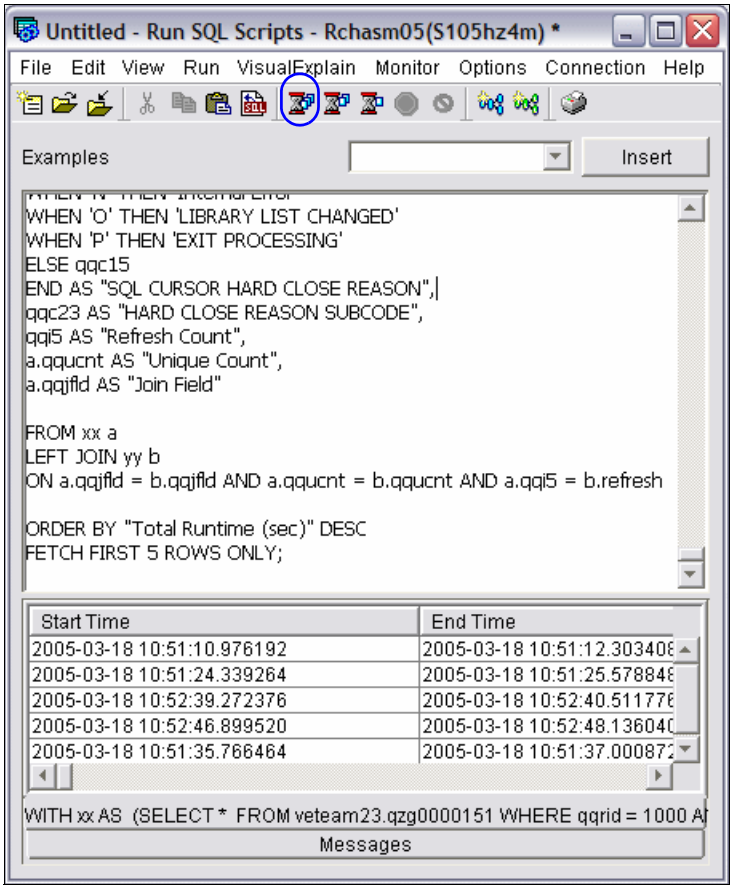


Figure 5-38 Modifying Pre-defined report

You can customize this report further. For example, you can omit columns from the report in which you are not interested. You can also add grouping and totaling functions at the end of the query. For example, if you want to know only the query statements and total runtime accounting for the most runtime listed by each statements, you can customize query to insert a grouping clause (by Total run time) and get a result like the one shown in Figure 5-39.

This query shows individual query instances and the amount of time they took, sorted by run time (largest to smallest). It does not include non-ODP SQL operations such as prepare or describe (QQUCNT=0 and QQSTN = 0). This provides a way to quickly find a query that is taking a large amount of time with knowing any text about that query.

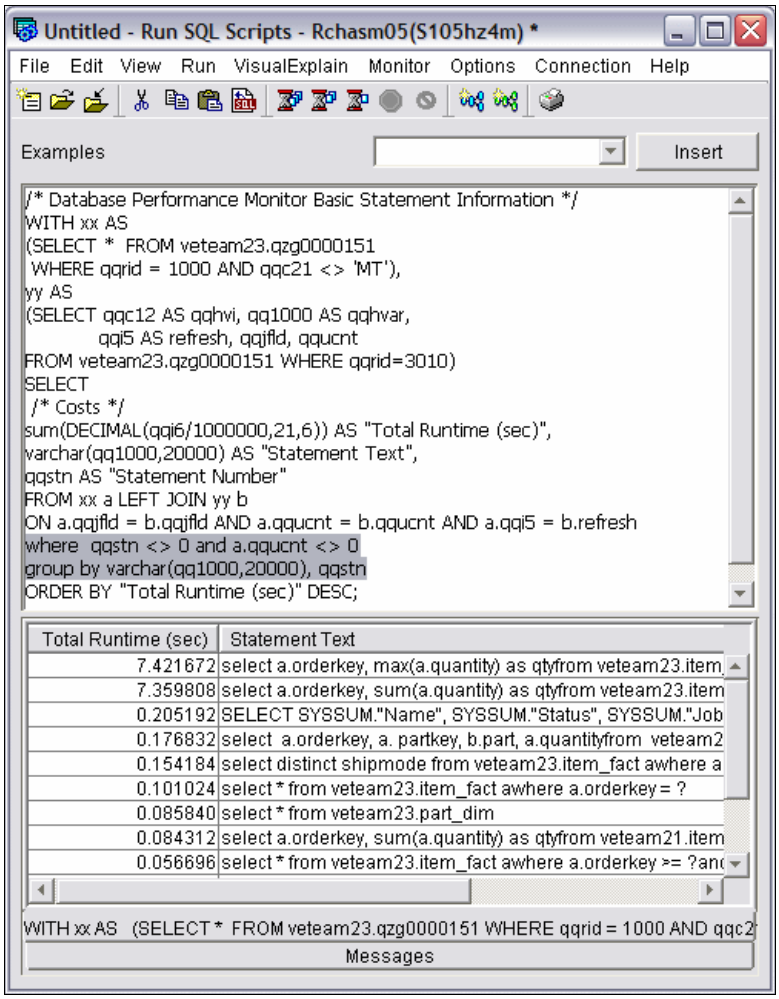


Figure 5-39 User Customized report

5.4 Query tuning example with SQL Performance Monitor reports

Now we take you through a query tuning example using the Extended Detailed Reports. There are two approaches to tuning the queries: *a proactive approach* and *a reactive approach*. As the name implies, *proactive tuning* involves anticipating which queries will be most often used for selection, joining, grouping, and ordering. In the *reactive tuning*, the queries are tuned based on optimizer feedback, the query implementation plan, and system performance measurements. In practice, both methods will be used iteratively.

As the numbers of users increase, more indexes are useful. Also, as the users become more adept at using the application, they might start using additional columns that require more indexes. Therefore, perfect index creation is the most effective tuning method.

The following example shows how the query is tuned by analyzing the Extended Detailed Performance Monitor reports. This provides mainly a proactive approach for index creation. For the purpose of this example, we assume that the sample query shown in Example 5-2 on page 114 is the most time consuming and the object of query tuning. No index exists over both tables used in sample query.

Figure 5-40 shows that the detailed SQL Performance Monitor is started and data is collected from the Run SQL Script center.

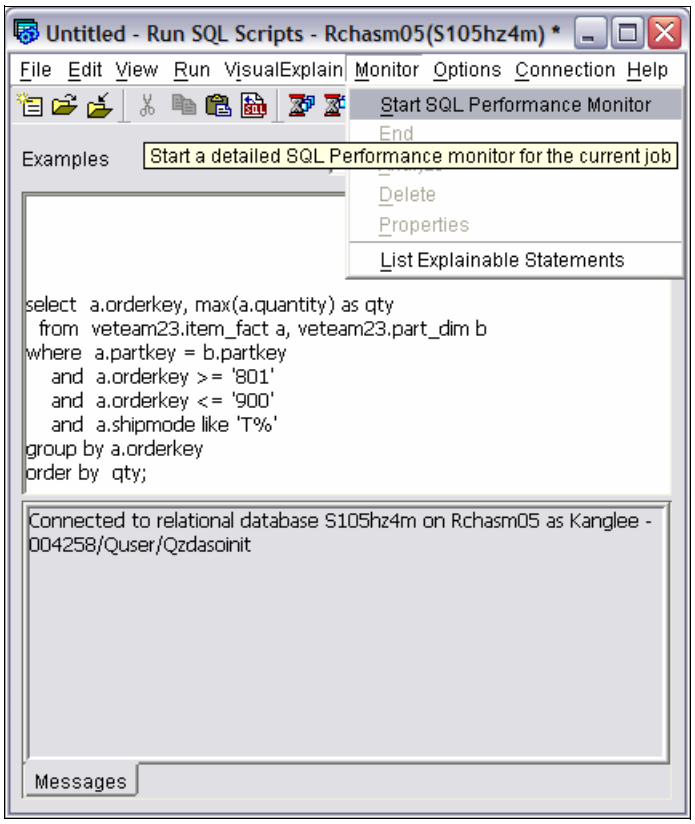


Figure 5-40 Start SQL Performance Monitor

First we need to know the overall access plan chosen by the optimizer to find the starting point for tuning. Since Visual Explain is the most useful tool to see an access plan, look at the Visual Explain diagram in Figure 5-41, which shows a somewhat complex access plan.

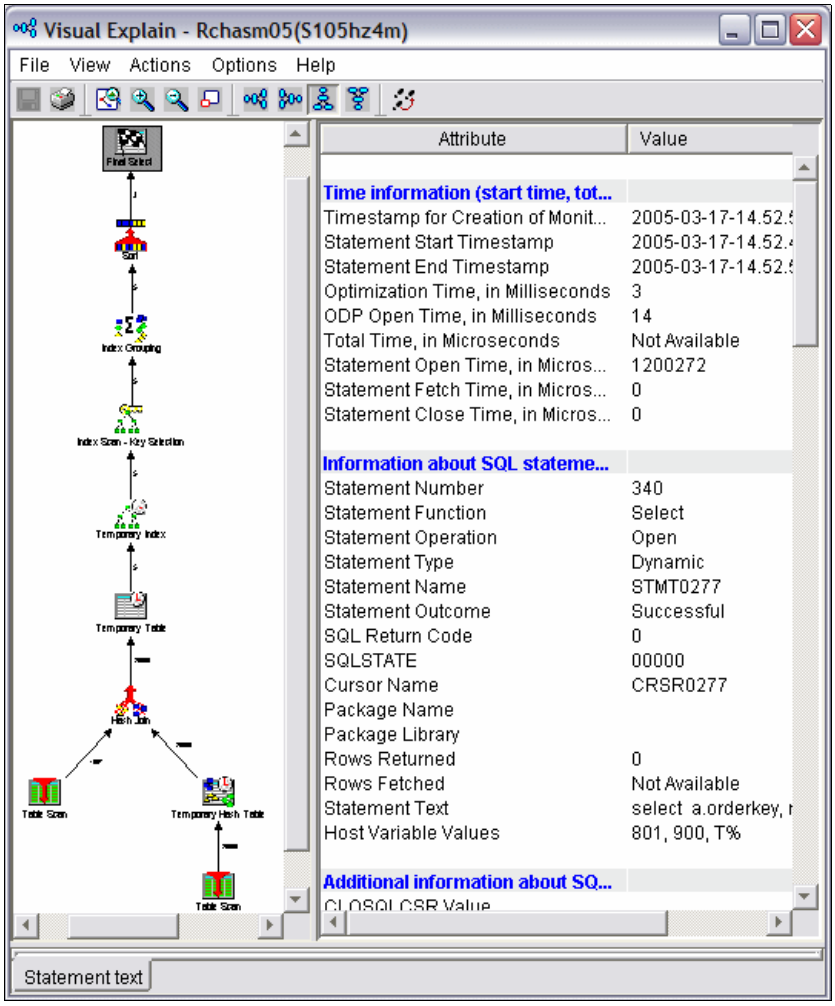
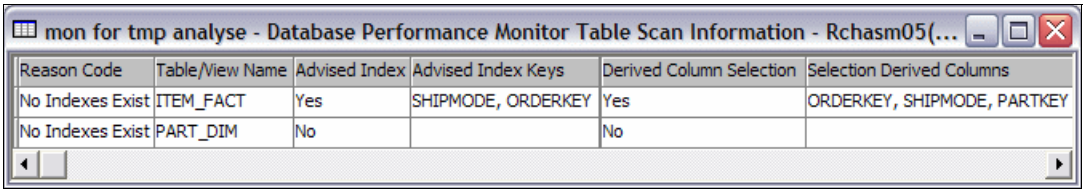


Figure 5-41 Overall access plan for a sample query

First step

The first access method chosen by the optimizer is a table scan over both tables. To understand why a table scan is chosen, you can see the Table Scan Information report (Figure 5-42).



Reason Code	Table/View Name	Advised Index	Advised Index Keys	Derived Column Selection	Selection Derived Columns
No Indexes Exist	ITEM_FACT	Yes	SHIPMODE, ORDERKEY	Yes	ORDERKEY, SHIPMODE, PARTKEY
No Indexes Exist	PART_DIM	No		No	

Figure 5-42 Table Scan Information report for tuning

This report shows that the reason for table scan is *No Index Exists*. According to the Advised Index Keys column, we can choose index keys for the ITEM_FACT table. The index key columns for local selection are SHIPMODE, ORDERKEY of the ITEM_FACT table.

Second step

Because the report from monitoring data can advise an index key only about Selection Key columns, you can't see the advised index keys for the PART_DIM table. Since the Join Key columns can also be an Index Key column, we must consider information about the join operation. From the Optimizer Information report (Figure 5-43), you can see that the hash join method is chosen.

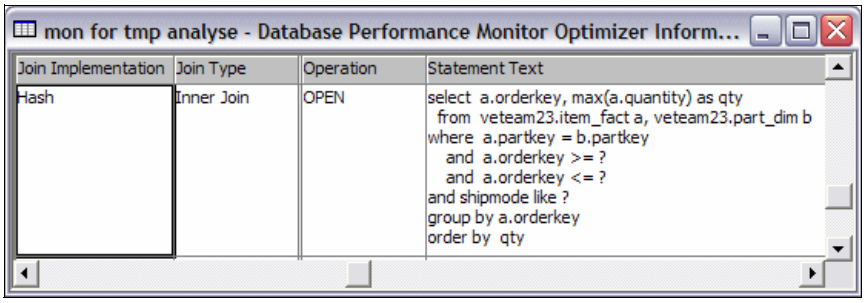


Figure 5-43 Optimizer Information report showing Hash join condition

The hash table from the PART_DIM table is created for the hash join operation as shown in Figure 5-41. Since the Join Key columns of both tables are PARTKEY, you can select the Join Key column of both tables as the Index Key column. Notice that hash table creation for the join operation can be one of the useful candidates for index creation. The Index Key column for join is the PARTKEY of the ITEM_FACT and PART_DIM tables.

From the results, we create indexes for both tables, and the order sequence of the columns is:

- ITEM_FACT: SHIPMODE, ORDERKEY, PARTKEY(IDX_TMP6)
- PART_DIM: PARTKEY(IDX_TMP7)

To see the result of applying new indexes, new Extended Detailed SQL Performance Monitor data is collected again. After we apply new the indexes over both of the tables, we can't see records any more in the Table Scan Information report. However, the Index Used Information report shows that the index scan method is chosen by optimizer as shown in Figure 5-44.

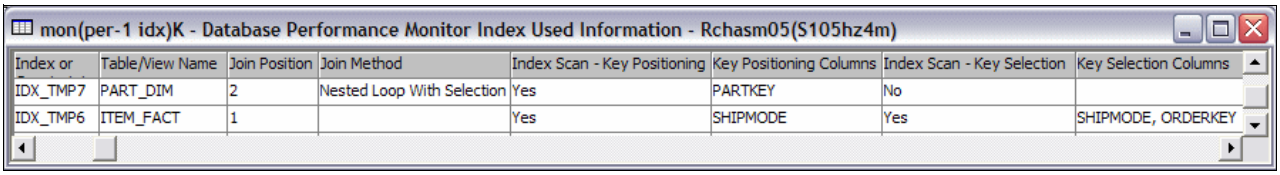


Figure 5-44 Index Used Information report after applying new indexes

The Optimizer Information report (Figure 5-45) also shows that the hash join is no longer used.

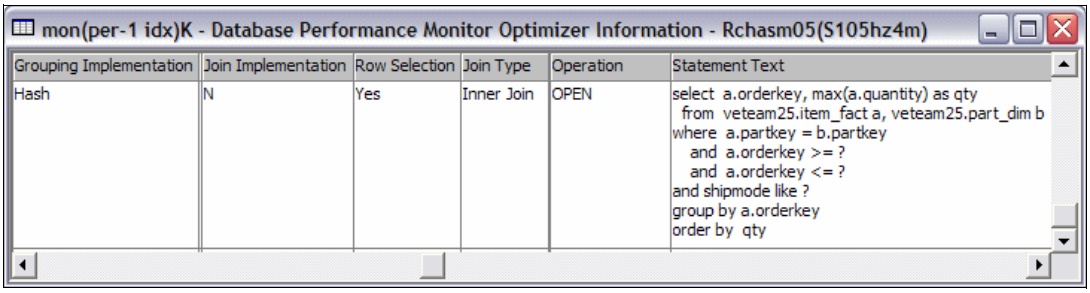


Figure 5-45 Optimizer Information report after applying indexes

You can see the same result from the Visual Explain diagram as shown in Figure 5-46.

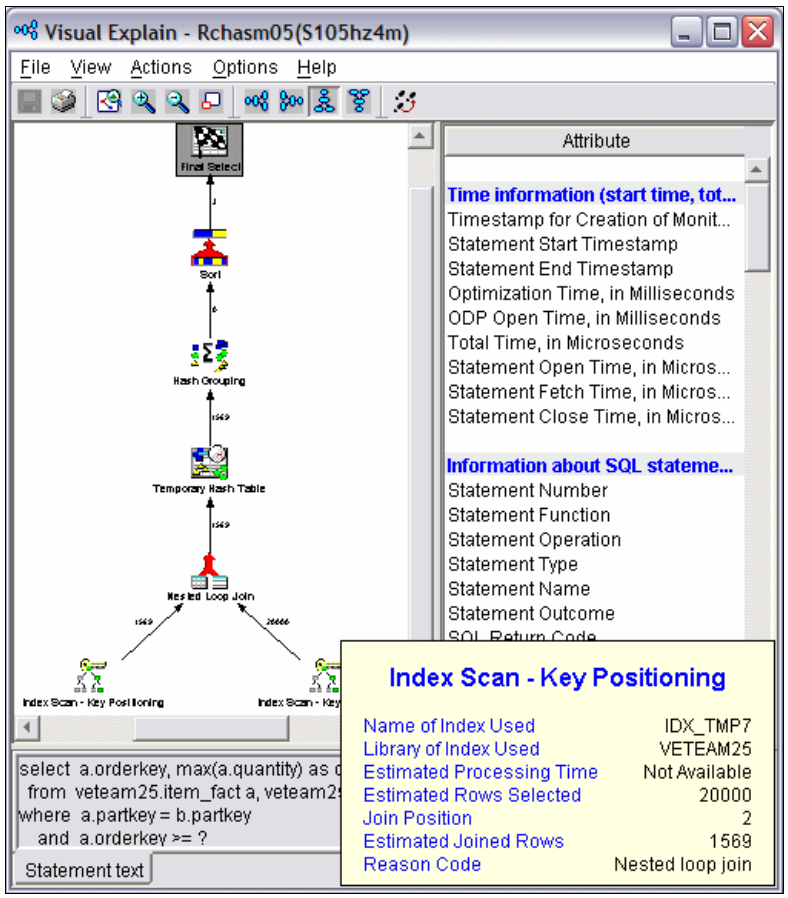


Figure 5-46 Visual Explain diagram after applying indexes

Third step

Figure 5-46 shows that the optimizer still uses the Hash Grouping and Sort method. Therefore, we must apply a new index for the grouping and sorting columns. You can see information about each column from Group By information and Sort Information report.

From these reports, a new index is created over ITEM_FACT table and the key columns are ITEM_FACT: ORDERKEY, QUANTITY(IDX_TMP8).

From the finally collected SQL Performance Monitor data, the Index Used Information report shows that Hash Grouping is changed to Index Grouping as shown in Figure 5-47.

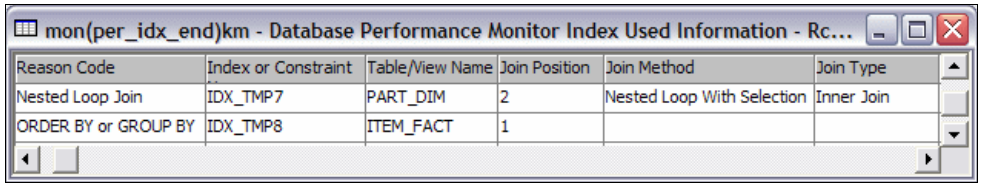


Figure 5-47 Index Used Information showing Index Grouping method

Although we applied the IDX_TMP8 index, the optimizer still uses the Sort method to get a final result as shown in Figure 5-48.

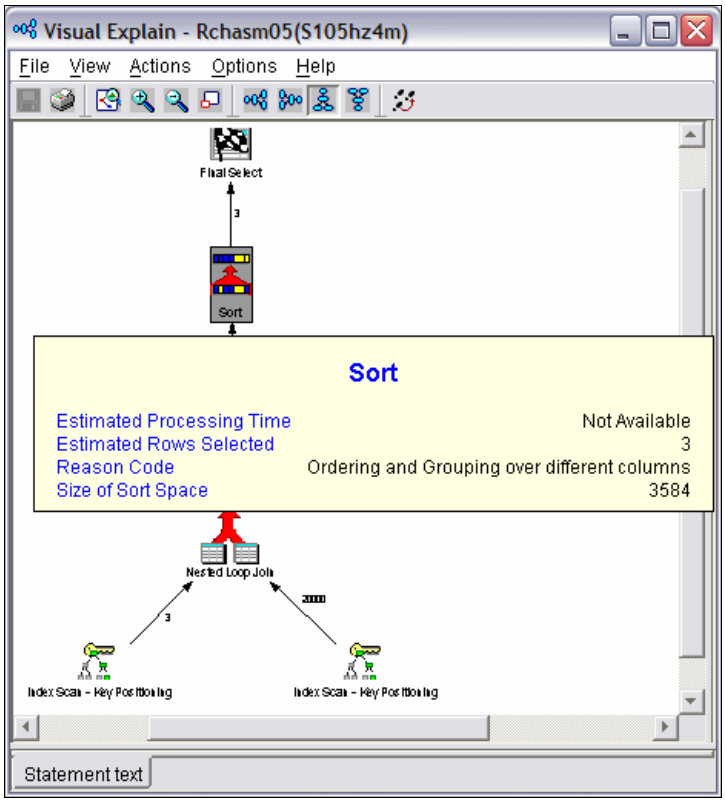


Figure 5-48 Visual Explain diagram after applying the IDX_8 index

You can find the reason in the Sort Information report as shown in Figure 5-49.

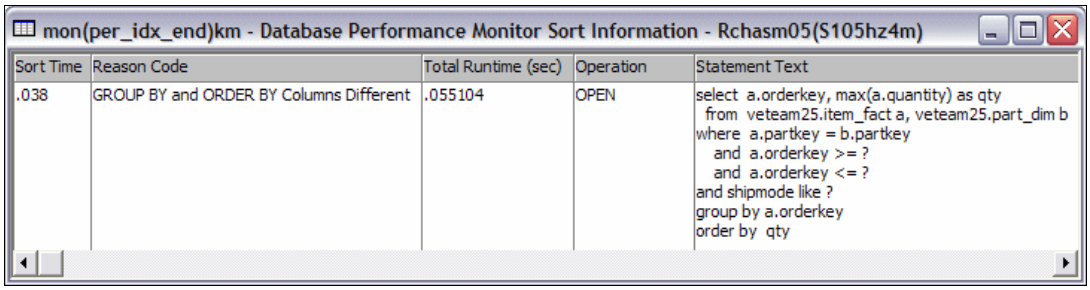


Figure 5-49 Sort Information report after applying the IDX_8 index

From this report, we can see that the Sort method cannot be avoided because two different columns are used for sorting.

Finally, we can see the OPEN process time from Basic Statement Information report as shown in Figure 5-50.

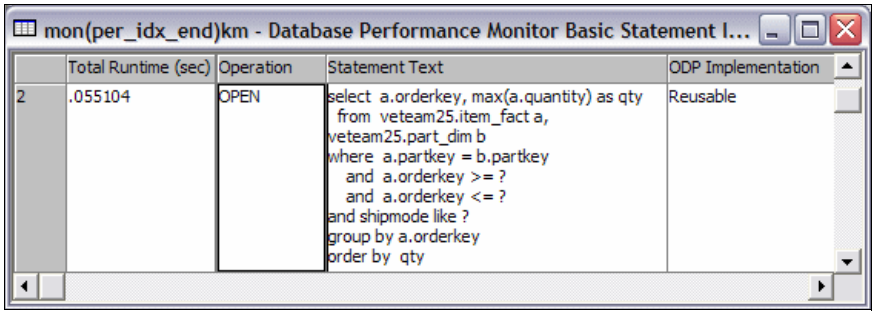


Figure 5-50 Open process time after query tuning

Compared to the Basic Statement Information report, which shows the total run time before applying indexes as shown in Figure 5-51, you can see that the OPEN process time has decreased by less than 20 times.

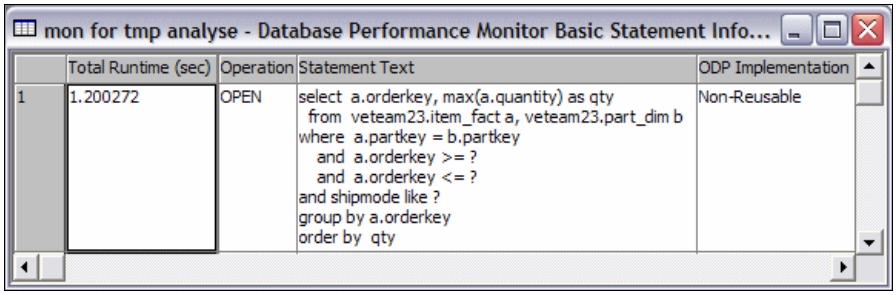


Figure 5-51 OPEN process time before applying indexes

5.4.1 List Explainable Statements

The List Explainable Statements option from SQL Performance Monitor pane lists the SQL statements for which a detailed SQL Performance Monitor has collected data and for which a Visual Explain graph can be produced.

To access this function:

1. Click **iSeries Navigator** → **Database** → **SQL Performance Monitors**.
2. From the list of the SQL Performance Monitors that are currently on the system, right-click a **detailed collection** and select **List Explainable Statements**, as shown in Figure 5-52.

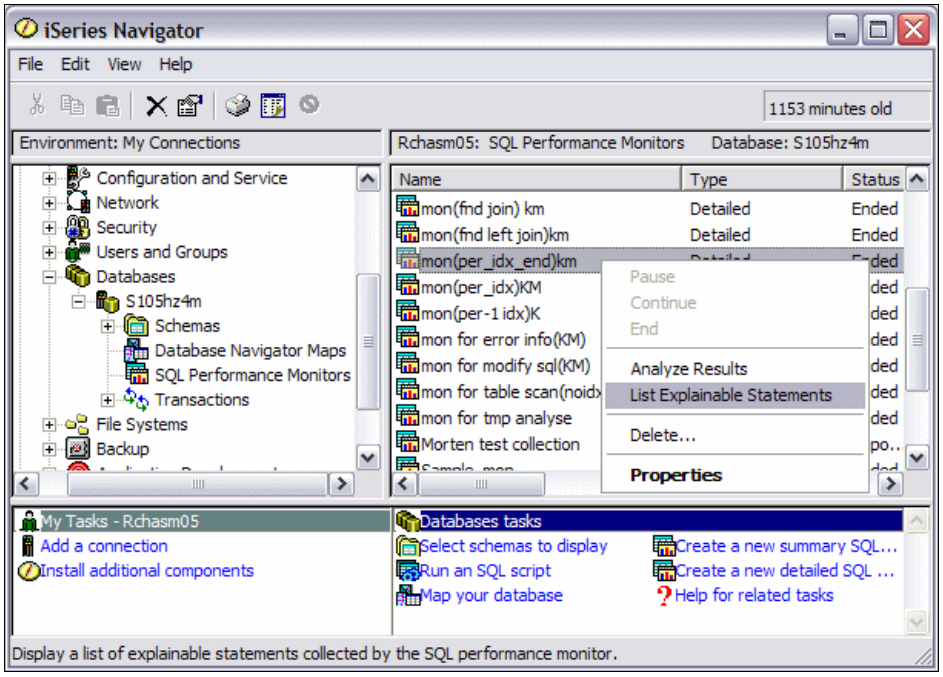


Figure 5-52 Selecting the List Explainable Statements option

3. In the Explainable statements window (Figure 5-53), the upper half of the panel shows the SQL statements that are monitored during the data collection session.

   a. Click to select a statement that you are interested in analyzing. The selected statement is then displayed in the lower half of the panel.

   b. With the statement in focus, it is possible to have it analyzed and explained. Click the **Run Visual Explain** button. For more information about “Visual Explain”, refer to Chapter 8, “Analyzing database performance data with Visual Explain” on page 197.

   c. With V5R3, since the columns in the upper half of the panel are sortable, you can rearrange the data by clicking column head tab that you want to need to analyze. In our example, we click the Time column to sort the columns by time sequence.

   As you might have already noticed, the database analysis options and tools provided by iSeries Navigator are well interconnected and meant to be used together in an iterative manner.

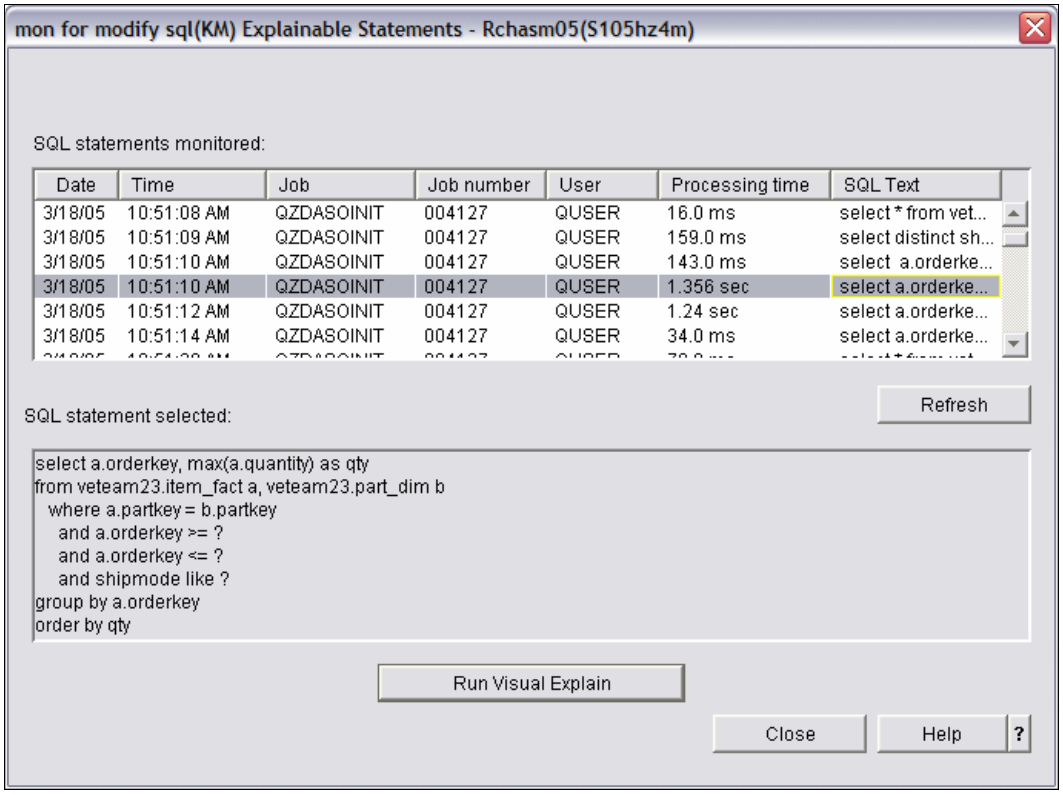


Figure 5-53 Using List Explainable Statements

6

Querying the performance data of the Database Monitor

In the previous chapters, we explained how the Detailed Database Monitor dumps all the performance data into one table. We also explained the different columns in the table. Plus we illustrated that the iSeries Navigator interface has some predefined reports that you can use to understand and identify possible SQL performance problems.

In this chapter, we explain how to directly query the Database Monitor data. This is useful because you can make your own queries that might not be available through the graphical interface.

6.1 Introduction to query analysis

The Detailed Database Monitor table can be analyzed by using SQL. This is a time consuming approach unless you have predefined queries. A lot of predefined queries exist via the iSeries Navigator interface as explained in Chapter 5, “Analyzing database performance data using iSeries Navigator” on page 93, but you can also write you own queries.

In this chapter, we present several queries to help you analyze the database performance data. You can run the queries under all SQL interfaces that access the iSeries server. The green-screen interface is intentionally not selected, because it can have a major negative performance impact on some server models with reduced interactive capacity. All queries in this chapter are run through the Run SQL Scripts window in iSeries Navigator.

To start the interface, in iSeries Navigator, select **Databases** → **your database**, right-click, and select **Run SQL Scripts** (see Figure 6-1).

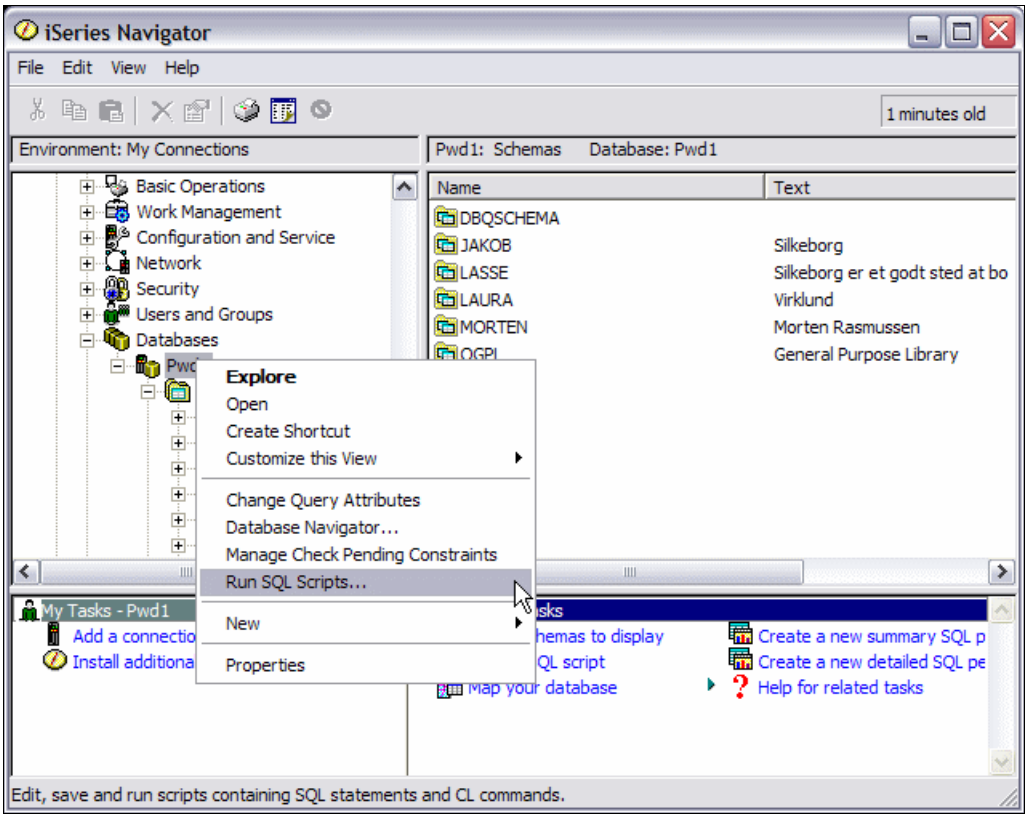


Figure 6-1 Starting the Run SQL Scripts option

6.2 Tips for analyzing the Database Monitor files

In the following sections, we present different tips to make it easier to analyze the performance data using custom-made queries. The idea is to let you copy the different SQL requests, so you can use them in your own analysis.

6.2.1 Using an SQL ALIAS for the Database Monitor table

By creating an SQL alias for the Detailed Database Monitor table that you are analyzing, you can use the same names for the analysis. When you analyze the next Detailed Database Monitor table, you use the SQL DROP ALIAS statement and then create an SQL ALIAS statement with the same name over the other table.

If you want to use DBMONLIB.MYDBMON, then use the following SQL CREATE ALIAS statement:

```
CREATE ALIAS DBMONLIB.MYDBMON FOR ibmfr.LAURA1608;
```

Before you analyze the next Database Monitor data, be sure to enter the SQL DROP ALIAS statement:

```
DROP ALIAS DBMONLIB.MYDBMON
```

6.2.2 Using a subset of the Database Monitor table for faster analysis

The Database Monitor table often is large and contains information about many jobs. Therefore, running queries on the data can sometimes be slower than desired. You can try to reduce the time that the queries take by collecting only the job that you want. However, sometimes this is not possible and, even if it is, batch jobs can generate a lot of Database Monitor output. Also, using interactive tools, such as Start SQL (STRSQL), can result in longer run times on server models.

If the response time is slow during the analysis, consider the following tips:

- Before you start the analysis, see how big the output table is for the collected Database Monitor.
- Create a smaller table from the main Database Monitor table with only the rows in which you are interested. You can use the following technique:

  ```
  CREATE TABLE smaller-table AS (SELECT * FROM big-dbmon-table
  WHERE QQJNUM IN('job-number','job-number'....) WITH DATA
  ```

- Another way to reduce the Database Monitor data is to include a time range in the SQL selection criteria, for example:

  ```
  and qqtime > '2005-03-16-12.00.00.000000' and qqtime < '2005-03-16-12.05.00.000000'
  ```

  This shows only five minutes of collection.

You can adjust these techniques as needed to speed up your analysis.

6.2.3 Using SQL views for the Database Monitor table

When using queries to analyze the Database Monitor table, you can make the queries more readable by using views. For example, we look at a query that shows table scans, which can be through a simple view, making it easier to see an overview of the query.

Example 6-1 shows the query before we create a view.

Example 6-1 Query before using an SQL view

```
WITH tablescans AS (
  SELECT qqjfld,qqucnt,qqrest,qqtotr
  FROM MYDBMON
  WHERE qqrid=3000)
SELECT SUM(qqi6) "Total Time", COUNT(*) "Times Run",
  a.qqucnt, integer(avg(b.qqrest)) "Est Rows Selected",
  integer(avg(b.qqtotr)) "Total Rows in Table", qq1000
FROM MYDBMON a, tablescans b
WHERE qqrid=1000 AND a.qqjfld=b.qqjfld AND
      qqc21 IN ('OP','SI','UP','IN','DL')
GROUP BY a.qqucnt, qq1000
ORDER BY "Total Time" DESC;
```

Then we create an SQL view as shown in Example 6-2.

Example 6-2 Creating an SQL view

```
CREATE VIEW LASSE0410.TABLESCANS AS SELECT QQJFLD, QQUCNT, QQRCOD
FROM MYDBMON WHERE qqrid=3000;
```

The query runs with the new view, with the assumption that the schema is in the library list as shown in Example 6-3.

Example 6-3 Query after using the SQL view

```
SELECT SUM(qqi6) "Total Time", COUNT(*) "Times Run",
  a.qqucnt, integer(avg(b.qqrest)) "Est Rows Selected",
  integer(avg(b.qqtotr)) "Total Rows in Table", qq1000
FROM MYDBMON a, tablescans b
WHERE qqrid=1000 AND a.qqjfld=b.qqjfld AND
      qqc21 IN ('OP','SI','UP','IN','DL')
GROUP BY a.qqucnt, qq1000
ORDER BY "Total Time" DESC;
```

In the remainder of this chapter, we do not use views, but rather table expressions. When you analyze your own Database Monitor data collection, you might find situations where you can benefit from using SQL views.

6.2.4 Creating additional indexes over the Database Monitor table

In Chapter 2, “DB2 Universal Database for iSeries performance basics” on page 9, we cover the importance of indexes for SQL performance. The Database Monitor table tends to become quite large in size. Therefore, it is important to create indexes on the Database Monitor table over the common selection, grouping, and ordering clauses.

The following examples are some key combinations to use:

- QQRID, QQC21
- QQJFLD, QQUCNT, QQI5
- QQRID & QQ1000
- QQRID & QQC21

Figure 6-2 shows the SQL statements to create the indexes described previously.

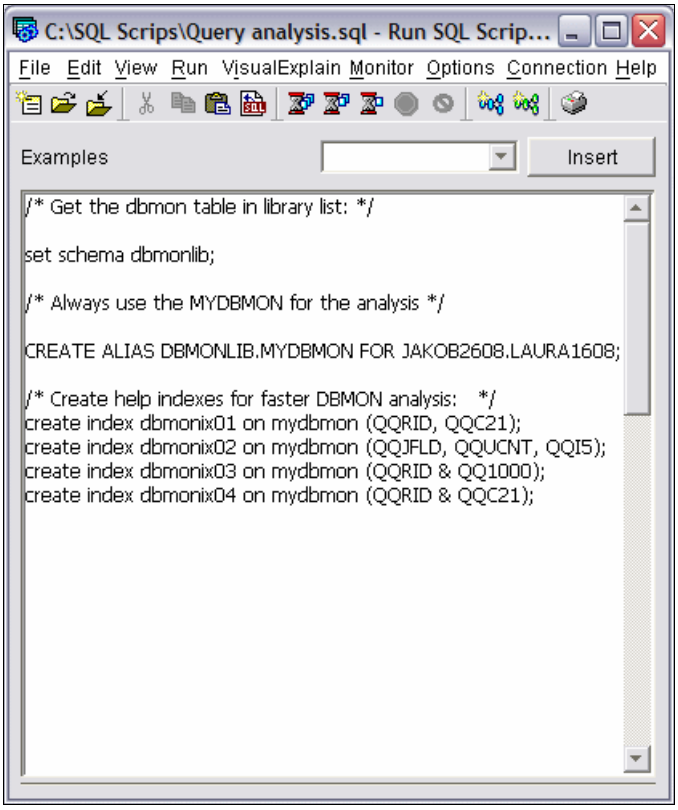


Figure 6-2 Creating additional indexes for faster analysis

You can try other combinations as necessary. Remember to combine the selection, grouping, and ordering clauses.

Note: When you use iSeries Navigator to start a detailed SQL Performance Monitor, as soon as you end the data collection, it provides a couple of indexes, one based on QQJFLD, QQUCNT & QQI5 and another based on QQRID & QQC21.

6.3 Database Monitor query examples

In this section, we present a series of queries to help solve specific questions in the detection and resolution of SQL performance issues. Most of these queries have more elaborate equivalents in the SQL Performance Monitor predefined reports as indicated. However, it is still useful to be familiar with the Database Monitor table.

Before running the queries, we presume that an SQL SET SCHEMA and an SQL CREATE ALIAS are done as shown in Figure 6-2.

6.3.1 Finding SQL requests that are causing problems

To find the SQL that is causing problems, expect to use more queries for the investigation, because it is not only the running time that matters.

First take an overview over the data collected. To determine the number of SQL request that were done during the collection, run the query shown in Example 6-4.

Example 6-4 Number of SQL requests in the Database Monitor table

```
SELECT count(*) FROM mydbmon
WHERE qqrid=1000 AND qqc21 <> 'MT';
```

Figure 6-3 shows the result of the query in Example 6-4.

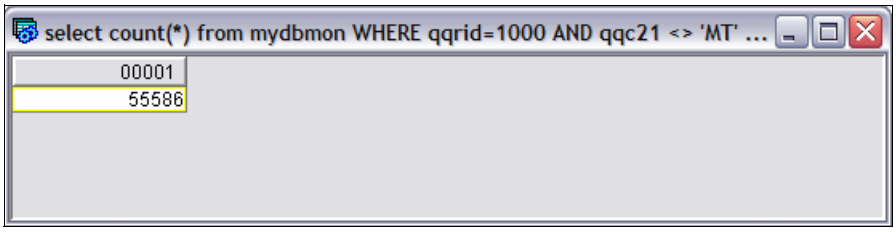


Figure 6-3 Number of SQL requests in the DBMON collection

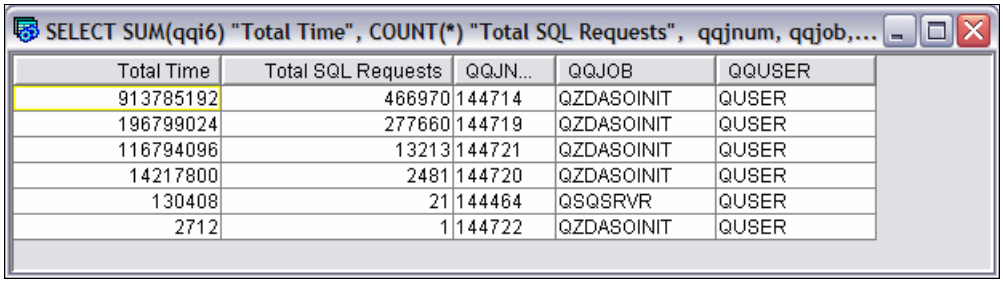
The result is smaller than the number of rows in the Database Monitor table. The number of rows per SQL request can be from 1 to over 20, depending on the complexity of the SQL request. Each SQL request has one QQRID = 1000 record. Sometimes the QQC21 row contains the value MT for more text about the same SQL request, so it should only be counted once.

For an overview of the most time consuming jobs running SQL, we use the query shown in Example 6-5.

Example 6-5 Time consuming jobs

```
SELECT SUM(qqi6) "Total Time", COUNT(*) "Total SQL Requests",
qqjnum, qqjob, qquser FROM mydbmon
WHERE qqrid=1000 AND qqc21 <> 'MT'
GROUP BY qqjob,qquser,qqjnum ORDER BY 1 DESC;
```

Figure 6-4 shows the output of the query in Example 6-5.



Total Time	Total SQL Requests	QQJN...	QQJOB	QQUSER
913785192	466970	144714	QZDASOINIT	QUSER
196799024	277660	144719	QZDASOINIT	QUSER
116794096	13213	144721	QZDASOINIT	QUSER
14217800	2481	144720	QZDASOINIT	QUSER
130408	21	144464	QSQSRVR	QUSER
2712	1	144722	QZDASOINIT	QUSER

Figure 6-4 Time consuming jobs

The total time is in microseconds. Therefore, you must divide the time by 1000000 to see it in seconds.

To find SQL requests that might cause the problems, look at different information. For example, you should know whether the Database Monitor collection is from a sample period of the day or from a period when response problems were observed. You can also determine which SQL requests in a specific job or jobs take the most processing time.

6.3.2 Total time spent in SQL

During the analysis of monitor data, you can see the percentage of time that is spent in DB2. To begin, you find the start time and end time to have a duration of the Database Monitor data collection as shown in Example 6-6.

Example 6-6 Duration of the Database Monitor data collection

```
SELECT MIN(qqtime) "Start Time", MAX(qqtime) "End Time",
MAX((DAY(qqtime)*24*3600)+(HOUR(qqtime)*3600)+(MINUTE(qqtime)*60)+
(SECOND(qqtime))+(MICROSECOND(qqtime)*.000001)) -
MIN((DAY(qqtime)*24*3600)+(HOUR(qqtime)*3600)+(MINUTE(qqtime)*60)+
(SECOND(qqtime))+(MICROSECOND(qqtime)*.000001)) AS "Duration"
FROM mydbmon WHERE qqrid<>3018
```

Figure 6-5 shows the result of Example 6-6.

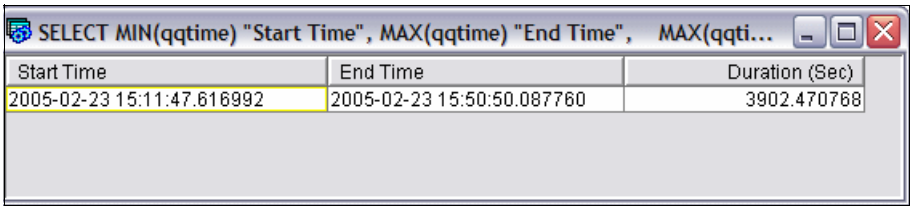


Figure 6-5 Duration of Database Monitor data collection

To get the duration of the Database Monitor data collection for the job, select the job number as shown in Example 6-7. For `qqjnum = '999999'`, substitute the job number.

Example 6-7 Duration of the Database Monitor collection for one job

```
SELECT MIN(qqtime) "Start Time", MAX(qqtime) "End Time",
MAX(qqtime) - MIN(qqtime) "Duration (Sec)"
FROM mydbmon WHERE qqrid<>3018 and qqjnum = '999999';
```

You can also find the total number of seconds spent by using the SQL statement shown in Example 6-8. The QQC21 has the value `'MT'` when a More Text record exists.

Example 6-8 Time spent in SQL

```
SELECT SUM(qqi6)/1000000 "Total Time (Sec)" FROM mydbmon
WHERE qqrid=1000 AND qqc21 <> 'MT';
```

If stored procedures are used, then they count as double, because both the stored procedure and the SQL requests in the stored procedure generate records in the Database Monitor table. A good approximation is to exclude the stored procedure from the total time used in the SQL shown Example 6-9.

Example 6-9 Time spent in SQL excluding stored procedures

```
SELECT SUM(qqi6)/1000000 "Total Time (Sec)" FROM mydbmon
WHERE qqrid=1000 AND qqc21 <> 'MT' AND qqc21 <> 'CA';
```

Figure 6-6 shows the output for Example 6-9.

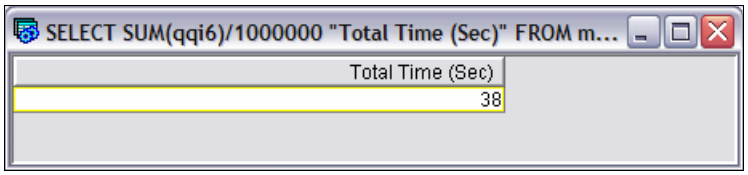


Figure 6-6 Time spent in SQL

The time spent in SQL might not seem so relevant for the total run, but when the focus is on individual jobs, or applications, then it is relevant. This means that a selection of the job should be added to the query.

6.3.3 Individual SQL elapsed time

To find the SQL requests that count for the most run time, use the query shown in Example 6-10.

Example 6-10 SQL request sorted on the total run time

```
SELECT SUM(qqi6) "Total Time" , COUNT(*) "Nbr Times Run", qq1000 "SQL Request"
FROM MYDBMON
WHERE qqrid=1000 AND qqucnt<>0
AND qqc21<>'MT'
GROUP BY qq1000 ORDER BY 1 DESC;
```

The result shown in Figure 6-7 comes from our test collection.

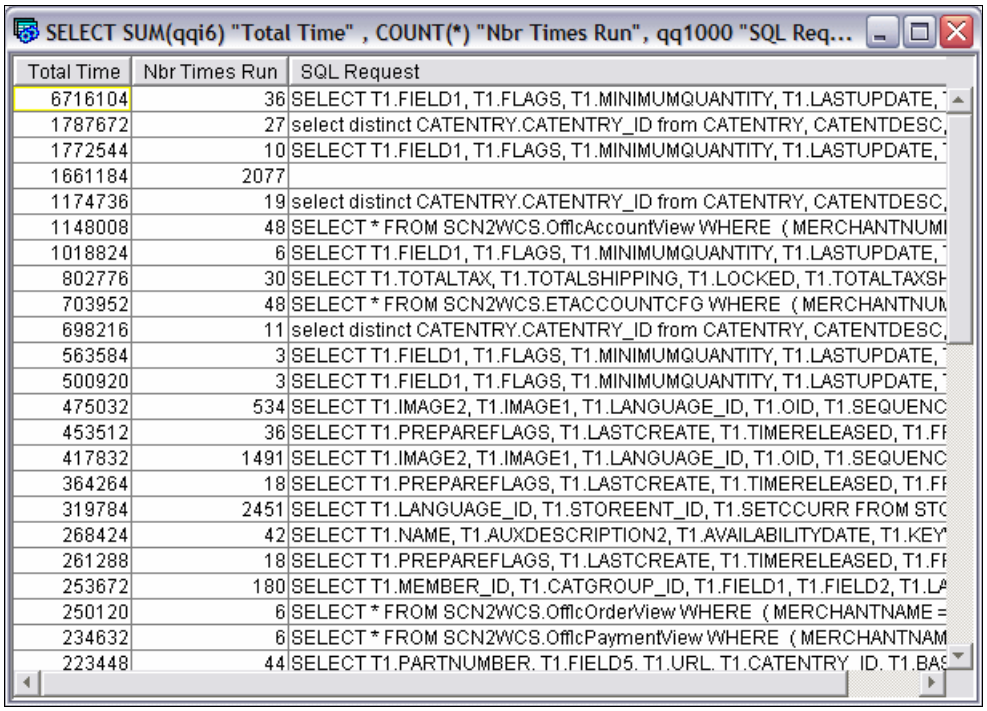


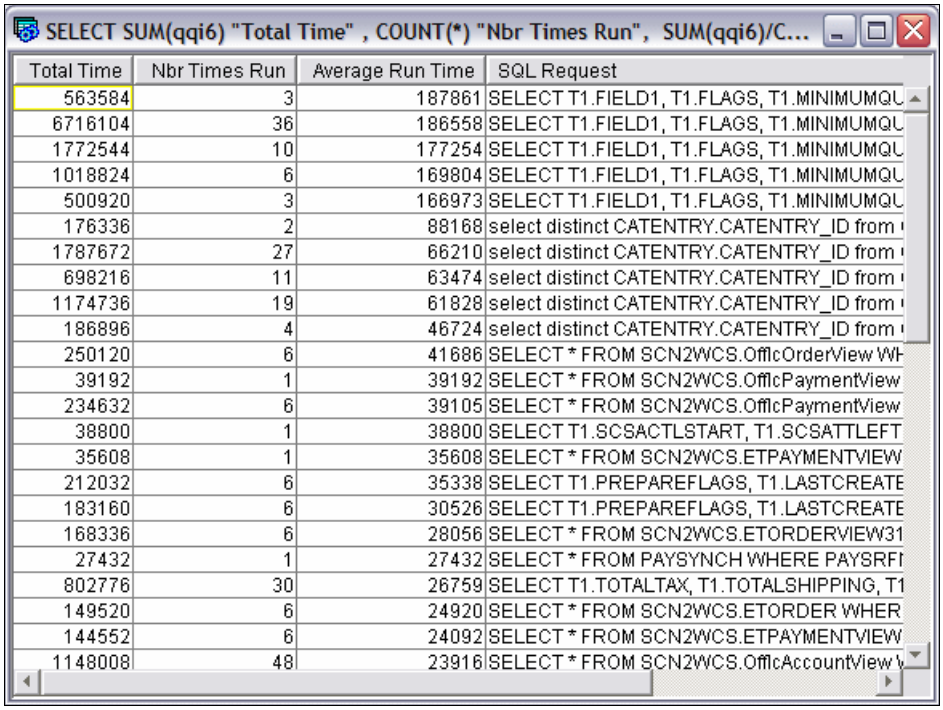
Figure 6-7 SQL requests sorted by Total Time

A large number of the same SQL request can give a high total run time. Therefore, it is also relevant to look at the SQL requests with the longest average run time as shown in the query in Example 6-11.

Example 6-11 SQL requests sorted by average run time

```
SELECT SUM(qqi6) "Total Time" , COUNT(*) "Nbr Times Run",
SUM(qqi6)/COUNT(*) "Average Run Time", qq1000 "SQL Request"
FROM MYDBMON
WHERE qqrid=1000 AND qqucnt<>0 AND qqc21<>'MT'
GROUP BY qq1000 ORDER BY 3 DESC;
```

The execution of the query in Example 6-11 produces the result shown in Figure 6-8.


SELECT SUM(qqi6) "Total Time" , COUNT(*) "Nbr Times Run", SUM(qqi6)/C...

Total Time	Nbr Times Run	Average Run Time	SQL Request
563584	3	187861	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQU
6716104	36	186558	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQU
1772544	10	177254	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQU
1018824	6	169804	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQU
500920	3	166973	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQU
176336	2	88168	select distinct CATENTRY.CATENTRY_ID from
1787672	27	66210	select distinct CATENTRY.CATENTRY_ID from
698216	11	63474	select distinct CATENTRY.CATENTRY_ID from
1174736	19	61828	select distinct CATENTRY.CATENTRY_ID from
186896	4	46724	select distinct CATENTRY.CATENTRY_ID from
250120	6	41686	SELECT * FROM SCN2WCS.OfflcOrderView WH
39192	1	39192	SELECT * FROM SCN2WCS.OfflcPaymentView
234632	6	39105	SELECT * FROM SCN2WCS.OfflcPaymentView
38800	1	38800	SELECT T1.SCSACTLSTART, T1.SCSATTLEFT
35608	1	35608	SELECT * FROM SCN2WCS.ETPAYMENTVIEW
212032	6	35338	SELECT T1.PREPAREFLAGS, T1.LASTCREATE
183160	6	30526	SELECT T1.PREPAREFLAGS, T1.LASTCREATE
168336	6	28056	SELECT * FROM SCN2WCS.ETORDERVIEW31
27432	1	27432	SELECT * FROM PAYSYNCH WHERE PAYSRFI
802776	30	26759	SELECT T1.TOTALTAX, T1.TOTALSHIPPING, T1
149520	6	24920	SELECT * FROM SCN2WCS.ETORDER WHER
144552	6	24092	SELECT * FROM SCN2WCS.ETPAYMENTVIEW
1148008	48	23916	SELECT * FROM SCN2WCS.OfflcAccountView

Figure 6-8 SQL requests sorted on Average Run Time

Important: Notice that the queries in Example 6-10 and Example 6-11 use GROUP BY in the QQ1000 field whose length is 1000. If the monitor table contains SQL statements that exceed this length and the first 1000 characters are identical, the result of the SUM and COUNT functions will not be accurate.

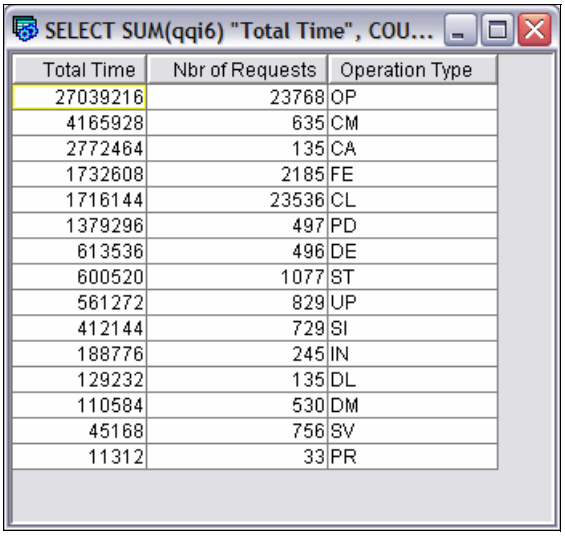
6.3.4 Analyzing SQL operation types

To get an overview of the different SQL operation types that are run during the performance data collection, use the query shown in Example 6-12.

Example 6-12 SQL operation types

```
SELECT SUM(qqi6) "Total Time", COUNT(*) "Nbr of Requests",
qqc21 "Operation Type" FROM MYDBMON
WHERE qqrid=1000 AND qqc21 <> 'MT'
GROUP BY qqc21 ORDER BY 1 DESC;
```

Figure 6-9 shows the output of the query in Example 6-12.



Total Time	Nbr of Requests	Operation Type
27039216	23768	OP
4165928	635	CM
2772464	135	CA
1732608	2185	FE
1716144	23536	CL
1379296	497	PD
613536	496	DE
600520	1077	ST
561272	829	UP
412144	729	SI
188776	245	IN
129232	135	DL
110584	530	DM
45168	756	SV
11312	33	PR

Figure 6-9 SQL operation types

This example gives you an idea of the SQL operations that run the most and the amount of time that they account for. In our example, there were 23768 OPEN operations.

6.3.5 Full open analysis

The first time (or times) that an open occurs for a specific statement in a job, a full open operation is required. A *full open* creates an open data path (ODP) that is then used to fetch, update, delete, or insert rows. An ODP might be cached at close time, so that if the SQL statement is run again during the job, the ODP is reused. Such an open is called a *pseudo open* and is much less expensive than a full open.

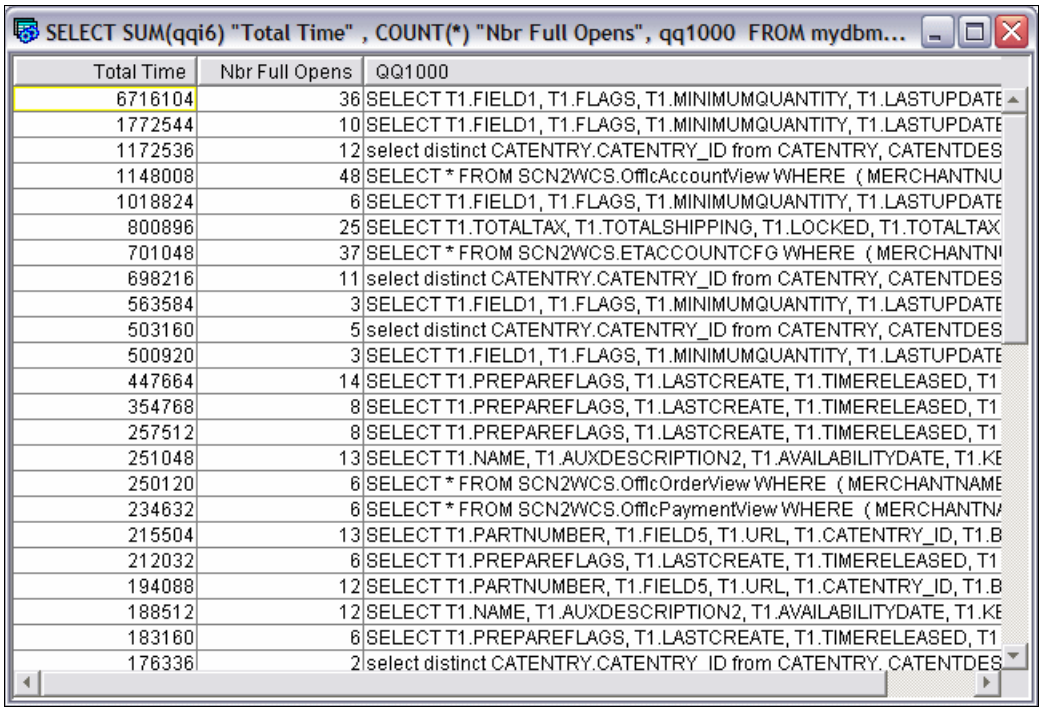
A normal SQL application has many fetches, inserts, updates, and deletes. A desirable situation is that most of the operations share the ODP so that a full open does not have to be done over and over again.

To find the number of SQL requests affected by full opens, you use the query shown in Example 6-13.

Example 6-13 SQL requests affected by Full Opens

```
SELECT SUM(qqi6) "Total Time" , COUNT(*) "Nbr Full Opens", qq1000
FROM mydbmon
WHERE qqrid=1000 AND qqi5=0
AND qqc21 IN ('OP','SI', 'DL', 'IN', 'UP')
GROUP BY qq1000 ORDER BY 1 DESC;
```

Figure 6-10 shows the results of the query in Example 6-13.


SELECT SUM(qqi6) "Total Time" , COUNT(*) "Nbr Full Opens", qq1000 FROM mydbm...

Total Time	Nbr Full Opens	QQ1000
6716104	36	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE
1772544	10	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE
1172536	12	select distinct CATENTRY.CATENTRY_ID from CATENTRY, CATENTDES
1148008	48	SELECT * FROM SCN2WCS.OfflcAccountView WHERE (MERCHANTNU
1018824	6	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE
800896	25	SELECT T1.TOTALTAX, T1.TOTALSHIPPING, T1.LOCKED, T1.TOTALTAX
701048	37	SELECT * FROM SCN2WCS.ETACCOUNTCFG WHERE (MERCHANTN
698216	11	select distinct CATENTRY.CATENTRY_ID from CATENTRY, CATENTDES
563584	3	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE
503160	5	select distinct CATENTRY.CATENTRY_ID from CATENTRY, CATENTDES
500920	3	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE
447664	14	SELECT T1.PREPAREFLAGS, T1.LASTCREATE, T1.TIMERELEASED, T1
354768	8	SELECT T1.PREPAREFLAGS, T1.LASTCREATE, T1.TIMERELEASED, T1
257512	8	SELECT T1.PREPAREFLAGS, T1.LASTCREATE, T1.TIMERELEASED, T1
251048	13	SELECT T1.NAME, T1.AUXDESCRIPTION2, T1.AVAILABILITYDATE, T1.KE
250120	6	SELECT * FROM SCN2WCS.OfflcOrderView WHERE (MERCHANTNAME
234632	6	SELECT * FROM SCN2WCS.OfflcPaymentView WHERE (MERCHANTNA
215504	13	SELECT T1.PARTNUMBER, T1.FIELD5, T1.URL, T1.CATENTRY_ID, T1.B
212032	6	SELECT T1.PREPAREFLAGS, T1.LASTCREATE, T1.TIMERELEASED, T1
194088	12	SELECT T1.PARTNUMBER, T1.FIELD5, T1.URL, T1.CATENTRY_ID, T1.B
188512	12	SELECT T1.NAME, T1.AUXDESCRIPTION2, T1.AVAILABILITYDATE, T1.KE
183160	6	SELECT T1.PREPAREFLAGS, T1.LASTCREATE, T1.TIMERELEASED, T1
176336	2	select distinct CATENTRY.CATENTRY_ID from CATENTRY, CATENTDES

Figure 6-10 SQL requests affected by Full Opens

To analyze the full opens, you must next copy and paste the SQL request or part of the SQL request into a query that we use for analysis. From the previous example, copy the following SQL statement:

“SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE, T1.ENDDATE,
T1.TRADEPOSCN_ID, T1.QTYUNIT_ID, T1.STARTDATE, T1.OID, T1.MAXIMUMQUANTITY, T1.....”

Copy the SQL statement into the query shown in Example 6-14. Be sure to substitute the Xs that follow the LIKE predicate.

Example 6-14 SQL requests affected by Full Opens in a single job

```
SELECT qqc181 "Cursor Name",qqi6 "Exec Time", qqucnt, qqi5, qqc21,
qqc15 "HC Reason", qq1000, qqc22 "Rebuild Reason Code",
qqc182 "Stmt Name"
FROM mydbmon
WHERE qqjnum='999999' AND
(qqrid = 1000 and qqc21 in ('OP') AND
qq1000 LIKE 'XXXXXXXXXXXXXXXXXXXXXXXXXXXX%')
OR (qqc21 IN ('HC','DI', 'ST', 'CM', 'RO') )
ORDER BY qqtime;
```

Figure 6-11 shows the results for Example 6-14.

SELECT qqc181 "Cursor Name",qqi6 "Exec Time", qqucnt, qqi5, qqc21, ...

Cursor Name	Exec Time	QQUCNT	QQI5	QQC21	HC Reason	QQ1000
	5384	0	0	CM		COMMIT
	744	0	0	ST		SET TRANSACTION
	53880	0	0	CM		COMMIT
	688	0	0	ST		SET TRANSACTION
	53576	0	0	CM		COMMIT
	616	0	0	ST		SET TRANSACTION
	2664	0	0	CM		COMMIT
	1312	0	0	ST		SET TRANSACTION
	54736	0	0	CM		COMMIT
	752	0	0	ST		SET TRANSACTION
	776	0	0	ST		SET TRANSACTION
	328	0	0	ST		SET TRANSACTION
	2344	0	0	CM		COMMIT
	584	0	0	ST		SET TRANSACTION
	384	0	0	ST		SET TRANSACTION
	288	0	0	ST		SET TRANSACTION
	1184	0	0	CM		COMMIT

Figure 6-11 SQL requests affected by Full Opens in a single job

Look at the values in the QQC21 column for ODP analysis, which shows the following abbreviations:

HC Hard Close
DI Disconnect
ST Set Transaction
CM Commit
RO Rollback

Identify the reasons why a Hard Close is being done by looking in the QQC15 column, which has these reason codes:

1 Internal Error
2 Exclusive Lock
3 Interactive SQL Reuse Restriction
4 Host Variable Reuse Restriction
5 Temporary Result Restriction
6 Cursor Restriction (after first execution)
7 Cursor Hard Close Requested (proprietary attribute)
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 Dummy Cursor Threshold
J File Override Change
K Program Invocation Change
L File Open Options Change
M Stmt Reuse Restriction

N Internal Error
O Library List Change
P Exit Processing (End Connection)

Example 6-15 shows another query to find the number of Full Opens and Pseudo Opens for the SQL request.

Example 6-15 Number of Full Opens and Pseudo Opens

```
SELECT SUM(qqi6) "Total Time" ,
SUM(CASE WHEN  qvc12 = 'N' THEN 1 ELSE 0 END) "Full Opens",
SUM(CASE WHEN  qvc12 = 'Y' THEN 1 ELSE 0 END) "Pseudo Opens",
QQ1000
FROM dbmon_file
WHERE qqrid=1000 AND qqc21 <> 'MT'
AND qqc21 IN ('OP','SI', 'DL', 'IN', 'UP')
GROUP BY qq1000
ORDER BY 1 DESC;
```

Figure 6-12 shows the results of the query in Example 6-15.

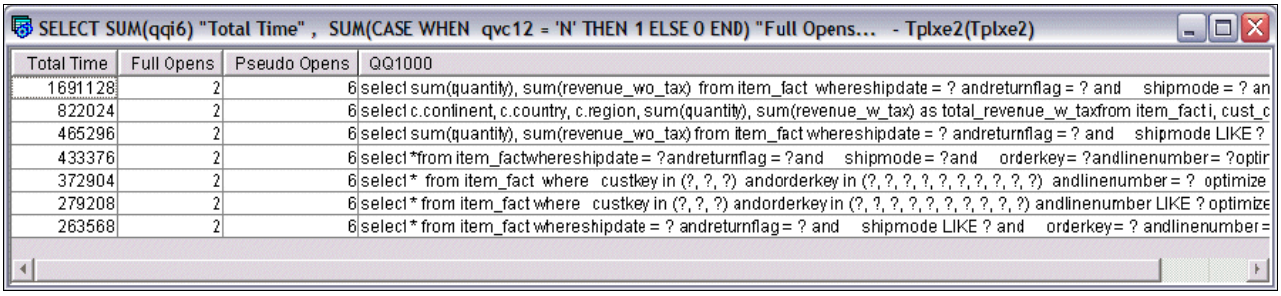


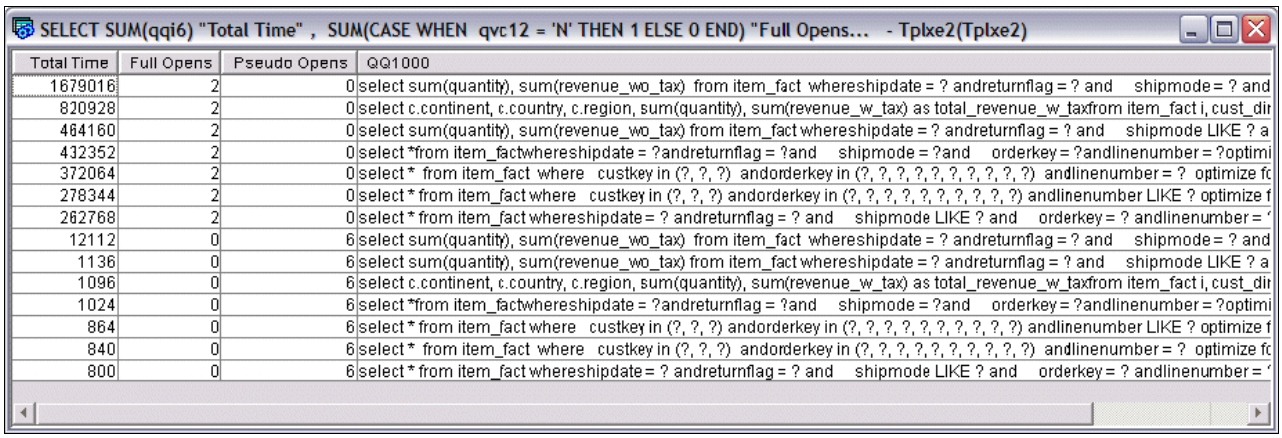
Figure 6-12 Number of Full Opens and Pseudo Opens

The total time in Figure 6-12 includes the Full Opens and Pseudo Opens for each SQL request. To look at the time for Full Opens and Pseudo Opens separately, you must add QVC12 to the GROUP BY clause from the previous query, as shown in Example 6-16.

Example 6-16 Number of Full Opens and Pseudo Opens looked separately

```
SELECT SUM(qqi6) "Total Time" ,
SUM(CASE WHEN  qvc12 = 'N' THEN 1 ELSE 0 END) "Full Opens",
SUM(CASE WHEN  qvc12 = 'Y' THEN 1 ELSE 0 END) "Pseudo Opens",
QQ1000
FROM dbmon_file
WHERE qqrid=1000 AND qqc21 <> 'MT'
AND qqc21 IN ('OP','SI', 'DL', 'IN', 'UP')
GROUP BY qvc12,qq1000
ORDER BY 1 DESC;
```

Figure 6-13 shows the results of the query in Example 6-16.


SELECT SUM(qqi6) "Total Time" , SUM(CASE WHEN qvc12 = 'N' THEN 1 ELSE 0 END) "Full Opens... - Tplxe2(Tplxe2)

Total Time	Full Opens	Pseudo Opens	QQ1000
1679016	2	0	select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? andreturnflag = ? and shipmode = ? and
820928	2	0	select c.continent, c.country, c.region, sum(quantity), sum(revenue_w_tax) as total_revenue_w_taxfrom item_fact i, cust_dir
464160	2	0	select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? andreturnflag = ? and shipmode LIKE ? a
432352	2	0	select *from item_factwhereshipdate = ?andreturnflag = ?and shipmode = ?and orderkey = ?andlinenumber = ?optimi
372064	2	0	select * from item_fact where custkey in (?, ?, ?) andorderkey in (?, ?, ?, ?, ?, ?, ?, ?, ?) andlinenumber = ? optimize fc
278344	2	0	select * from item_fact where custkey in (?, ?, ?) andorderkey in (?, ?, ?, ?, ?, ?, ?, ?, ?) andlinenumber LIKE ? optimize f
262768	2	0	select * from item_fact whereshipdate = ? andreturnflag = ? and shipmode LIKE ? and orderkey = ? andlinenumber = '
12112	0	6	select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? andreturnflag = ? and shipmode = ? and
1136	0	6	select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? andreturnflag = ? and shipmode LIKE ? a
1096	0	6	select c.continent, c.country, c.region, sum(quantity), sum(revenue_w_tax) as total_revenue_w_taxfrom item_fact i, cust_dir
1024	0	6	select *from item_factwhereshipdate = ?andreturnflag = ?and shipmode = ?and orderkey = ?andlinenumber = ?optimi
864	0	6	select * from item_fact where custkey in (?, ?, ?) andorderkey in (?, ?, ?, ?, ?, ?, ?, ?, ?) andlinenumber LIKE ? optimize f
840	0	6	select * from item_fact where custkey in (?, ?, ?) andorderkey in (?, ?, ?, ?, ?, ?, ?, ?, ?) andlinenumber = ? optimize fc
800	0	6	select * from item_fact whereshipdate = ? andreturnflag = ? and shipmode LIKE ? and orderkey = ? andlinenumber = '

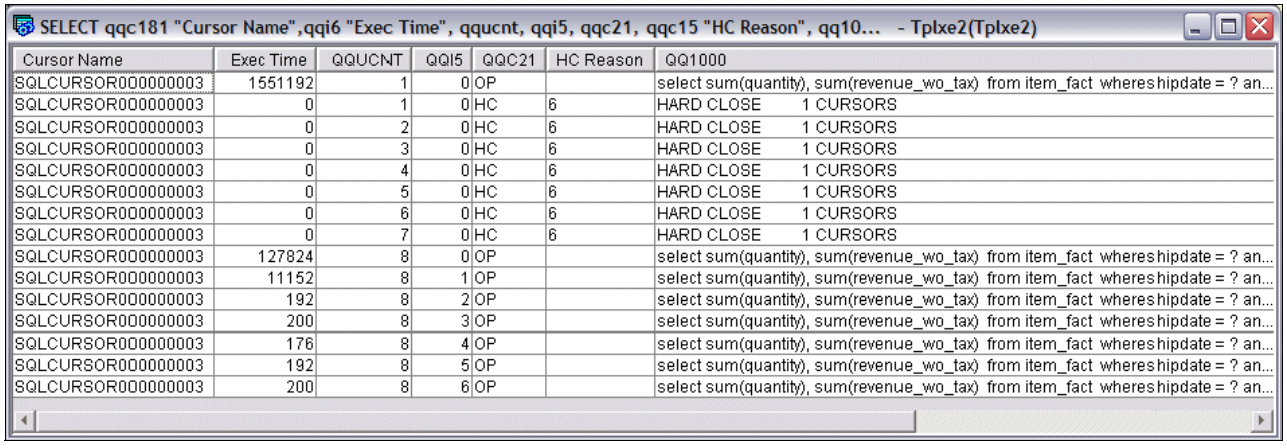
Figure 6-13 Full Opens and Pseudo Opens shown separately

To analyze the Full Opens, copy and paste the SQL request or part of the SQL request into a query that used for analysis. From the previous query and result, we analyze the first SQL request which is the most expensive. The first SQL request is similar to the third request. Therefore, we use the LIKE predicate with two wildcards to ensure that we only retrieve information for the first SQL request. Example 6-17 shows the query.

Example 6-17 Looking at the number of Full Opens and Pseudo Opens separately

```
SELECT qqc181 "Cursor Name",qqi6 "Exec Time", qqucnt, qqi5, qqc21,
qqc15 "HC Reason", qq1000, qqc22 "Rebuild Reason Code",
qqc182 "Stmt Name"
FROM dbmon_file
WHERE qqjnum='459263' AND
(qqrid = 1000 and qqc21 in ('OP') AND
UCASE(qq1000) LIKE 'SELECT SUM(QUANTITY)%SHIPMODE =%')
OR (qqc21 IN ('HC','DI', 'ST', 'CM', 'RO') )
ORDER BY qqtime;
```

Figure 6-14 shows the result of the query in Example 6-17.


SELECT qqc181 "Cursor Name",qqi6 "Exec Time", qqucnt, qqi5, qqc21, qqc15 "HC Reason", qq10... - Tplxe2(Tplxe2)

Cursor Name	Exec Time	QQUCNT	QQI5	QQC21	HC Reason	QQ1000
SQLCURSOR000000003	1551192	1	0	OP		select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? an...
SQLCURSOR000000003	0	1	0	HC	6	HARD CLOSE 1 CURSORS
SQLCURSOR000000003	0	2	0	HC	6	HARD CLOSE 1 CURSORS
SQLCURSOR000000003	0	3	0	HC	6	HARD CLOSE 1 CURSORS
SQLCURSOR000000003	0	4	0	HC	6	HARD CLOSE 1 CURSORS
SQLCURSOR000000003	0	5	0	HC	6	HARD CLOSE 1 CURSORS
SQLCURSOR000000003	0	6	0	HC	6	HARD CLOSE 1 CURSORS
SQLCURSOR000000003	0	7	0	HC	6	HARD CLOSE 1 CURSORS
SQLCURSOR000000003	127824	8	0	OP		select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? an...
SQLCURSOR000000003	11152	8	1	OP		select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? an...
SQLCURSOR000000003	192	8	2	OP		select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? an...
SQLCURSOR000000003	200	8	3	OP		select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? an...
SQLCURSOR000000003	176	8	4	OP		select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? an...
SQLCURSOR000000003	192	8	5	OP		select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? an...
SQLCURSOR000000003	200	8	6	OP		select sum(quantity), sum(revenue_wo_tax) from item_fact whereshipdate = ? an...

Figure 6-14 Analysis of the Full Opens

From the previous example, we can see that a Full Open took place the first time. The indication that it was a Full Open is the code in QQC21 and that QQI5 is 0. The Full Open

took 1.5 seconds. The query did not go into reusable mode and the cursor was hard closed because of a cursor restriction (reason 6 in QQC15). The second Full Open took place again, but this time a hard close didn't occur leaving the ODP to be reused. Subsequent executions reused the ODP. We can see this by looking at the QQUCNT and QQI5 fields. The number 8 in QQUCNT was assigned during the second Full Open and stayed constant for all subsequent instances of that query. QQI5 has the number assigned to each instance of the query. Notice that the execution time is minimum when the query entered into reusable mode.

For the complete list of statement types (QQC21) and the list of hard close reasons (QQC15), search on `Database Monitor: DDS` in the V5R3 iSeries Information Center.

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp

6.3.6 Reusable ODPs

Reusable ODP usually happens after the second execution of an SQL statement within the connection or job, if the statement is reusable. Because the reusable statements are significantly faster than the nonreusable ones, you can find the statements that are not reusing the ODP. The QQUCNT value is assigned at full open time and stays constant for all subsequent reusable instances of that particular query.

Nonreusable ODPs are indicated by the presence of optimization records each time a particular query is run (full open). Reusable ODPs are indicated by 3010 and 1000 records each time the given query is run (no optimization records or full open). To understand why an ODP is not reusable, look at the hard close reason.

6.3.7 Isolation level used

You can see the number of statements that were run under each isolation level. This information provides you with a high level indication of the isolation level used. The higher the isolation level is, the higher the chance of contention is between users, which are seen as job locks and seizes. A high level of Repeatable Read or Read Stability use is likely to produce a high level of contention. Always use the lowest level isolation level that still satisfies the application design requirement as indicated in the query shown in Example 6-18.

Example 6-18 Isolation level summary

```
select qvc41, count(qvc41) from mydbmon
WHERE  QQRID = 1000 AND QQC21 <> 'MT'
GROUP BY qvc41 order by 2 desc;
```

Figure 6-15 shows the results of the query in Example 6-18.

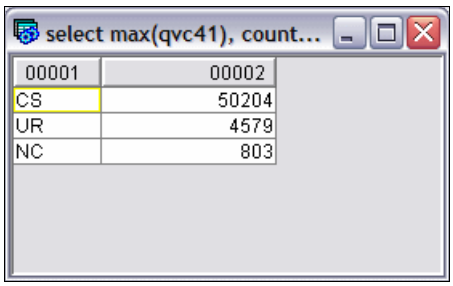


Figure 6-15 Isolation level summary

The values can be translated using the following codes:

RR Repeatable Read. In the SQL 1999 Core standard, Repeatable Read is called *serializable*.

RS Read Stability. In the SQL 1999 Core standard, Read Stability is called *Repeatable Read*.

CS Cursor Stability. In the SQL 1999 Core standard, Cursor Stability is called *Read Committed*.

CSKL Cursor Stability KEEP LOCKS.

UR Uncommitted Read. In the SQL 1999 Core standard, Uncommitted Read is called *Read Uncommitted*.

NC No Commit.

As you can see from the previous example, most reads are done using Cursor Stability. This means that the isolation level is low, and therefore, the possibility for contention is low.

6.3.8 Table scan

A table scan operation is an efficient way to process all the rows in the table and verify that they satisfy the selection criteria specified in the query. Its efficiency is accomplished by bringing necessary data into main memory via a large I/O request and asynchronous prefetches.

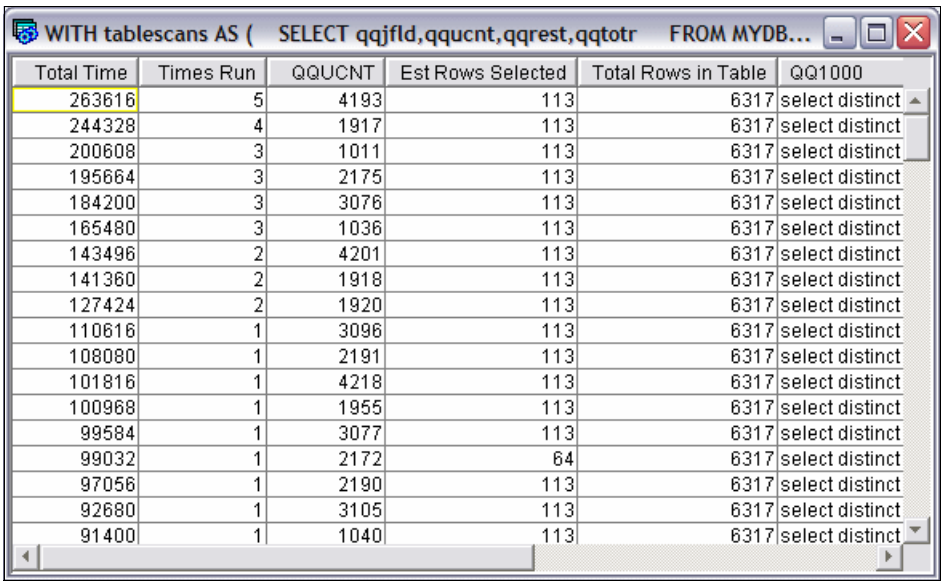
The table scan is generally acceptable in cases where a large portion of the table is selected or the selected table contains a small number of records. To address cases where the entire table is scanned, but a relatively small number of rows is selected, building an index on the selection criteria is the best alternative and fully supported by the Database Monitor data.

Assuming that you have collected Detailed Database Monitor data, you can use the query shown in the Example 6-19 to see the statements that have resulted in table scan operations.

Example 6-19 Table scan operations

```
WITH tablescans AS (
  SELECT qqjfld,qqucnt,qqrest,qqtotr
  FROM MYDBMON
  WHERE qqrid=3000)
SELECT SUM(qqi6) "Total Time", COUNT(*) "Times Run",
  a.qqucnt, integer(avg(b.qqrest)) "Est Rows Selected",
  integer(avg(b.qqtotr)) "Total Rows in Table", qq1000
FROM MYDBMON a, tablescans b
WHERE qqrid=1000 AND a.qqjfld=b.qqjfld AND
      qqc21 IN ('OP','SI','UP','IN','DL')
GROUP BY a.qqucnt, qq1000
ORDER BY "Total Time" DESC;
```

Figure 6-16 shows the results of the query in Example 6-19.


WITH tablescans AS (SELECT qqjfld,qqucnt,qqrest,qqtotr FROM MYDB...

Total Time	Times Run	QQUCNT	Est Rows Selected	Total Rows in Table	QQ1000
263616	5	4193	113	6317	select distinct
244328	4	1917	113	6317	select distinct
200608	3	1011	113	6317	select distinct
195664	3	2175	113	6317	select distinct
184200	3	3076	113	6317	select distinct
165480	3	1036	113	6317	select distinct
143496	2	4201	113	6317	select distinct
141360	2	1918	113	6317	select distinct
127424	2	1920	113	6317	select distinct
110616	1	3096	113	6317	select distinct
108080	1	2191	113	6317	select distinct
101816	1	4218	113	6317	select distinct
100968	1	1955	113	6317	select distinct
99584	1	3077	113	6317	select distinct
99032	1	2172	64	6317	select distinct
97056	1	2190	113	6317	select distinct
92680	1	3105	113	6317	select distinct
91400	1	1040	113	6317	select distinct

Figure 6-16 Table scan operations

In the previous query, notice the following columns:

- QQJFLD and QQUCNT

  These columns are join fields that are required to uniquely identify an SQL statement in the Database Monitor file.

- QQC21

  Since we are joining common table expression table scans back to the 1000 record in the Database Monitor file, we must ensure that we only join to 1000 records that can cause table scans to occur. We accomplish this by verifying that the QQC21 field operation is either open, select, update, delete, or insert. We also include the last three operations because they might have subselects or correlated subqueries.

- QQREST and QQTOTR

  We included the QQREST (estimated rows selected) and QQTOTR (total rows in table) columns to give you an idea of the selectivity of the statement. A great difference between these two columns is a good indicator that index is a better alternative to a table scan operation.

- QQRID

  A table scan operation is uniquely identified by the record ID (QQRID) value of 3000. We include it as selection criteria in the common table expression `'tablescans'`.

- QQI6

  This column indicates a table scan operation with a cumulative elapse time in microseconds for each individual query. Since we use it as a cost indicator, we ordered the output in descending order based on this value.

Note: Focus your efforts on optimizing statements that are displayed at the top. Ignore statements whose total time was inflated due to numerous executions (times run field). For other statements, consider the total number of rows in the table before taking further action. If this number is relatively small, your optimization efforts are best spent elsewhere.

The query that we have outlined so far is insufficient in helping us to decide if we should build an index. Using the query shown in Example 6-19, we include data that is necessary to make that decision as shown by Example 6-20.

Example 6-20 Keys advised

```
WITH tablescans AS (
SELECT qqjfld,qqucnt,qqrest,qqtotr FROM MYDBMON WHERE qqrid = 3000),
details AS ( SELECT qqjfld, qqucnt, qqi7 FROM MYDBMON WHERE qqrid = 3019),
summation AS ( SELECT  a.qqucnt, a.qqjfld, SUM(qqi6) "Total Time", COUNT(*) "Times Run",
integer(avg(b.qqrest)) "Est Rows Selected",
integer(avg(b.qqtotr)) "Total Rows in Table",
integer(avg(c.qqi7)) as "Rows Returned", qq1000
FROM MYDBMON a, tablescans b, details c
WHERE qqrid=1000 AND a.qqjfld=b.qqjfld AND a.qqjfld = c.qqjfld
AND qqc21 IN ('OP','SI','UP','IN','DL')
GROUP BY a.qqjfld, a.qqucnt, qq1000)
SELECT   a.qqtfn "File", a.qqtln "Library", a.qqi2 "Nbr of Primary Keys",
         a.qqidxd "Keys Advised", a.qqrcod "Reason Code", b."Total Time",
         b."Times Run", b."Est Rows Selected", b. "Rows Returned",
         b."Total Rows in Table", b.qq1000
FROM     MYDBMON a, summation b WHERE a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt AND
         a.qqidxa = 'Y' AND a.qqrid = 3000 ORDER BY b."Total Time" DESC;
```

This query includes the following columns among others:

- QQI7

  We included details for a common table expression to illustrate usage of the 3019 row type - Detailed Row information. This row is written when *DETAILED is specified on Start Database Monitor (STRDBMON) command or if using the Detailed option in iSeries Navigator.

  Note: Compare Rows Returned with the Total Rows in Table to gain an idea of the selectivity for the query that caused the table scan operation.

  You should also compare Estimated Rows Selected with Rows Returned. Consistently great differences between these two columns are indicative that the query optimizer needs better statistics (through additional statistics or indexes). The difference does not indicate poor performance.

- QQTFN and QQTLN

  These two columns refer to the base table and schema over which a table scan operation was performed and for which we are considering building an index.

- QQIDXA

  This column refers to the flag that specifies if an index was advised as indicated by Y or N. We use it to filter out table scans for which no index was recommended by the query optimizer (QQIDXA = 'Y').

- QQIDXD

  This column indicates which index was advised. It lists the primary keys first, followed by zero or more secondary keys. Secondary selection keys are less likely to have a significant positive impact on the query's performance.

- QQI2

  This column indicates the number of primary keys contained in the QQIDXD column. You can build an index over primary key fields for the most benefit.

- QQRCOD

  This column indicates the reason code for a table scan operation. For individual reason code descriptions, search on `Database Monitor: DDS` in the V5R3 iSeries Information Center.

  `http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp`

Note: Build indexes to reduce the cost of table scan operations where high selectivity dictates usage of an index.

Always keep in mind that database performance optimization is an iterative process. Therefore, after each action you take, recollect the Database Monitor data and re-analyze it to validate that your action has resulted in better performance.

6.3.9 Temporary index analysis

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. The temporary index can be used to satisfy a variety of query requests, but it is only considered by the Classic Query Engine (CQE) when the query contains ordering, grouping, or joins.

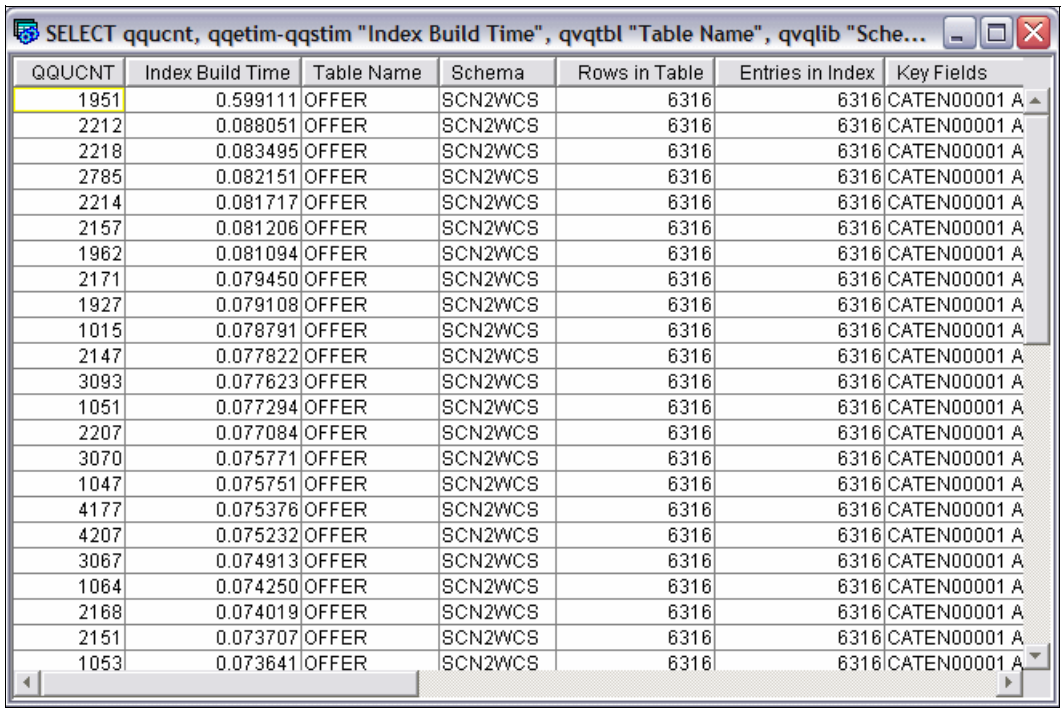
The created indexes might only contain keys for rows that satisfy the query (such indexes are known as *sparse indexes* or *select/omit logical files*). In many cases, the index created might be perfectly normal and the most efficient way to perform the query.

Look at the query in Example 6-21, which list temporary index builds ordered by the cost of the index build time.

Example 6-21 Temporary index builds

```
SELECT qqucnt, qqetim-qqstim "Index Build Time", qvqtbl "Table Name", qvqlib "Schema",
qqtotr "Rows in Table", qqridx "Entries in Index", SUBSTR(qq1000, 1, 100) "Key Fields",
qqidxa "Index Advised", SUBSTR(qqidxd, 1,100) "Keys Advised" FROM MYDBMON
WHERE  qqrid=3002 ORDER BY "Index Build Time" DESC;
```

Figure 6-17 shows the result of the query in Example 6-21.



SELECT qqucnt, qqetim-qqstim "Index Build Time", qvqtbl "Table Name", qvqlib "Sche...

QQUCNT	Index Build Time	Table Name	Schema	Rows in Table	Entries in Index	Key Fields
1951	0.599111	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2212	0.088051	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2218	0.083495	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2785	0.082151	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2214	0.081717	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2157	0.081206	OFFER	SCN2WCS	6316	6316	CATEN00001 A
1962	0.081094	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2171	0.079450	OFFER	SCN2WCS	6316	6316	CATEN00001 A
1927	0.079108	OFFER	SCN2WCS	6316	6316	CATEN00001 A
1015	0.078791	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2147	0.077822	OFFER	SCN2WCS	6316	6316	CATEN00001 A
3093	0.077623	OFFER	SCN2WCS	6316	6316	CATEN00001 A
1051	0.077294	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2207	0.077084	OFFER	SCN2WCS	6316	6316	CATEN00001 A
3070	0.075771	OFFER	SCN2WCS	6316	6316	CATEN00001 A
1047	0.075751	OFFER	SCN2WCS	6316	6316	CATEN00001 A
4177	0.075376	OFFER	SCN2WCS	6316	6316	CATEN00001 A
4207	0.075232	OFFER	SCN2WCS	6316	6316	CATEN00001 A
3067	0.074913	OFFER	SCN2WCS	6316	6316	CATEN00001 A
1064	0.074250	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2168	0.074019	OFFER	SCN2WCS	6316	6316	CATEN00001 A
2151	0.073707	OFFER	SCN2WCS	6316	6316	CATEN00001 A
1053	0.073641	OFFER	SCN2WCS	6316	6316	CATEN00001 A

Figure 6-17 Temporary index builds

We explain some of the columns used in the previous query:

- QQUCNT

  This column uniquely identifies a query for a given job. It includes QQJFLD to uniquely identify the query across many jobs.

- QVQTBL and QVQLIB

  These columns indicate the long SQL table name and the long SQL schema name. Use QQTFN and QQTLN for short object names.

- QQETIM-QQSTIM

  The difference between ending and starting time stamps is represented as a decimal (20.6). It is acceptable to use it as index build costing criteria, which is why our query uses it in the ORDER BY clause.

- QQTOTR and QQRIDX

  Compare the total rows in the table with the entries in the temporary index to gauge whether the selection is built into a temporary index. If QQTOTR is greater than QQRIDX, then the selection is built into a temporary index and you should carefully consider any advised indexes located in QQIDXD field. In general, selection criteria keys should precede QQ1000 keys, since QQ1000 fields are usually advised to satisfy join, ordering, or grouping criteria.

- QQ1000

  For this particular row type, the QQ1000 column contains key fields that are advised by the query optimizer to satisfy criteria for join, ordering, grouping, scroll able cursors, and so on. These are the reasons for the temporary index build. Therefore, use these keys in any permanent index that you are going to build. Pay attention to any selection criteria

advised fields in QQIDXD and consider including them as the left-most key values in the permanent index that you are going to build.

- QQIDXD

  In the list of key columns in column QQIDXD, the optimizer lists 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 might or might not significantly reduce the number of keys selected.

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

- QQRID

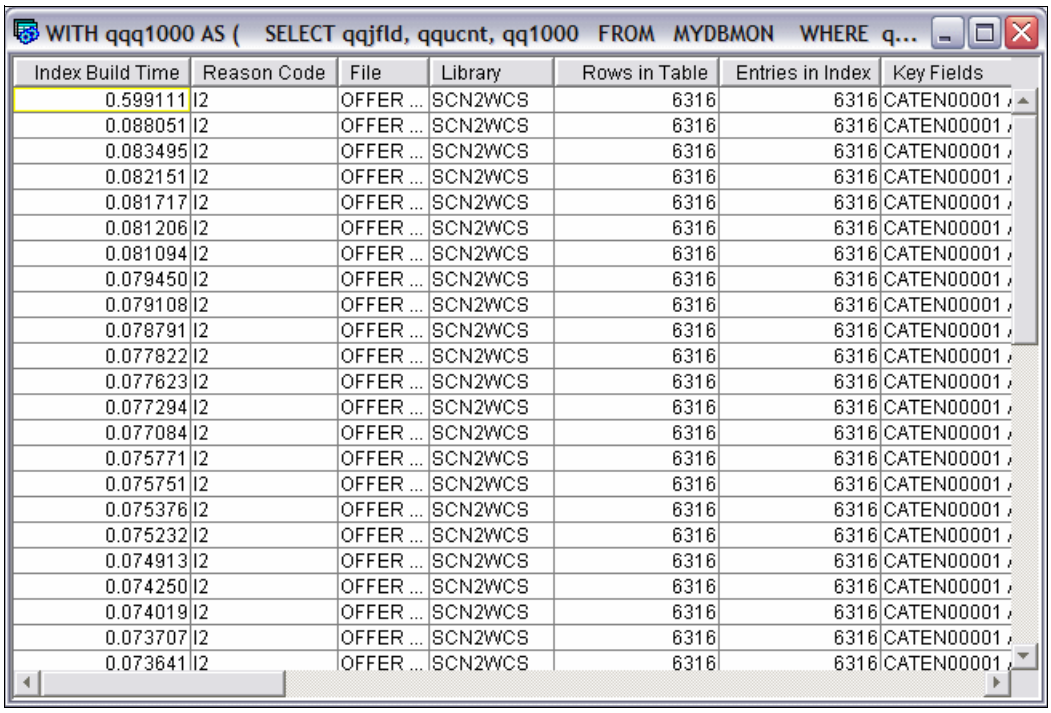
  A row type of 3002 indicates a temporary index build so we have it in our selection criteria.

We modify the previous query to gather more information about the advised index and query optimizer reasons for building the temporary index. Example 6-22 shows the modified query.

Example 6-22 Reason for building a temporary index

```
WITH qqq1000 AS (
  SELECT qqjfld, qqucnt, qq1000   FROM   MYDBMON
  WHERE  qqrid = 1000 and qqc21 <> 'MT' AND
         (qvc1c = 'Y' OR (qqc21 IN('DL', 'UP') AND qqc181 <= ' ') OR
qqc21 IN ('IN', 'IC', 'SK', 'SI') OR qqc21 LIKE 'O%'))
SELECT a.qqetim-a.qqstim "Index Build Time", a.qqrcod "Reason Code", a.qqtfn "File",
a.qqtln "Library", a.qqtotr "Rows in Table", a.qqridx "Entries in Index",
SUBSTR(a.qq1000, 1, 100) "Key Fields", a.qqidxa "Index Advised", a.qqi2 "Nbr of Primary
Keys", SUBSTR(a.qqidxd, 1,100) "Keys Advised", a.qvc16 "Index from index", b.qq1000
FROM MYDBMON a LEFT JOIN qqq1000 b ON a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt
WHERE a.qqrid=3002  ORDER BY "Index Build Time" DESC;
```

Figure 6-18 shows the output of the query in Example 6-22.


WITH qqq1000 AS (SELECT qqjfld, qqucnt, qq1000 FROM MYDBMON WHERE q...

Index Build Time	Reason Code	File	Library	Rows in Table	Entries in Index	Key Fields
0.599111	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.088051	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.083495	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.082151	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.081717	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.081206	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.081094	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.079450	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.079108	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.078791	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.077822	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.077623	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.077294	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.077084	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.075771	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.075751	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.075376	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.075232	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.074913	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.074250	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.074019	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.073707	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /
0.073641	I2	OFFER ...	SCN2WCS	6316	6316	CATEN00001 /

Figure 6-18 Reason for building temporary index

We joined our initial query back to record 1000 of the base Database Monitor table to obtain the SQL text for which the temporary index was built. The query uses the following columns:

- QQ1000

  This column indicates that SQL statement that caused query optimizer to build a temporary index.

- QQC21, QVC1C and QQC181

  When joining back to row type 1000, we care only about the operations that can cause query optimizer to advise an index. Therefore, we check for all the appropriate operation types contained in field QQC21. Additional criteria might be contained in the QVC1C field, SQL Statement Explainable, and QQC181, Cursor Name for the statement.

- QQRCOD

  This column indicates the reason code for an index build. You most commonly see the following reason codes:

  I2 Ordering/grouping
  I4 Nested loop join
  I3 Selection and ordering/grouping

  For a detailed list of possible reason codes, search on `Database Monitor: DDS` in the V5R3 iSeries Information Center.

  `http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp`

- QQI2

  This column 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 that QQIDXK contains a value of 4

and QQIDXD specifies seven key columns, then the first 4 key columns specified in QQIDXK are the primary key columns. The remaining three key columns are the suggested secondary key columns.

- QVC16

  This column indicates that a temporary index was built from an existing index, which is usually a short running operation.

Note: Building permanent indexes to replace temporary indexes can provide great returns for a little time spent in analyzing Database Monitor data. Do not overuse this easy method for short running and nonrepetitive temporary index builds.

6.3.10 Index advised

The query optimizer advises indexes and places the advise in the Database Monitor table when the *DETAILED option is specified. The optimizer analyzes the row selection in the query and determines, based on default values, if the 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. Advised indexes can be used to quickly tell if the optimizer recommends creating a specific permanent index to improve performance.

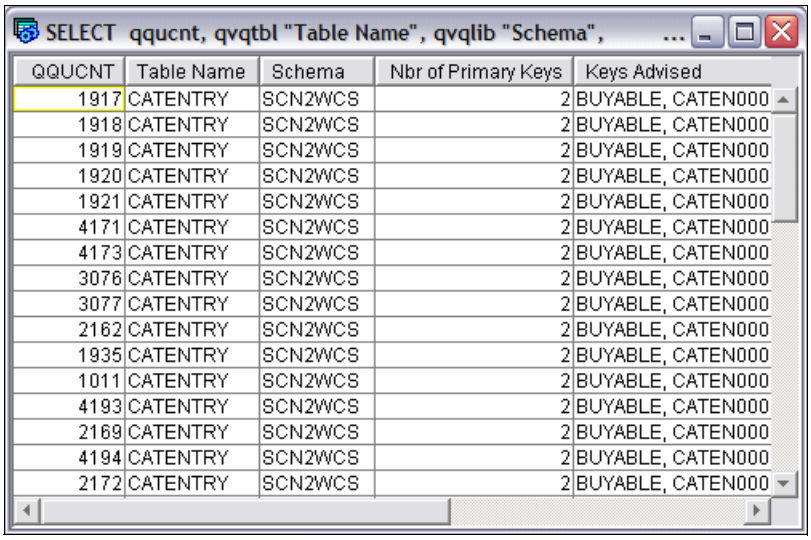
While creating an index that is advised typically improves performance, this is not a guarantee. After the index is created, much more accurate estimates of the actual costs are available. Based on this new information, the optimizer might decide that the cost of using the index is too high. Even if the optimizer does not use the index to implement the query, the new estimates available from the new index provide more detailed information to the optimizer that might produce better performance.

To look for the indexes advised by the optimizer, use the query shown in Example 6-23.

Example 6-23 Index advised

```
SELECT  qqucnt, qvqtbl "Table Name", qvqlib "Schema",
        qqi2 "Nbr of Primary Keys",
        SUBSTR(qqidxd, 1,100) "Keys Advised"
FROM MYDBMON
WHERE   qqrid IN (3000, 3001, 3002) and qqidxa='Y'
ORDER BY 5,2;
```

Figure 6-19 shows the results of the query in Example 6-23.


SELECT qqucnt, qvqtbl "Table Name", qvqlib "Schema", ...

QQUCNT	Table Name	Schema	Nbr of Primary Keys	Keys Advised
1917	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
1918	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
1919	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
1920	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
1921	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
4171	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
4173	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
3076	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
3077	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
2162	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
1935	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
1011	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
4193	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
2169	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
4194	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000
2172	CATENTRY	SCN2WCS	2	BUYABLE, CATEN000

Figure 6-19 Index advised

This query uses the following columns:

- QQUCNT

  This column uniquely identifies a query for a given job. Include QQJFLD to uniquely identify a query across many jobs.

- QVQTBL and QVQLIB

  These columns indicate the long SQL table name and the long SQL schema name. Use QQTFN and QQTLN for the short object names.

- QQIDXD

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

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

- QQI2

  This column contains the number of suggested primary key columns that are listed in the QQIDXD column. 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 that QQIDXK contains a value of 4 and QQIDXD specifies seven key columns, then the first 4 key columns specified in QQIDXK are the primary key columns. The remaining three key columns are the suggested secondary key columns.

- QQRID

  Index advice is contained in three different row types and the query looks at all of them:

  – 3000 - Table scan operation

    We discussed the table scan operation previously as well as the need for indexes where a query is highly selective.

  – 3001 - Index used

    In cases where an existing index was used, query optimizer might still recommend an index. A recommended index might be better than the selected index, but not always. Keep in mind that advised keys are for selection only and that you need to consider JOIN, ORDER BY, or GROUP BY clause criteria.

  – 3002 - Temporary index created

    For temporary indexes, we recommend that you use a different query altogether because we don't illustrate the QQ1000 column for row type 3002 in this query. In this case, QQ1000 includes keys used for a temporary index as well as their order (ascending or descending).

- QQIDXA

  We specified Y in this column since we are interested only in the rows for which query optimizer has advised indexes.

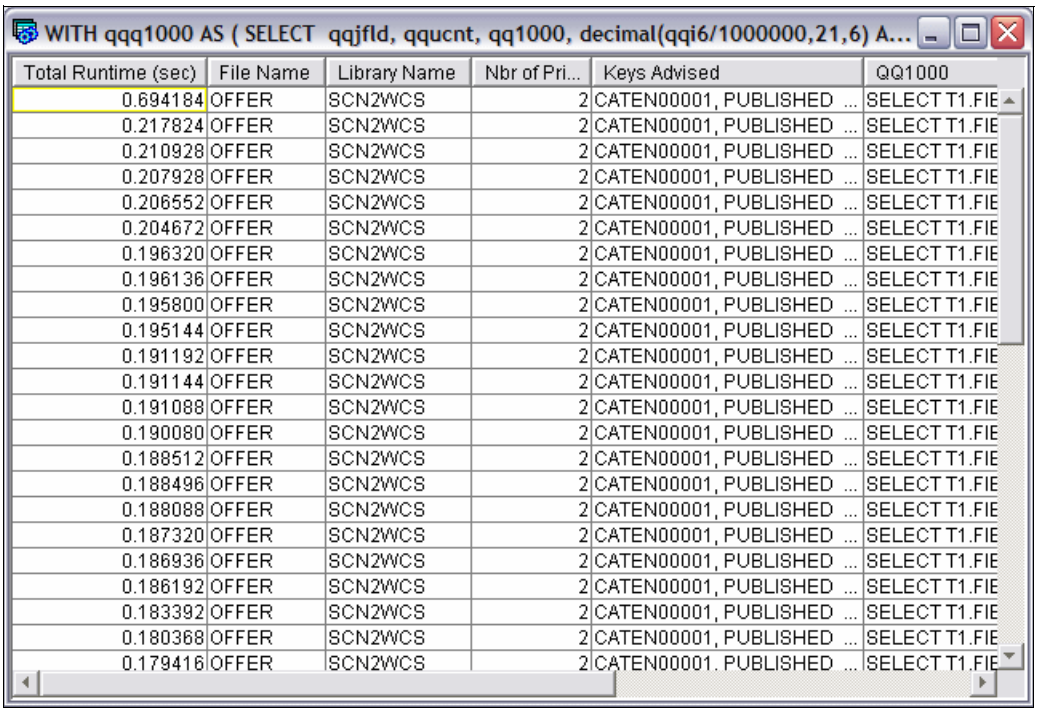
Our initial query serves our need perfectly when we collect a very specific set of data (that is, a single job). However, if you perform a system-wide Database Monitor collection, you must use a query that is a bit more sophisticated.

We look at a query that enables us to perform costing by total run time required by the SQL statements that caused indexes to be advised. Example 6-24 shows this query.

Example 6-24 Costing of SQL statements where an index is advised

```
WITH qqq1000 AS ( SELECT  qqjfld, qqucnt, qq1000,
decimal(qqi6/1000000,21,6) AS "Total Runtime (sec)"
FROM MYDBMON
WHERE qqrid = 1000 AND qqi5 = 0 and qqc21 <> 'MT' AND (qvc1c = 'Y' OR (qqc21 IN('DL', 'UP')
AND qqc181 <= ' ') OR
qqc21 IN ('IN', 'IC', 'SK', 'SI') OR qqc21 LIKE 'O%'))
SELECT   b."Total Runtime (sec)", a.qqtfn "File Name",
         a.qqtln "Library Name", a.qqi2 "Nbr of Primary Keys",
         substr(a.qqidxd,1,100) "Keys Advised", b.qq1000
FROM     MYDBMON a LEFT JOIN qqq1000 b ON
         a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt WHERE    qqrid IN (3000,3001,3002) AND
qqidxa = 'Y' ORDER BY "Total Runtime (sec)" DESC;
```

Figure 6-20 shows the output of the query in Example 6-24.



WITH qqq1000 AS (SELECT qqjfld, qqucnt, qq1000, decimal(qqi6/1000000,21,6) A...

Total Runtime (sec)	File Name	Library Name	Nbr of Pri...	Keys Advised	QQ1000
0.694184	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.217824	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.210928	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.207928	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.206552	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.204672	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.196320	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.196136	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.195800	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.195144	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.191192	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.191144	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.191088	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.190080	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.188512	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.188496	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.188088	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.187320	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.186936	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.186192	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.183392	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.180368	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE
0.179416	OFFER	SCN2WCS	2	CATEN00001, PUBLISHED ...	SELECT T1.FIE

Figure 6-20 Costing of SQL statements where an index is advised

We joined our initial query back to record 1000 of the base Database Monitor table to obtain the SQL text and total statement run time. This query uses the following columns:

- QQ1000

  This column indicates that an SQL statement caused query optimizer to advise an index.

- QQI6

  This column indicates the total runtime required by the query, which is a good indicator of the index benefit for a table scan and temporary index build operations. For the existing index factors, such as join, grouping and ordering play into total runtime calculation, so this field might not be an accurate cost indicator.

- QQC21, QVC1C and QQC181

  When joining back to row type 1000, we are concerned about only the operations that can cause query optimizer to advise an index. Therefore, we check for all the appropriate operation types contained in column QQC21. Additional criteria might be contained in the QVC1C column, SQL Statement Explainable, and QQC181, Cursor Name for the statement.

The remainder of the statement is much like the initial query. One difference is that here we order by total runtime of the SQL statement, which provides us with a good costing indicator. This costing indicator helps us to focus on the worst performing statements and build indexes intelligently.

You can see the number of different indexes that are advised and how many times they are advised. To see this information, you run the query shown in Example 6-25.

Example 6-25 Listing of distinct index advised

```
SELECT  distinct qvqtbl "Table", qqidxd "Key Fields", count(*) "Times adviced"
FROM MYDBMON
WHERE qqrid IN (3000, 3001, 3002) and qqidxa='Y'
group by qvqtbl, qqidxd
order by qvqtbl, qqidxd;
```

Figure 6-21 shows the results of the query in Example 6-25.

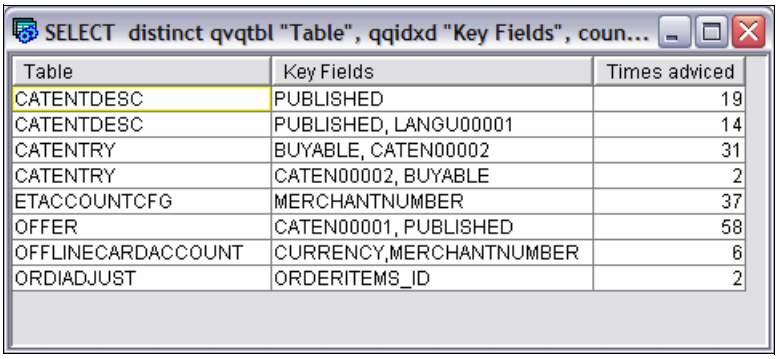


Table	Key Fields	Times adviced
CATENTDESC	PUBLISHED	19
CATENTDESC	PUBLISHED, LANGU00001	14
CATENTRY	BUYABLE, CATEN00002	31
CATENTRY	CATEN00002, BUYABLE	2
ETACCOUNTCFG	MERCHANTNUMBER	37
OFFER	CATEN00001, PUBLISHED	58
OFFLINECARDACCOUNT	CURRENCY,MERCHANTNUMBER	6
ORDIADJUST	ORDERITEMS_ID	2

Figure 6-21 Different index advised

This is a good example where creation of the index with the key field PUBLISHED is not necessary, because creation of the index with the key fields PUBLISHED, LANGU00001 covers both index recommendations, with a total of 33 times advised.

6.3.11 Access plan rebuilt

An access plan consists of one or more integrated steps (nodes) that are assembled to retrieve and massage data from DB2 tables to produce results that are desired by the information requestor. These steps might involve selecting, ordering, summarizing, and aggregating data elements from a single table or from related (joined) rows from multiple tables.

Each SQL query executes an access plan to retrieve the data that you requested. If the access plan does not exist already, the system builds one dynamically, adding overhead to the total time required to satisfy your request.

As a general rule, we want to avoid access plan rebuilds. That said, there are several perfectly valid reasons for access plan rebuilds, for example:

- Deleting or recreating the table to which the access plan refers
- Deleting an index that is used by the access plan
- Applying database PTFs
- Table size changing by 10%
- Creating a new statistic, automatically or manually
- Refreshing a statistic, automatically or manually
- Removing a stale statistic
- CQE rebuilding an access plan if there was a two-fold change in memory pool size

- SQL Query Engine (SQE) looking for a ten-fold change in memory pool size if memory pool is defined with a pool paging option of *CALC (also known as *expert cache*) (If paging is set to *FIXED, SQE behaves the same as CQE.)
- Specifying REOPTIMIZE_ACCESS_PLAN (*YES) in the QAQQINI table or in the SQL script
- Specifying REOPTIMIZE_ACCESS_PLAN (*FORCE) in the QAQQINI table or in the SQL script
- Changing the number of CPUs (whole or fractions using logical partition (LPAR)) that are available to a query results in rebuilding the access plan.
- Access plans are marked as invalid after an OS/400 release upgrade.
- The SQE Plan Cache is cleared when a system initial program load (IPL) is performed.

The SQE access plan rebuild activity takes place below the machine interface (MI). Therefore, compared to CQE, you should see much less performance degradation caused by lock contention on SQL packages, caches, and program objects.

At times, even though the optimizer rebuilt the access plan, the system fails to update the program object. The most common reason for this failure is that other jobs are using the same program and optimizer cannot obtain the exclusive lock on the program object to save the rebuilt access plan. Another reason is that the job does not have proper authority to the program or the program is currently being saved. The query still runs, but access plan rebuilds continue to occur until the program is updated.

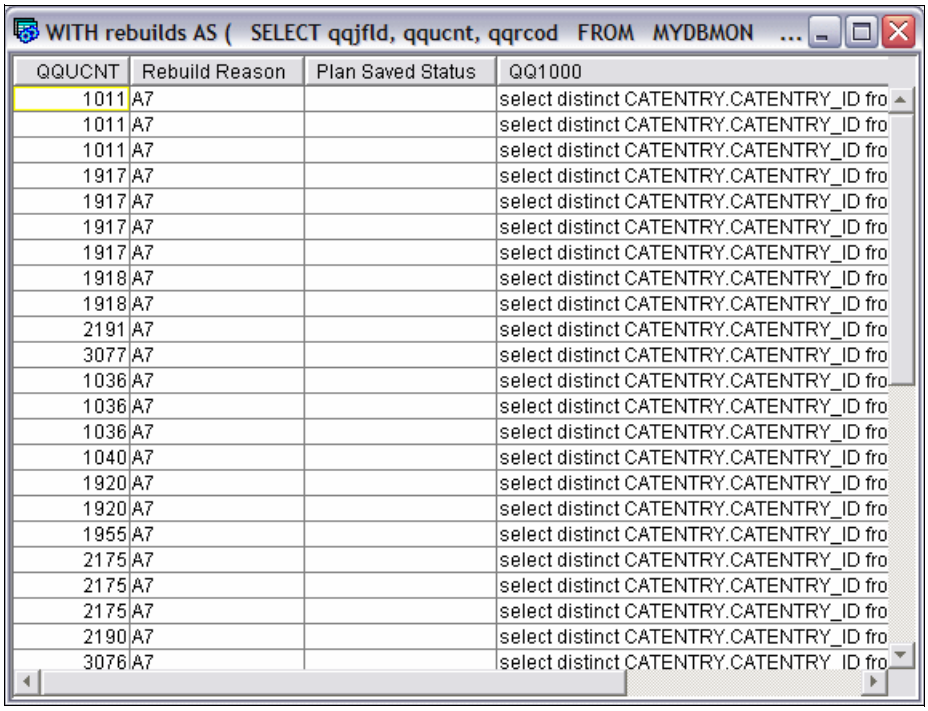
The rebuilt access plan might be saved in the existing access plan space within the program, SQL package, or cache. If a new access plan is greater than the existing one, new space is allocated and the plan is saved in that newly allocated space. If the number of access plan rebuilds is high, some application redesign might be necessary.

We look at access plan rebuild information that is available in Database Monitor data by using the query shown in Example 6-26.

Example 6-26 Access plan rebuilds information

```
WITH rebuilds AS (
  SELECT qqjfld, qqucnt, qqrcod
  FROM   MYDBMON
  WHERE  qqrid=3006 )
SELECT   a.qqucnt, b.qqrcod "Rebuild Reason",
         qvc24 "Plan Saved Status", qq1000
FROM     MYDBMON a, rebuilds b
WHERE    a.qqjfld=b.qqjfld AND a.qqrid=1000 AND
         a.qqc21 NOT IN ('MT','FE','CL','HC')
ORDER BY 4, 1;
```

Figure 6-22 shows the result of the query in Example 6-26.



QQUCNT	Rebuild Reason	Plan Saved Status	QQ1000
1011	A7		select distinct CATENTRY.CATENTRY_ID fro
1011	A7		select distinct CATENTRY.CATENTRY_ID fro
1011	A7		select distinct CATENTRY.CATENTRY_ID fro
1917	A7		select distinct CATENTRY.CATENTRY_ID fro
1917	A7		select distinct CATENTRY.CATENTRY_ID fro
1917	A7		select distinct CATENTRY.CATENTRY_ID fro
1917	A7		select distinct CATENTRY.CATENTRY_ID fro
1918	A7		select distinct CATENTRY.CATENTRY_ID fro
1918	A7		select distinct CATENTRY.CATENTRY_ID fro
2191	A7		select distinct CATENTRY.CATENTRY_ID fro
3077	A7		select distinct CATENTRY.CATENTRY_ID fro
1036	A7		select distinct CATENTRY.CATENTRY_ID fro
1036	A7		select distinct CATENTRY.CATENTRY_ID fro
1036	A7		select distinct CATENTRY.CATENTRY_ID fro
1040	A7		select distinct CATENTRY.CATENTRY_ID fro
1920	A7		select distinct CATENTRY.CATENTRY_ID fro
1920	A7		select distinct CATENTRY.CATENTRY_ID fro
1955	A7		select distinct CATENTRY.CATENTRY_ID fro
2175	A7		select distinct CATENTRY.CATENTRY_ID fro
2175	A7		select distinct CATENTRY.CATENTRY_ID fro
2175	A7		select distinct CATENTRY.CATENTRY_ID fro
2190	A7		select distinct CATENTRY.CATENTRY_ID fro
3076	A7		select distinct CATENTRY.CATENTRY_ID fro

Figure 6-22 Access plan rebuilds information

This query uses the following columns:

- QQRID

  Our common table expression rebuild contains only row type 3006, which has information specific to access plan rebuilds. Row type 3006 is *not* present on every full open. It is only generated when an access plan previously existed and now has to be rebuilt.

  Row type 3006 is also not generated when SQE switches between cached access plans (up to three) in the SQE plan cache for an SQL statement.

- QQJFLD and QQUCNT

  By now you know that QQJFLD and QQUCNT are join fields required to uniquely identify the SQL statement in the Database Monitor file.

- QQRCOD

  This column provides the reason code for the access plan rebuild. There are over twenty possible reason codes. For a detailed description of the reason codes, search on `Database Monitor: DDS` in the V5R3 iSeries Information Center.

  `http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp`

- QVC24

  This column indicates the reason code for why the access plan was saved or not saved.

  All A* codes mean that the access plan was *not* saved. AB means that a lock could not be obtained, A6-A9 means that not enough space was available, and AA means that a plan was saved by another job.

  All B* codes mean that the access plan was saved, with a blank value or B3, B4, or B6 meaning that a plan was saved “in place”, and B1, B2, B5, B7, or B8 meaning that the plan was saved in a “new” space.

For a detailed description of each reason code, search on `Database Monitor: DDS` in the V5R3 iSeries Information Center.

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp

- QQ1000

  This column indicates the SQL statement that caused query optimizer to rebuild the access plan.

- QQC21

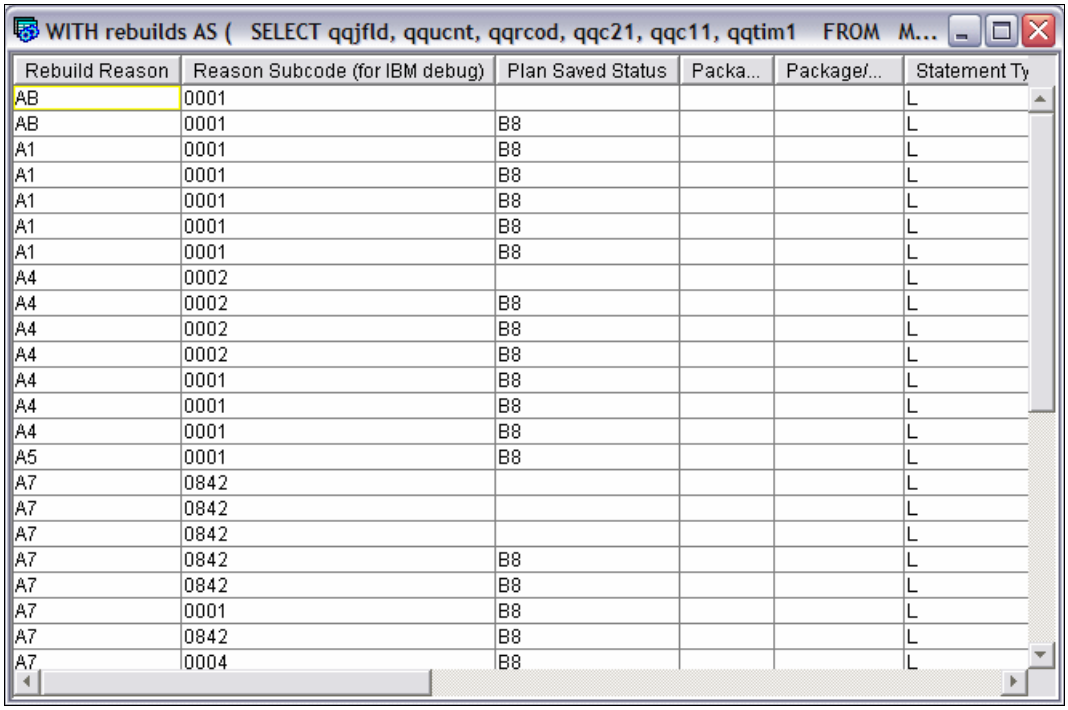
  In this query field, QQC21 is pulled from row type 1000 and represents the operation type for the SQL statement. We exclude continuation records, fetches, closes, and hard closes. The output is in ascending order based on the statement text and the unique statement counter. This query provides basic information about access plan rebuilds.

Let us rewrite the query to include more information related to access plan rebuilds as shown in Example 6-27.

Example 6-27 Extended access plan rebuild information

```
WITH rebuilds AS (
  SELECT qqjfld, qqucnt, qqrcod, qqc21, qqc11, qqtim1    FROM   MYDBMON
  WHERE  qqrid=3006 ) SELECT   b.qqrcod "Rebuild Reason",
         hex(b.qqc21) "Reason Subcode (for IBM debug)", a.qvc24 "Plan Saved Status",
a.QQC103 "Package/Program Name",
a.QQC104 "Package/Program Library", a.qvc18 "Statement Type", b.qqc11 "Plan Reoptimized",
b.qqtim1 "Last Rebuilt", a.qq1000
FROM MYDBMON a, rebuilds b WHERE a.qqjfld=b.qqjfld AND a.qqucnt = b.qqucnt and a.qqrid=1000
AND a.qqc21 NOT IN ('MT','FE','CL','HC')  ORDER BY "Rebuild Reason", "Plan Saved Status";
```

Figure 6-23 shows the output of the query in Example 6-27.



Rebuild Reason	Reason Subcode (for IBM debug)	Plan Saved Status	Packa...	Package/...	Statement Ty
AB	0001				L
AB	0001	B8			L
A1	0001	B8			L
A1	0001	B8			L
A1	0001	B8			L
A1	0001	B8			L
A1	0001	B8			L
A4	0002				L
A4	0002	B8			L
A4	0002	B8			L
A4	0002	B8			L
A4	0001	B8			L
A4	0001	B8			L
A4	0001	B8			L
A5	0001	B8			L
A7	0842				L
A7	0842				L
A7	0842				L
A7	0842	B8			L
A7	0842	B8			L
A7	0001	B8			L
A7	0842	B8			L
A7	0004	B8			L

Figure 6-23 Extended access plan rebuild information

The query uses the following columns:

- QQC21

  In this query, we pull the QQC21 column from row type 3006. In this row type, field QQC21 contains the access plan rebuild reason subcode. This subcode is in hex and should only be used when reporting a problem to IBM Support.

  We still use QQC21 from row type 1000 to exclude undesired 1000 records, like we did in the initial query.

- QQC11

  This column contains a simple character flag that indicates if an access plan required optimization. If the value is Y, then the plan was reoptimized. If the value is N, then the plan was not reoptimized. If this value is never Y, it is possible that QAQQINI file contains a REOPTIMIZE_ACCESS_PLAN setting that prevents the query optimizer from reoptimizing the access plan.

- QQTIM1

  The value in this column indicates the time stamp of last access plan rebuild.

- QQC103

  This column contains the name of the package or program that contains the SQL statement that caused query optimizer to rebuild the access plan.

- QQC104

  This column indicates the name of the library that contains the program or package listed in the QQC103 column.

- QVC18

  This column describes query statement type. If statement was done as part of dynamic processing, it is flagged with E for “extended dynamic”, with S for “system wide cache”, and L for “local prepared statement”.

6.3.12 Query sorting

The 3003 record from the Database Monitor table shows that the database optimizer has decided to place selected rows into a temporary space and sort them. The presence of a 3003 record does not necessarily indicate poor performance. The optimizer selected a query sort because it is either cheaper than the alternative indexed methods or it is forced to do so, for example when UNION is used or ORDER BY uses columns from more than one table.

Indexes can still be used to select or join rows before the sort occurs. The 3006 record does not indicate that the ODP is nonreusable.

Sort buffers are refilled and sorted at open time, even in reusable ODP mode.

Sorting might increase the open (OP) time/cost since sorting is often performed at open (OP) time. This means that it might take some time to return the first row in the result set to the end user.

High elapsed times for a query sort might indicate a large answer set. In this case, the sort outperforms index usage (the situation in most cases). You should not attempt to build indexes for queries with large result sets, unless you are going to add selection criteria to the SQL statement’s WHERE clause to further reduce the result set.

If the answer set is small, but the optimizer does not have the right indexes available to know that, creating indexes over selection columns can help by giving the optimizer statistics and

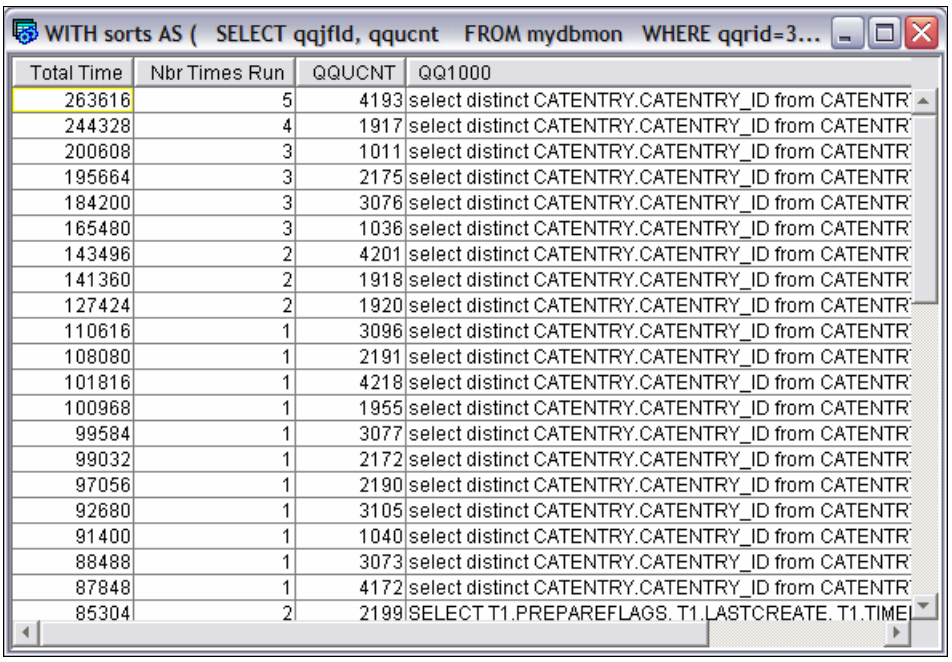
an alternative method of accessing the data. This is possible only if the optimizer is not forced to use the sort (that is via a UNION or ORDER BY on columns from more than one table).

Look at which queries involve the use of a query sort. You can do a query sort by using the query shown in Example 6-28.

Example 6-28 Use of a query sort

```
WITH sorts AS (
  SELECT qqjfld, qqucnt
  FROM mydbmon
  WHERE qqrid=3003 )
SELECT SUM(qqi6) "Total Time" , COUNT(*) "Nbr Times Run", a.qqucnt, qq1000
FROM mydbmon a, sorts b
WHERE qqrid=1000 AND a.qqjfld=b.qqjfld and a.qqucnt=b.qqucnt AND
      qqc21 IN ('OP','SI','UP','IN','DL')
GROUP BY a.qqucnt, qq1000
ORDER BY "Total Time" DESC;
```

Figure 6-24 shows the output of the query in Example 6-28.



Total Time	Nbr Times Run	QQUCNT	QQ1000
263616	5	4193	select distinct CATENTRY.CATENTRY_ID from CATENTR
244328	4	1917	select distinct CATENTRY.CATENTRY_ID from CATENTR
200608	3	1011	select distinct CATENTRY.CATENTRY_ID from CATENTR
195664	3	2175	select distinct CATENTRY.CATENTRY_ID from CATENTR
184200	3	3076	select distinct CATENTRY.CATENTRY_ID from CATENTR
165480	3	1036	select distinct CATENTRY.CATENTRY_ID from CATENTR
143496	2	4201	select distinct CATENTRY.CATENTRY_ID from CATENTR
141360	2	1918	select distinct CATENTRY.CATENTRY_ID from CATENTR
127424	2	1920	select distinct CATENTRY.CATENTRY_ID from CATENTR
110616	1	3096	select distinct CATENTRY.CATENTRY_ID from CATENTR
108080	1	2191	select distinct CATENTRY.CATENTRY_ID from CATENTR
101816	1	4218	select distinct CATENTRY.CATENTRY_ID from CATENTR
100968	1	1955	select distinct CATENTRY.CATENTRY_ID from CATENTR
99584	1	3077	select distinct CATENTRY.CATENTRY_ID from CATENTR
99032	1	2172	select distinct CATENTRY.CATENTRY_ID from CATENTR
97056	1	2190	select distinct CATENTRY.CATENTRY_ID from CATENTR
92680	1	3105	select distinct CATENTRY.CATENTRY_ID from CATENTR
91400	1	1040	select distinct CATENTRY.CATENTRY_ID from CATENTR
88488	1	3073	select distinct CATENTRY.CATENTRY_ID from CATENTR
87848	1	4172	select distinct CATENTRY.CATENTRY_ID from CATENTR
85304	2	2199	SELECT T1.PREPAREFLAGS, T1.LASTCREATE, T1.TIME

Figure 6-24 Use of a query sort

This query uses the following columns:

- QQRID

  Our common table expression sort contains only row type 3003, which has information specific to SQL statements using query sorts.

- QQJFLD and QQUCNT

  The QQJFLD and QQUCNT are join fields required to uniquely identify the SQL statement in the Database Monitor file.

- QQC21

  Since we are joining common table expression sorts back to the 1000 record in the Database Monitor file, we must ensure that we only join to 1000 records that can cause

query sorts to occur. This is accomplished by verifying that QQC21 field operation is either open, select, update, delete, or insert. We need to include last three operations because they might have subselects or correlated subqueries.

- QQI6

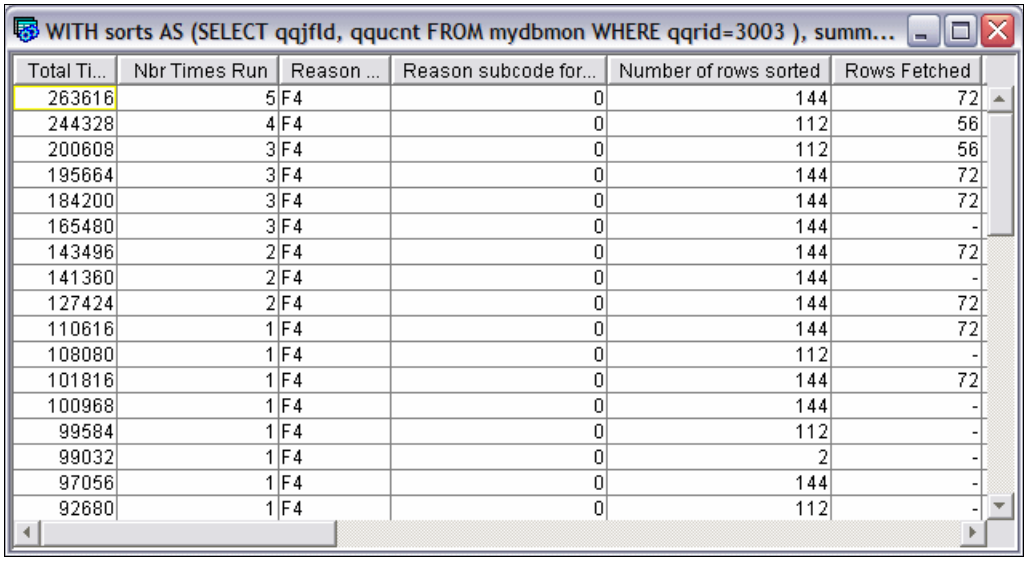
  This column indicates a table scan operation cumulative elapse time, in microseconds for each individual query. Since we use it as a cost indicator, we have ordered the output in descending order based on this value.

The query that we have outlined so far is insufficient in helping us to decide if building an index or modifying the SQL statement is desired. Therefore, we revise the query (see Example 6-29) to include data that is necessary to make the decision if any action is possible.

Example 6-29 Including data showing possible action

```
WITH sorts AS (SELECT qqjfld, qqucnt FROM mydbmon WHERE qqrid=3003 ),
summation AS (SELECT SUM(qqi6) "Total Time" , COUNT(*) "Nbr Times Run", a.qqjfld,
a.qqucnt, qq1000 FROM mydbmon a, sorts b WHERE qqrid=1000 AND a.qqjfld=b.qqjfld AND
a.qqucnt = b.qqucnt and qqc21 IN ('OP','SI','UP','IN','DL') GROUP BY a.qqjfld, a.qqucnt,
qq1000), fetches AS (SELECT a.qqjfld, a.qqucnt, integer(avg(a.qqi3)) "Rows Fetched" FROM
mydbmon a, summation b   WHERE qqrid=1000 AND a.qqjfld=b.qqjfld and a.qqucnt = b.qqucnt AND
          qqc21 = 'FE'   GROUP BY a.qqjfld, a.qqucnt) SELECT   b."Total Time", b."Nbr
Times Run",
         a.qqrcod "Reason Code", a.qqi7 "Reason subcode for Union",
         a.qqrss "Number of rows sorted", c."Rows Fetched",
         a.qqi1 "Size of Sort Space",  a.qqi2 "Pool Size",
         a.qqi3 "Pool ID", a.qvbndy "I/O or CPU bound", a.qqucnt, b.qq1000 FROM
summation b LEFT OUTER JOIN fetches c ON b.qqjfld = c.qqjfld AND
         b.qqucnt = c.qqucnt INNER JOIN mydbmon a
         ON b.qqjfld = a.qqjfld AND b.qqucnt = a.qqucnt WHERE    a.qqrid = 3003 ORDER BY
b."Total Time" DESC;
```

Figure 6-25 shows the output from the query in Example 6-29.



Total Ti...	Nbr Times Run	Reason ...	Reason subcode for...	Number of rows sorted	Rows Fetched
263616	5	F4	0	144	72
244328	4	F4	0	112	56
200608	3	F4	0	112	56
195664	3	F4	0	144	72
184200	3	F4	0	144	72
165480	3	F4	0	144	-
143496	2	F4	0	144	72
141360	2	F4	0	144	-
127424	2	F4	0	144	72
110616	1	F4	0	144	72
108080	1	F4	0	112	-
101816	1	F4	0	144	72
100968	1	F4	0	144	-
99584	1	F4	0	112	-
99032	1	F4	0	2	-
97056	1	F4	0	144	-
92680	1	F4	0	112	-

Figure 6-25 Include data showing action possible

This query uses the following columns:

- QQRCOD

  This column indicates the reason for choosing the query sort technique. The value in this column helps to identify whether the sort required of the query optimizer determined that the cost of the sort is better than any other implementation (such as an index).

  If you can change the SQL statement itself, any reason code is available for optimization efforts. Or perhaps you cannot change the SQL statement (that is to optimize the third-party ERP application) and can only build indexes and change other environmental factors to help performance (that is, increase the pool size). In this case, focus your optimization efforts on query sorts with reason code F7 (optimizer chose sort rather than index due to performance reasons) and F8 (optimizer chose sort to minimize I/O wait time).

  For a detailed description of each reason code, search on `Database Monitor: DDS` in the V5R3 iSeries Information Center.

  `http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp`

- QQI7

  This column indicates the reason subcode for the UNION clause. If the query sort reason code lists F5 (UNION was specified for query), this column contains one of two subcodes. A value of 51 means that there is also an ORDER BY in the statement. A value of 52 means that the query specifies UNION ALL rather than simply UNION.

- QQRSS

  This column tells us the number of rows that are contained in the sort space. You can use this value, along with the reason code, to determine if the indexed approach is possible and possibly cheaper (for a small result set). Compare the value of QQRSS with the value of QQI3 from the corresponding 1000 FE record for this open to determine the number of rows that were fetched from the sort space.

  If the number of rows in sort space is large, but the actual number of rows fetched is small, consider adding OPTIMIZE FOR *n* ROWS to the query to help the optimizer make a better decision.

  Building a more perfect index for the selection criteria might also help the optimizer make a better decision and use index for the implementation method rather than a query sort.

- QQI3 from row type 1000 operation id FE (fetch)

  This column tells us the number of rows that were fetched from the sort space to satisfy a user request. As described in the QQRSS column description, the value in this column is used to gauge whether more information is required by the query optimizer to make better costing decisions.

- QQI1

  This column indicates the size of the sort space.

- QQI2

  This column indicates the pool size.

- QQI3 from row type 3003

  This column indicates the pool ID.

- QVBNDY

  This column contains a flag that indicates whether the query sort is CPU or I/O bound.

We have taken the base query and modified it to include more information about the query sort implementation. This additional information helps you make more intelligent decisions

when deciding to optimize SQL statements using query sorts as the implementation method. The most valuable new columns indicate a reason code and the number of actual rows fetched for the query.

Changing your SQL statement or adding the OPTIMIZE FOR *x* ROWS syntax is most likely to help alleviate issues that pertain to long query sort times. For highly selective queries where sort space is disproportionately larger than actual rows fetched, building a more perfect index might help the optimizer.

6.3.13 SQE advised statistics analysis

With the introduction of SQE to OS/400 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 has two major functions:

- Create and maintain column statistics
- Answer questions that the optimizer asks when finding the best way to implement a given query

These answers can be derived from table header information, existing indexes, or single-column statistics. Single-column statistics provide estimates of column cardinality, most frequent values, and value ranges.

These values might have been previously available through an index, but statistics have the advantage of being precalculated and are stored with the table for faster access. Column statistics stored with a table do not dramatically increase the size of the table object. Statistics per column average only 8 to 12 KB in size. If none of these sources are available to provide a statistical answer, then the Statistics Manager bases the answer on default values (filter factors).

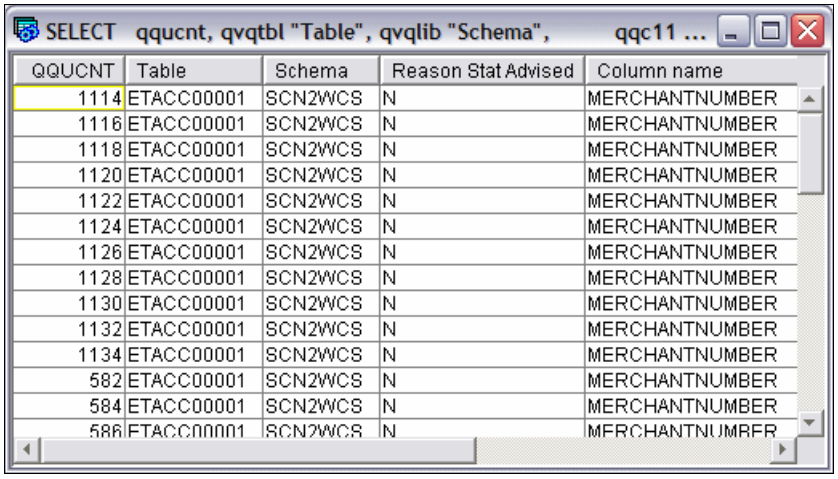
By default, this information is collected automatically by the system. You can manually control the collection of statistics by manipulating the QDBFSTCCOL system value or by using the iSeries Navigator graphical user interface (GUI). However, unlike indexes, statistics are not maintained immediately as data in the tables changes.

There are cases were the optimizer advises the creation of statistics. The query shown in Example 6-30 lists the optimizer advised statistics.

Example 6-30 Query optimizer advised statistics

```
SELECT   qqucnt, qvqtbl "Table", qvqlib "Schema",
         qqc11 "Reason Stat Advised",
         SUBSTR(qq1000,1,100) "Column name"
FROM mydbmon
WHERE qqrid=3015
ORDER BY 2,5;
```

Figure 6-26 shows the output of the query in Example 6-30.



QQUCNT	Table	Schema	Reason Stat Advised	Column name
1114	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1116	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1118	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1120	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1122	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1124	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1126	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1128	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1130	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1132	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
1134	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
582	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
584	ETACC00001	SCN2WCS	N	MERCHANTNUMBER
586	ETACC00001	SCN2WCS	N	MERCHANTNUMBER

Figure 6-26 Query optimizer advised statistics

This query uses the following columns:

- QQRID

  Our selection criteria selects only 3015 rows, which contain information exclusive to SQE advised statistics.

- QQJFLD and QQUCNT

  QQJFLD and QQUCNT are join fields that are required to uniquely identify the SQL statement in the Database Monitor file.

- QVQTBL and QVQLIB

  These columns refer to the long SQL table name and the long SQL schema name. Use QQTFN and QQTLN for short object names.

- QQC11

  This column indicates the reason that the statistic was advised. This can happen only for two reasons, where N indicates a new statistic and S indicates a stale statistic. A statistic can become stale for several reasons; one of the most common reasons is that a base physical table's number of rows has changed by 15 percent.

- QQ1000

  Column QQ1000 for row type 3015 contains the name of the column for which a statistic is advised. There might be multiple recommendations for a single SQL query, with each row containing a different column name in the QQ1000 column.

Keep in mind that column statistics are created in the background automatically by default for all advised statistics. Therefore, in general, no manual action is required on your end to build these statistics. The only exception is if the automatic statistics collection is turned off.

Although statistics provide a powerful mechanism for optimizing queries, do not underestimate and disregard the importance of implementing a sound indexing strategy. Well-defined indexes enable SQE to consistently provide efficient query optimization and performance. Statistics cannot be used to access a table or sort the data during query execution.

A good indexing strategy is both beneficial for providing statistics and mandatory for efficient and fast query execution. Therefore, you should replace indexes with statistics only if the

indexes were created for the sole purpose of providing statistical information to the query optimizer. In cases where an index provides fast and efficient access to a set of rows in a table, DB2 Universal Database for iSeries continues to rely on its indexing technology to provide statistics information and a data access method.

Since indexes are the premier way to improve query optimizer intelligence and data access implementation choices, we look at the query in Example 6-31, which correlates SQE advised statistics with any query optimizer recommended indexes.

Example 6-31 Correlation of SQE statistics and optimizer recommended indexes

```
WITH advisedIndexes AS (SELECT qqjfld, qqucnt, qqi2 "Nbr of Primary Keys",
SUBSTR(qqidxd, 1,100) "Keys Advised" FROM mydbmon
WHERE qqrid IN (3000, 3001, 3002) and qqidxa='Y') SELECT a.qqucnt, a.qqtfn "Table", a.qqtln
"Schema", CASE a.qqc11
WHEN 'N' THEN 'No Statistic Exists'
WHEN 'S' THEN 'Stale Statistic Exists'
ELSE 'Unknown'
END AS "Reason Stat Advised", SUBSTR(a.qq1000,1,100) "Column name", a.qqi2 "Statistics
Importance", a.qvc1000 "Statistics Identifier", b."Nbr of Primary Keys", b."Keys Advised"
FROM mydbmon a LEFT OUTER JOIN advisedIndexes b
on a.qqjfld = b.qqjfld and a.qqucnt = b.qqucnt WHERE qqrid=3015 ORDER BY a.qqucnt DESC;
```

Figure 6-27 shows the output of the query in Example 6-31.

WITH advisedIndexes AS (SELECT qqjfld, qqucnt, qqi2 "Nbr of Primary Keys", ...

QQUCNT	Table	Schema	Reason Stat Advised	Column name	Statistics Imp...
4216	ORDERITEMS	SCN2WCS	No Statistic Exists	INVENTORYSTATUS ...	0
3087	ORDERITEMS	SCN2WCS	No Statistic Exists	INVENTORYSTATUS ...	0
3080	ORDERITEMS	SCN2WCS	No Statistic Exists	INVENTORYSTATUS ...	0
2780	ORDERITEMS	SCN2WCS	No Statistic Exists	INVENTORYSTATUS ...	0
2186	ORDERITEMS	SCN2WCS	No Statistic Exists	INVENTORYSTATUS ...	0
2182	ORDERITEMS	SCN2WCS	No Statistic Exists	INVENTORYSTATUS ...	0
1270	ETACC00001	SCN2WCS	No Statistic Exists	MERCHANTNUMBER ...	0
1268	ETACC00001	SCN2WCS	No Statistic Exists	MERCHANTNUMBER ...	0
1266	ETACC00001	SCN2WCS	No Statistic Exists	MERCHANTNUMBER ...	0
1264	ETACC00001	SCN2WCS	No Statistic Exists	MERCHANTNUMBER ...	0
1262	ETACC00001	SCN2WCS	No Statistic Exists	MERCHANTNUMBER ...	0
1257	OFFLI00001	SCN2WCS	No Statistic Exists	CURRENCY ...	0
1254	ETACC00001	SCN2WCS	No Statistic Exists	MERCHANTNUMBER ...	0

Figure 6-27 Correlation of SQE statistic and optimizer recommended indexes

This query uses the following columns:

- QQIDXA

  This column contains a flag of Y or N that indicates whether index was advised. We use this information to filter out queries for which no index was recommended by the query optimizer (QQIDXA = 'Y').

- QQIDXD

  This column indicates columns for which an index was advised. This field lists primary keys first followed by zero or more secondary keys. Secondary selection keys are less likely to have a significant positive impact on a query's performance.

- QQI2

  This column indicates the number of primary keys contained in the QQIDXD field. For the most benefit, build an index over the primary key fields.

- QQRID

  We only focus on row types 3000, 3001, and 3002, which contain query optimizer index suggestions.

- QQI2

  This column indicates the importance of a statistic.

- QVC1000

  This column contains the statistics identifier.

This query attempts to correlate the SQE advised statistics to the query optimizer index suggestions. The idea is that we should attempt to build indexes for cases where an index can be used by the query optimizer for the data access method.

If an index is solely used for statistical purposes, the advantage should be given to SQE statistics due to their low maintenance overhead. An exception to this recommendation is if statistics must be current at all times. The only way to accomplish this is by having an index set with the *IMMED maintenance attribute.

6.3.14 Rows with retrieved or fetched details

Specifying *DETAIL in the TYPE parameter of the STRDBMON command indicates that detail rows, as well as summary rows, must be collected for fetch operations. The same is true for detailed the SQL Performance Monitor in iSeries Navigator.

The purpose of detailed 3019 row is for tuning non-SQL queries, those which do not generate a QQ1000 row, such as OPNQRYF. For non-SQL queries, the only way to determine the number of rows that are returned and the total time to return those rows is to collect detail rows. While the detail row contains valuable information, it creates a slight performance degradation for each block of rows returned. Therefore you must closely monitor its use. You can use a detailed row for SQL analysis since the information it contains is also valuable in the SQL environment.

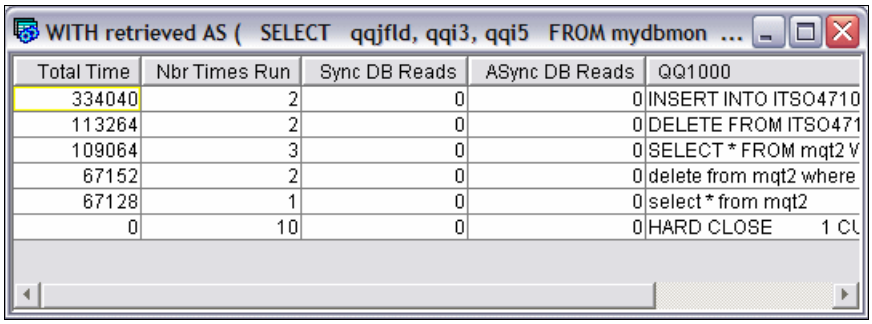
A large number of physical I/O operations can indicate that a larger pool is necessary or that SETOBJACC might be used to bring some of the data into main memory beforehand.

The query in Example 6-32 shows the most time consuming SQL statements.

Example 6-32 Most time consuming SQL statements

```
WITH retrieved AS (
  SELECT   qqjfld, qqi3, qqi5
  FROM mydbmon
  WHERE    qqrid=3019)
SELECT   SUM(qqi6) "Total Time" , COUNT(*) "Nbr Times Run",
         SUM(b.qqi3) "Sync DB Reads", SUM(b.qqi5) "ASync DB Reads", qq1000
FROM mydbmon a, retrieved b
WHERE    a.qqjfld=b.qqjfld AND qqrid=1000 AND qqucnt<>0 AND qqc21<>'MT'
GROUP BY qq1000 ORDER BY 1 DESC;
```

Figure 6-28 shows the output of the query in Example 6-32.



Total Time	Nbr Times Run	Sync DB Reads	ASync DB Reads	QQ1000
334040	2	0	0	INSERT INTO ITSO4710
113264	2	0	0	DELETE FROM ITSO471
109064	3	0	0	SELECT * FROM mqt2 V
67152	2	0	0	delete from mqt2 where
67128	1	0	0	select * from mqt2
0	10	0	0	HARD CLOSE 1 CU

Figure 6-28 Most time consuming SQL statements

This query uses the following columns:

- QQRID

  In the common table expression retrieved, we select only 3019 rows, getting a subset of data with detailed row information.

- QQJFLD

  This column indicates the join column (unique per job).

- QQI3

  This column indicates the number of synchronous database reads. We present a cumulative count for each SQL statement.

- QQI5

  This column indicates the number of asynchronous database reads. We present a cumulative count for each SQL statement.

- QQI6 from row type 1000

  This column indicates the cumulative elapse time in microseconds for each individual query. Since we use it as a cost indicator, we ordered the output in descending order based on this value.

- QQ1000

  This column contains the SQL statement.

- QQUCNT

  This column contains the unique statement identifier. We use this value to exclude nonunique statement identifiers.

- QQC21

  This column contains the SQL request operation identifier. We use this value to exclude continuation records from our analysis. Continuation records are used to display statement text for statements that cannot fit into the single QQ1000 field.

This query gives us detailed information about the amount of reads that the longest running SQL statements have performed.

Row 3019 contains other interesting statistics. We view them by running the query shown in Example 6-33.

Example 6-33 Row 3019 statistics

```
WITH retrieved AS (    SELECT   qqjfld,qqi1,qqi2,qqi3,qqi5, qqi4,qqi6,qqi7,qqi8    FROM
mydbmon WHERE    qqrid=3019) SELECT   SUM(a.qqi6) "Total Time" , COUNT(*) "Nbr Times Run",
         SUM(b.qqi1) "CPU time in milliseconds",
         SUM(b.qqi2) "Clock Time in milliseconds", SUM(b.qqi3) "Sync DB Reads", SUM(b.qqi5)
"Async DB Reads",
         SUM(b.qqi4) "Sync DB Writes", SUM(b.qqi6) "Async DB Writes",          SUM(b.qqi7)
"Number of rows returned",
        SUM(b.qqi8) "Nbr of calls to get rows", qq1000 FROM mydbmon a, retrieved b WHERE
a.qqjfld=b.qqjfld AND qqrid=1000 AND qqucnt<>0 AND qqc21<>'MT' GROUP BY qq1000
ORDER BY "Total Time" DESC;
```

Figure 6-29 shows the output of the query in Example 6-33 (we show two windows).

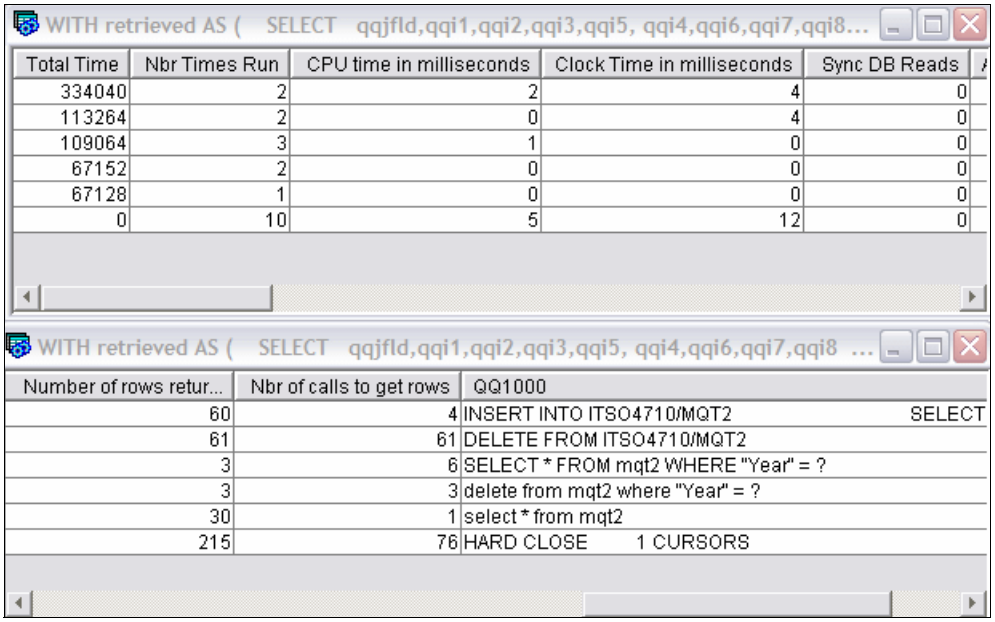


Figure 6-29 Row 3019 statistics

This query uses the following columns:

Note: In the following columns, we present cumulative values for each SQL statement that generated the 3019 record. A statement is deemed unique as long as it can fit in the single QQ1000 record.

- QQI1

  This column indicates the CPU time in milliseconds.

- QQI2

  This column indicates the clock time in milliseconds.

- QQI4

  This column indicates the number of synchronous database writes.

- QQI6

  This column indicates the number of asynchronous database writes.

- QQI7

  This column indicates the number of rows returned.

- QQI8

  This column indicates the number of calls to retrieve rows returned.

You can use QQI2 and QQI7 to calculate your row retrieval costs and then decide if using SETOBJACC to bring the data to the memory pool beforehand will benefit the query in question.

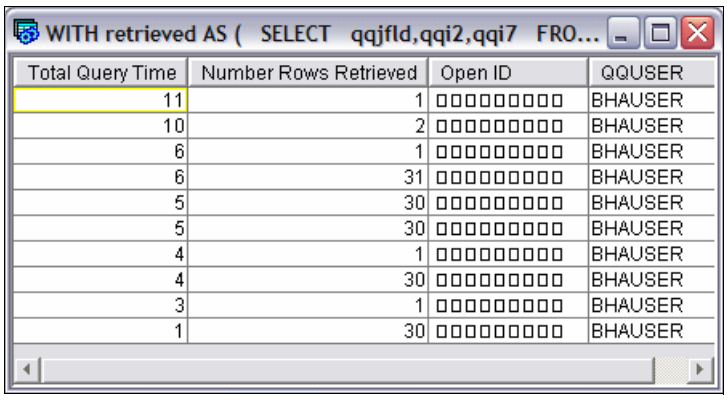
As mentioned earlier, row type 3019 is most useful when analyzing non-SQL queries, such as OPNQRYF.

Now we look at the query in Example 6-34 that lists these queries and orders them based on the elapsed time.

Example 6-34 Non-SQL queries

```
WITH retrieved AS (
  SELECT   qqjfld,qqi2,qqi7
  FROM mydbmon
  WHERE    qqrid=3019 )
SELECT   (qqi1+b.qqi2) "Total Query Time", b.qqi7 "Number Rows Retrieved",
         qqc101 "Open ID", qquser
FROM mydbmon a, retrieved b
WHERE a.qqjfld=b.qqjfld AND qqrid=3014
ORDER BY 1 DESC;
```

Figure 6-30 shows the output of the query in Example 6-34.



Total Query Time	Number Rows Retrieved	Open ID	QQUSER
11	1	□□□□□□□□□□	BHAUSER
10	2	□□□□□□□□□□	BHAUSER
6	1	□□□□□□□□□□	BHAUSER
6	31	□□□□□□□□□□	BHAUSER
5	30	□□□□□□□□□□	BHAUSER
5	30	□□□□□□□□□□	BHAUSER
4	1	□□□□□□□□□□	BHAUSER
4	30	□□□□□□□□□□	BHAUSER
3	1	□□□□□□□□□□	BHAUSER
1	30	□□□□□□□□□□	BHAUSER

Figure 6-30 Non-SQL queries

This query uses the following columns:

- QQI1 from row type 3014

  This column indicates the time spent to open a cursor in milliseconds. It is added to the fetch elapsed time and projected as the Total Query Time field.

  The value in this column is our costing indicator. Any optimization efforts on non-SQL queries should be focused on the most costly queries.

- QQC101

  This column indicates the query open identifier. Non-SQL interfaces require job scooped uniquely named identifiers at the open time. This identifier is contained in QQC101 field of the row type 3014.

- QQUSER

  This column indicates the job user name. Since most non-SQL interfaces run under the real profile rather than the QUSER type profile, the QQUSER field is adequate for user identification.

The intent of the query that we outlined here shows the most time consuming non-SQL queries. Since there is no explicit flag in the Database Monitor data that differentiates between SQL and non-SQL-based queries, we modify the query to try and exclude the open IDs that are most likely SQL related as shown in Example 6-35.

Example 6-35 Non-SQL generated requests only

```
WITH qqq1000 AS (SELECT qqjfld,qqucnt FROM mydbmon
  WHERE    qqrid=1000 AND qqucnt<>0 AND qqc21<>'MT'), retrieved AS (   SELECT
a.qqjfld,a.qqucnt,qqi2,qqi7   FROM mydbmon a EXCEPTION JOIN qqq1000 b ON
           (a.qqjfld = b.qqjfld and a.qqucnt = b.qqucnt)   WHERE    qqrid=3019) SELECT
(qqi1+b.qqi2) "Total Query Time", b.qqi7 "Number Rows Retrieved",
         qqc101 "Open ID", qquser "Job User", qqjob "Job Name",
         qqjnum "Job Number" FROM mydbmon a INNER JOIN retrieved b ON
         (a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt)
WHERE    qqrid=3014 AND SUBSTR(qqc101,1,1) NOT IN (' ','*',x'00')
ORDER BY "Total Query Time" DESC;
```

Figure 6-31 shows the output of the query in Example 6-35.

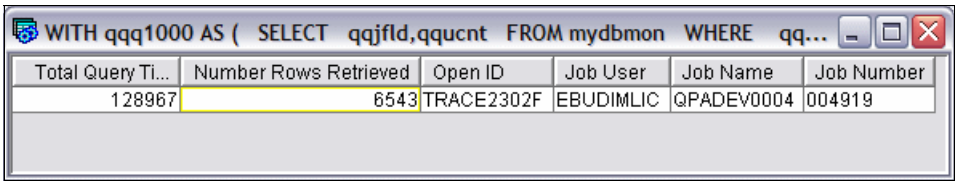


Figure 6-31 Non-SQL generated requests only

This query uses the following columns:

- QQJOB

  This column indicates the job name.

- QQJNUM

  This column indicates the job number.

SQL-related rows generate record 1000, and non-SQL ones do not. We have taken advantage of this fact to exclude SQL related rows from this query. Based on the start and end time of Database Monitor collection, it is still possible for some SQL-related queries to be included in our result set so we filter it further by checking the open ID's starting character for validity.

Even with this additional check, it is possible for some SQL-related rows to appear in our result set. You might need to further customize the WHERE clause in the final select statement.

6.3.15 Materialized query tables

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

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

Support for creating and maintaining MQTs was shipped with the base V5R3 release of i5/OS. The query optimizer support for recognizing and using MQTs is available with V5R3 i5/OS PTF SI17164 and the latest database group PTF SF99503 level 4.

For more information about MQTs, see the white paper *The creation and use of materialized query tables within IBM DB2 UDB for iSeries*, which is available from the DB2 Universal Database for iSeries Web site at:

http://www.ibm.com/iSeries/DB2

For the query shown here, you first build the MQT using SQL as shown in Example 6-36. For this example, the employee table has roughly 2 million rows and the department table has five rows.

Example 6-36 Creating the MQT

```
CREATE TABLE MQT1
         AS (SELECT D.deptname, D.location, E.firstname, 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
```

Then you run the query shown in Example 6-37, which returns information about employees whose job is DESIGNER.

Example 6-37 Running SQL using MQT

```
SELECT D.deptname, D.location, E.firstname, 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'
```

In this query, the MQT matches part of the user's query. The optimizer places the MQT in the FROM clause and replaces the DEPARTMENT and EMPLOYEE tables. Any remaining selection that is not done by the MQT query (M.job= 'DESIGNER') is done to remove the extra rows. Then the result expression, M.salary+M.comm+M.bonus, is calculated.

Figure 6-32 shows the Visual Explain diagram of the query when using the MQT.

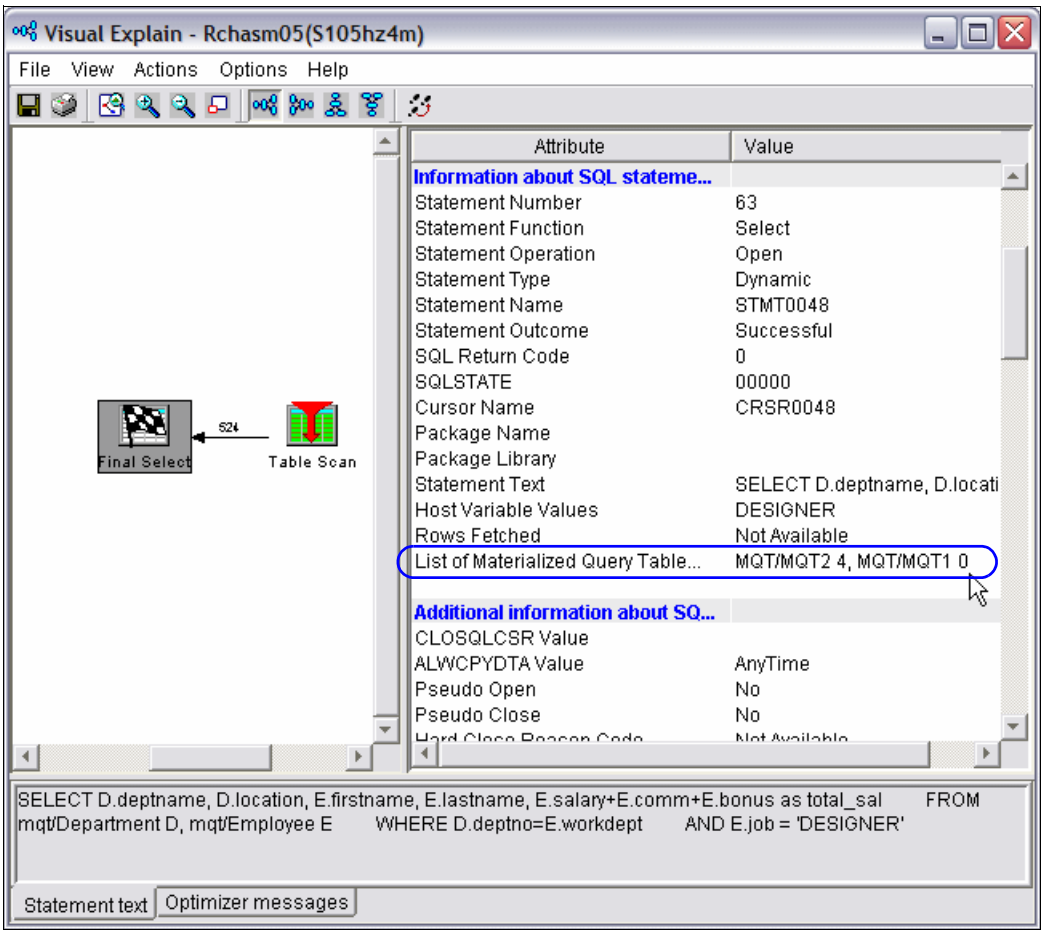


Figure 6-32 Visual Explain diagram for the MQT request

Notice the List of Materialized Query Tables in Figure 6-32. There are two tables, MQT2 and MQT1. MQT2 is followed by a value of 4, and MQT1 is followed by a value of 0. The value of 4 means that the grouping specified in the MQT is not compatible with the grouping specified in the query. The value of 0 means that the MQT was used in the query. These values are retrieved from the qq1000 field of the 3030 record in SQL Performance Monitor data.

For a listing of all the values for QQ1000 field of the 3030 record, see *DB2 Universal Database for iSeries Database Performance and Query Optimization*, which is available in the iSeries Information Center on the Web at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzajq/rzajqmst.htm`

There are a number of ways to check whether an MQT is being used. Example 6-38 shows a query that you can run to view the reasons why an MQT is not being used. You can also check whether MQTs are replacing the existing table names in the query. You can check the QQC13 column of either the 3000, 3001, or 3002 record of the SQL Performance Monitor data. The 3000 record is for use when a table scan is done. The 3001 record is for use when an index is used, and the 3002 record is for when an index is created.

To look at the MQTs on your system and see if they are being used, run the following Display File Description (DSPFD) command and SQL Performance Monitor data to check. First you must determine which tables are MQTs. Create a file with the following command:

```
DSPFD FILE(library or *ALLUSR/*ALL) TYPE(*ATR) OUTPUT(*OUTFILE) FILEATR(*PF)
OUTFILE(library/MQTS)
```

Then to see which MQTs are being used or are not in queries, run the query shown in Example 6-38 against the file created from the DSPFD command, using SQL performance data.

Example 6-38 Query to find MQTs

```
with MQTFILES as (select phfile, phlib from library/MQTS
                                        where phsqlt = 'M')
select count(*) as Times_Used,
a.phfile as  MQT_file,
a.phlib as MQT_library,
b.qqc13 as Used_in_Query
from MQTFILES a left outer join library/dbmondata b
on a.phfile = b.qqtfn and a.phlib = b.qqtln
group by a.phfile, a.phlib, b.qqc13
order by b.qqc13
```

Next, to determine why MQTs are not being used, run the query shown in Example 6-39.

Example 6-39 Reason why MQTs are not used

```
SELECT substr(a.qq1000,1,100) AS MQT_RC,b.qq1000
FROM library/dbmondata a,
library/dbmondata b where a.qqrid = 3030 and b.qqrid = 1000 and a.qqjfld
= b.qqjfld and a.qq1000 <> 'O' and b.qqc11 in ('S','U','I','D') and
b.qqc21 NOT IN ('MT', 'CL', 'HC', 'FE')
GROUP BY B.QQ1000, A.QQ1000
```

7

Using Collection Services data to identify jobs using system resources

This chapter explains how to find jobs using CPU or I/O with Collection Services data and how you can integrate it with the Database Monitor data. This chapter begins by describes how to start Collection Services. Then it guides you in using Collection Services data to find jobs using CPU, to find jobs with high disk I/O counts, and to use it in conjunction with the SQL Performance Monitors.

7.1 Collection Services and Database Monitor data

Collection Services allows you to gather performance data with little or no observable impact on system performance. Its data is analyzed using the IBM Performance Tools for iSeries licensed program (5722PT1) or other performance report applications, iSeries Navigator monitors, and the graph history function. If you prefer to view real-time performance data, system monitors provide an easy-to-use graphical interface for monitoring system performance. For more information about iSeries Navigator Monitors, see “iSeries Navigator monitors” in *iSeries Performance Version 5 Release 3* on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/rzahx/rzahx.pdf

Collection Services collects data that identifies the relative amount of system resource used by different areas of your system. While you are required to use Performance Tools for iSeries, it is not required to gather the data.

In most of the analysis when looking for job-level information in this chapter, we use the Component Report. The Component Report uses Collection Services data to provide information about the same components of system performance as a System Report, but at a greater level of detail. The Component Report helps you find the jobs that are consuming high amounts of system resources, such as CPU, disk, and so on.

Note: While running Collection Services, you should also run Database Monitor on all the jobs. After we identify the job information of the job or jobs that consume high amounts of system resources, such as CPU, disk and so on, we can the query the Database Monitor data to find the jobs as explained in Chapter 6, “Querying the performance data of the Database Monitor” on page 133.

7.1.1 Starting Collection Services

To analyze Collection Services data, you must first start Collection Services, which you can do in one of the following ways:

- From the PERFORM menu, using GO PERFORM
- From iSeries Navigator
- Using Performance Management APIs
- In V5R3, using the Start Performance Collection (STRPFRCOL) CL command

Note: Starting Collection Services from the PERFORM menu or from iSeries Navigator requires that you install Performance Tools for iSeries, 5722PT1.

In the following sections, we explain the details for each of the methods to start Collection Services to gather data.

PERFORM menu

To start Collection Services, from the PEFORM menu:

1. Enter the following command:

   ```
   GO PERFORM
   ```

2. On the IBM Performance Tools for iSeries display, type option `2` (Collect performance data).
3. On the Collect Performance Data display, select option `2` (Configure Performance Collection).
4. The Configure Perf Collection (CFGPFRCOL) display is shown. In setting the configuration, you should consider your system resources, for example, whether you have enough storage to handle the data being gathered. You must consider the following two parameters:
   - *Default interval*: This parameter indicates the sample interval time in minutes to collect data.
   - *Collection retention period:* This parameter indicates how long the management collection object (*MGTCOL) should be retained on the system.

   Set these parameters based on your system resources and the problem that you are trying to capture. For example, if you have a performance problem that occurs intermittently over a week, you might set the interval time to 15 minutes and the collection retention period to 7 days or longer if you want comparison data, but only if you have the resources to maintain the size of the *MGTCOL objects.

   After you set the configuration, press Enter.
5. Type option `1` (Start Performance Collection). Specify `*CFG` for the collection profile.

For more information, see *iSeries Performance Version 5 Release 3,* which you can find on the Web at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/rzahx/rzahx.pdf`

7.1.2 From iSeries Navigator

In iSeries Navigator, select **your system name** → **Configuration and Service**. Right-click **Collection Services** and select **Start Performance Collection**. In the Start Collection Services window (Figure 7-1), verify the configuration settings and click **OK** to start Collection Services.

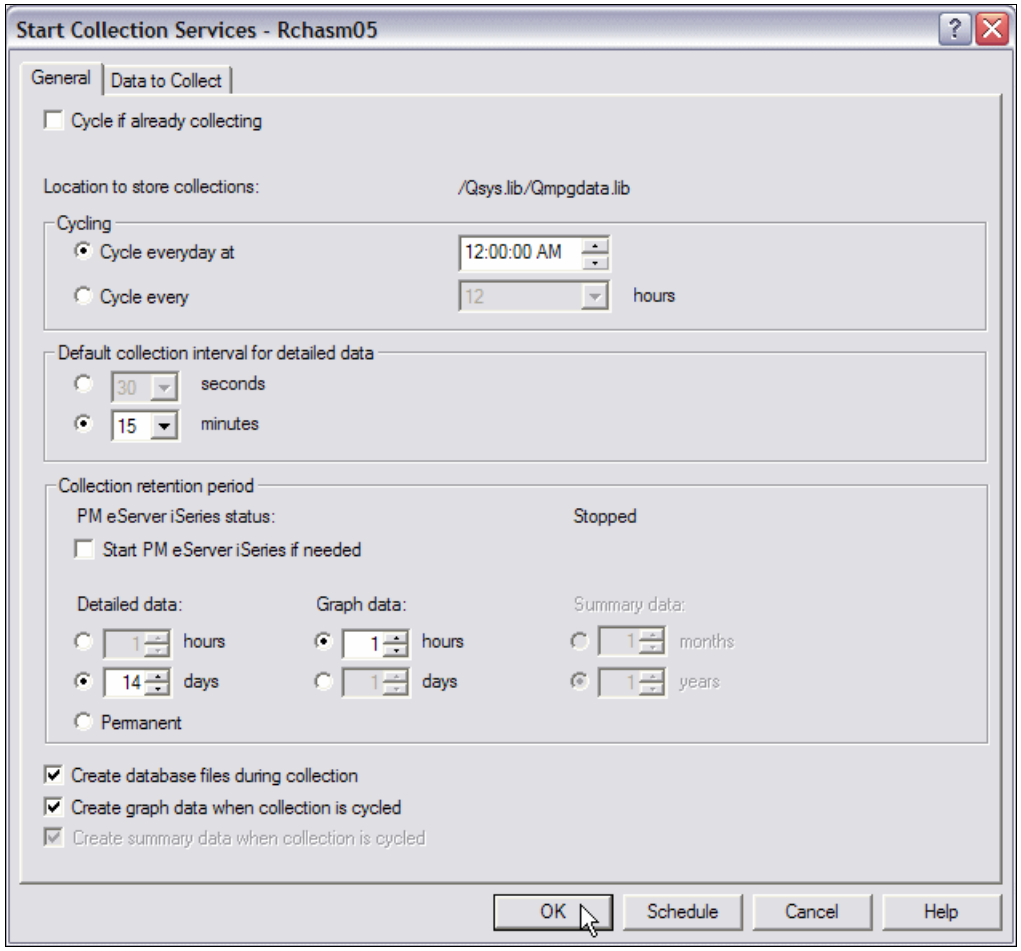


Figure 7-1 Configuring and starting Collection Services in iSeries Navigator

7.1.3 Using Performance Management APIs

You can use Collector APIs to start collecting performance data. The APIs do not require you to have Performance Tools for iSeries installed. To start Collection Services, you can use the following command:

```
CALL QYPSSTRC PARM('*PFR      ' '*STANDARDP' X'00000000')
```

For more information about the parameters of the API, see “Collector APIs” in *iSeries Performance Management APIs Version 5 Release 3* on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/apis/perfmg.pdf

7.1.4 V5R3 STRPFRCOL command

You can also start V5R3 of Collection Services by using the STRPFRCOL CL command. Then to configure Collection Services, you use the Configure Performance Collection (CFGPFRCOL) CL command.

For more information about the Collection Services CL commands, see *iSeries Performance Version 5 Release 3* on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/rzahx/rzahx.pdf

7.2 Using Collection Services data to find jobs using CPU

You can use Collection Services data to look for jobs using CPU. In this section, we explain how to find jobs that are using CPU by using the following methods:

- Component Report from the PERFORM menu
- iSeries Navigator Graph History
- Management Central System Monitors

Note: When the job information is found, you can use the job information to query the Database Monitor table as explained in Chapter 6, “Querying the performance data of the Database Monitor” on page 133.

7.2.1 Finding jobs using CPU with the Component Report

To use the PERFORM menu of IBM Performance Tools for iSeries to gather the Component Report, you must install Performance Tools, product 5722PT1. To access the PERFORM menu, enter the following command:

```
GO PERFORM
```

If you want to interactively examine the data, select option 7 (Display performance data). If you choose this option, you must keep in mind your system resources. This job runs interactively and uses CPU and I/O resources. Or you can select option 3 (Print performance report), which submit a job to batch.

After you select an option, you then select the member that you want to investigate, based on the date and time shown. If no members are shown and you have started Collection Services, run the Create Performance Data (CRTPFRDTA) command against the *MGTCOL object that contains your data or create the files in iSeries Navigator.

- To create the files using the CRTPFRDTA CL command, find the *MGTCOL object to create the files from by entering the following command:

  ```
  WRKOBJ OBJ(qmpgdata/*ALL) OBJTYPE(*MGTCOL)
  ```

  The attribute of the *MGTCOL object must be *PFR. In the WRKOBJ command shown, you replace *qmpgdata* with the library where you keep your performance data. After the *MGTCOL object is found, you run the following CL command to create the database files:

  ```
  CRTPFRDTA FROMMGTCOL(library/mgtcolname)
  ```

- To create the files using iSeries Navigator, select **your system name** → **Configuration and Service**. Right-click **Collection Services**. Then in the right pane, you see a list of the *MGTCOL objects. Right-click the **collection name**, which is the *MGTCOL object, and select **Create Database Files Now** as shown in Figure 7-2.

  For more information, see *iSeries Performance Version 5 Release 3* on the Web at:

  http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/rzahx/rzahx.pdf

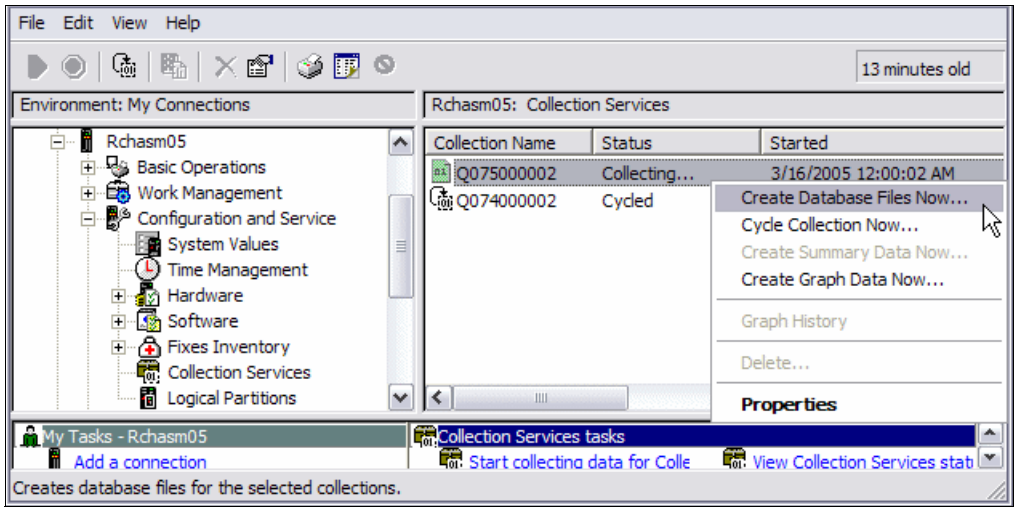


*Figure 7-2 Creating files from *MGTCOL objects in iSeries Navigator*

The following sections explain how to find the jobs that are using CPU by using either the option to print performance report or the option to display performance data.

Using option 3: Print performance report

When you select option 3, you must specify the library in which you keep your performance data. In most cases, the library is QMPGDATA. You can page up and down until you see a time frame that you want. Type 2 next to the member that you want to use to get a Component Report.

In the next Select Sections for Report display (Figure 7-3), you can press F6 to print the entire report to select all the sections to be in your report. Since you must often review all the performance data at the time of high CPU usage, we recommend that you use F6. In the display shown in Figure 7-3, you see that we choose option 1 to select Job Workload Activity so that we can only see the jobs that are using CPU.

```
                          Select Sections for Report

Member . . . . . :   Q075000002

Type options, press Enter.  Press F6 to print entire report.
  1=Select

Option     Section
           Component Interval Activity
  1        Job Workload Activity
           Storage Pool Activity
           Disk Activity
           IOP Utilizations
           Local Work Stations
           Remote Work Stations
           Exception Occurrence
           Data Base Journaling Summary
           TCP/IP Activity
           HTTP Server Activity


                                                                      Bottom
F3=Exit   F6=Print entire report   F12=Cancel
```

Figure 7-3 Select Sections for Report display

In the Select Categories for Report display (Figure 7-4), we type option 1 to select Time interval. Time interval is usually the best option if you don't know any other information. If you want more information, you can press F6 to print the entire report, but it is best to narrow the information down to a time frame in which you are interested.

```
                         Select Categories for Report

Member . . . . . :   Q075000002

Type options, press Enter.  Press F6 to print entire report.
  1=Select

Option     Category
  1        Time interval
           Job
           Job type
           Job run priority
           User ID
           Subsystem
           Pool
           Communications line
           Control unit
           Functional area



                                                                      Bottom
F3=Exit   F6=Print entire report   F12=Cancel
```

Figure 7-4 Select Categories for Report display

If you selected Time interval on the previous display, you see the Select Time Intervals display (Figure 7-5). You can select intervals that you want to look at based on CPU utilization. In this example, we select two time frames of interest by typing 1 in the Opt column. These time frames were chosen because the concern is the sudden CPU growth.

```
                          Select Time Intervals

Library . . . . . :   QMPGDATA      Performance data . . . . . :   Q074000002

Type options, press Enter.
  1=Select
O                                              Int    High        Pool
p                Transaction  -CPU Util--     Feat  --Util--  -Fault/Sec-
t  Date   Time   Count  Resp  Tot Int Bch     Util  Dsk Unit  Mch User ID    Excp
   03/15  07:45      0   .00    0   0   0        0    1 0001    0    0 02      39
   03/15  08:00      0   .00    0   0   0        0   -- ----    0    0 02      39
   03/15  08:15      2   .00    0   0   0        0    1 0003    0    0 02     120
   03/15  08:30      0   .00    0   0   0        0    1 0001    0    0 02     341
   03/15  08:45      8   .00    0   0   0        0    1 0001    0    0 02      96
   03/15  09:00      0   .00    0   0   0        0    1 0002    0    0 02     168
   03/15  09:15      0   .00    0   0   0        0    1 0005    0    0 02     505
   03/15  09:30    104   .24    2   2   0       23    4 0001    0    0 03     163
1  03/15  09:45     86   .59   10  10   0      127    6 0001    0    0 03      55
1  03/15  10:00    374   .14   17   9   8      124   11 0001    0    0 03    1710
                                                                        More...

 F3=Exit                    F5=Refresh                  F12=Cancel
 F13=Sort (date/time)       F14=Sort (count)            F24=More keys
```

Figure 7-5 Select Time Intervals display

In the Specify Report Options display (Figure 7-6), you specify any report title that you want. In this example, we specify CPU Report. You press Enter, and a job is submitted to batch.

```
                          Specify Report Options

Type choices, press Enter.

Report title . . . .   CPU Report

Report detail  . . .   *JOB                   *JOB, *THREAD

Job description  . .   QPFRJOBD               Name, *NONE
  Library  . . . . .     *LIBL                Name, *LIBL, *CURLIB

F3=Exit   F12=Cancel
```

Figure 7-6 Specify Report Options panel

You then return to the Print Performance Report - Sample data display and see the following message at the bottom of the display:

```
Job 003675/PCHIDESTER/PRTCPTRPT submitted to job queue QBATCH in library ...
```

This message gives you the submitted job for your report. You can find your report by entering the following command:

```
WRKSBMJOB *JOB
```

In the Work with Submitted Jobs display, your submitted job is called PRTCPTRPT. When the PRTCPTRPT job is in OUTQ status, select option 8 to view the spool file. The report is in the spool file QPPTCPTR. Type 5 on the line that precedes the QPPTCPTR file.

In Figure 7-7, you see part of the Component Report showing the jobs that are using CPU. In this example, there are three jobs of concern as highlighted in bold.

```
File  . . . . . :   QPPTCPTR
Control . . . . .
Find  . . . . . .
*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+....0....
Virtual Processors:   2    Processor Units :   2.0
                            T     P          DB
   Job     User Name/ Job    y     t    CPU   Cpb             Tns            -------- Disk I/O --------
   Name    Thread     Number p  Pl y    Util  Util   Tns      /Hour    Rsp     Sync    Async   Logical
---------- ---------- ------ -- -- -- ----- ----- ----- --------- ------- -------- -------- -------- -
QTVDEVICE  QTCP       003297 B  02 20   .00    .0 0         0    .000        0        0        0
QTVDEVICE  QTCP       003298 B  02 20   .00    .0       0         0    .000        0        0        0
QTVDEVICE  QTCP       003299 B  02 20   .00    .0       0         0    .000       60       40        0
QTVDEVICE  QTCP       003300 B  02 20   .00    .0       0         0    .000        0        0        0
QTVDEVICE  QTCP       003301 B  02 20   .00    .0       0         0    .000        0        0        0
QTVDEVICE  QTCP       003302 B  02 20   .00    .0       0         0    .000        0        0        0
QUERY      PCHIDESTER 003460 B  02 50  6.64   1.7       0         0    .000     5602    32789     5161
QUERY2     PCHIDESTER 003461 B  02 50  7.72   2.0       0         0    .000     5573    24476     5161
QUERY3     PCHIDESTER 003462 B  02 50  7.82   1.9       0         0    .000     5529    22520     5161
QUSRWRK    QSYS       003241 M  02 00   .00    .0       0         0    .000        0        0        0
QYPSJSVR   QYPSJSVR   003314 B  02 16   .09    .0       0         0    .000        9        0        0
QYPSPFRCOL QSYS       003354 B  02 01   .00    .0       0         0    .000        0       39        0
QYUSCMCRMD QSYS       003312 B  02 50   .00    .0       0         0    .000        1        0        0
Q1PSCH     QPM400     003281 B  02 50   .00    .0       0         0    .000        0        0      180

F3=Exit   F12=Cancel   F19=Left   F20=Right   F24=More keys
```

Figure 7-7 Partial print of Jobs using CPU

The disadvantage of using the printed report versus displaying the data interactively is that the printed Component Report is sorted by the job name, not by CPU. Displaying the data interactively allows you to sort on CPU.

Now that we have the jobs that are using the majority of CPU, we can now look at the Database Monitor data that was running at the same time to investigate what the jobs were doing. Chapter 6, “Querying the performance data of the Database Monitor” on page 133, gives examples on how to investigate the SQL, if any, that the jobs were running based on the job name, job user, and job number.

Option 7: Display performance data

If you select option 7, you must specify the library in which you keep your performance data. In most cases, the library is QMPGDATA. Next you select the member that you want to investigate, based on the date and time shown. You do this by typing 1 next to the member.

After you select the member, you can page up and down until you see a time frame you are looking for or until you see a CPU Utilization that concerns you. Then you type 1 next to the interval or intervals that you want to examine.

After you select the intervals, you see the Display Performance Data display (Figure 7-8).

```
                          Display Performance Data

Member . . . . . .   Q074000002            F4 for list
  Library  . . . . .    QMPGDATA

Elapsed time . . . :   00:15:00          Version  . . . . . :   5
System . . . . . :   RCHASM05            Release  . . . . . :   3.0
Start date . . . . :   03/15/05          Model  . . . . . . :   270
Start time . . . . :   00:00:02          Serial number  . . :   10-5HZ4M
Partition ID . . . :   000               Feature Code . . . :   23F5-2434-1520
QPFRADJ  . . . . . :   0                 Int Threshold  . . :   0.51 %
QDYNPTYSCD . . . . :   1                 Virtual Processors :   2
QDYNPTYADJ . . . . :   1                 Processor Units  . :   2.00

CPU utilization (interactive)  . . . . . :         10.10
CPU utilization (other)  . . . . . . . . . :           .54
Interactive Feature Utilization  . . . . . :        127.48
Time exceeding Int CPU Threshold (in seconds)  . :   85882
Job count  . . . . . . . . . . . . . . . . . :          22
Transaction count  . . . . . . . . . . . . . :          86
                                                                    More...
F3=Exit      F4=Prompt   F5=Refresh   F6=Display all jobs   F10=Command entry
F12=Cancel   F24=More keys
```

Figure 7-8 Display Performance Data display

From this display, press F6 to view all jobs. Then you see a listing of the jobs that were running during the interval as shown in Figure 7-9. You can use F19 to sort by CPU.

```
                                Display Jobs

Elapsed time  . . :   00:15:00          Member  . . . . . :   Y074000002
                                          Library . . . . :     QMPGDATA

Type options, press Enter.
  5=Display job detail   6=Wait detail

                                          Job      CPU      Tns     Avg    Disk
Option  Job         User        Number    Type    Util     Count    Rsp     I/O
        QPADEV0001  PCHIDESTER  003456    PTH     6.62         0     .0   69347
        QPADEV0003  PCHIDESTER  003457    PTH     3.30         0     .0   40359
        CFINT01                           LIC      .31         0     .0       0
        QPADEV0005  PCHIDESTER  003458    PTH      .13        20    1.6    2188
        QYPSJSVR    QYPSJSVR    003314    BCH      .10         0     .0       5
        CFINT02                           LIC      .09         0     .0       0
        QPADEV0004  PCHIDESTER  003459    PTH      .04        66     .2     500
        QINTER      QSYS        003247    SBS      .00         0     .0      14
        CRTPFRDTA   QSYS        003425    BCH      .00         0     .0     194
        QYPSPFRCOL  QSYS        003354    BCH      .00         0     .0      20
                                                                    More...
F3=Exit           F12=Cancel   F15=Sort by job   F16=Sort by job type
F19=Sort by CPU   F24=More keys
```

Figure 7-9 Listing of jobs using CPU

In this example, it appears as though there are two jobs that are using the majority of CPU. If this is unusual and Database Monitor data was gathered during this interval, you can run a query based on the job name, job user, and job number as shown in Chapter 6, “Querying the performance data of the Database Monitor” on page 133. The query helps to isolate the SQL, if any, that these jobs were running during this interval.

7.2.2 Finding jobs using CPU with iSeries Navigator Graph History

Graph history provides a graphical view of performance data collected over days, weeks, months, or years with Collection Services. You do not need to have a system monitor running to view performance data. As long as you use Collection Services to collect data, you can view the Graph History window. To access the graph history in iSeries Navigator, select the **system name** → **Configuration and Service**. Right-click **Collection Services** and select **Graph History** as shown in Figure 7-10.

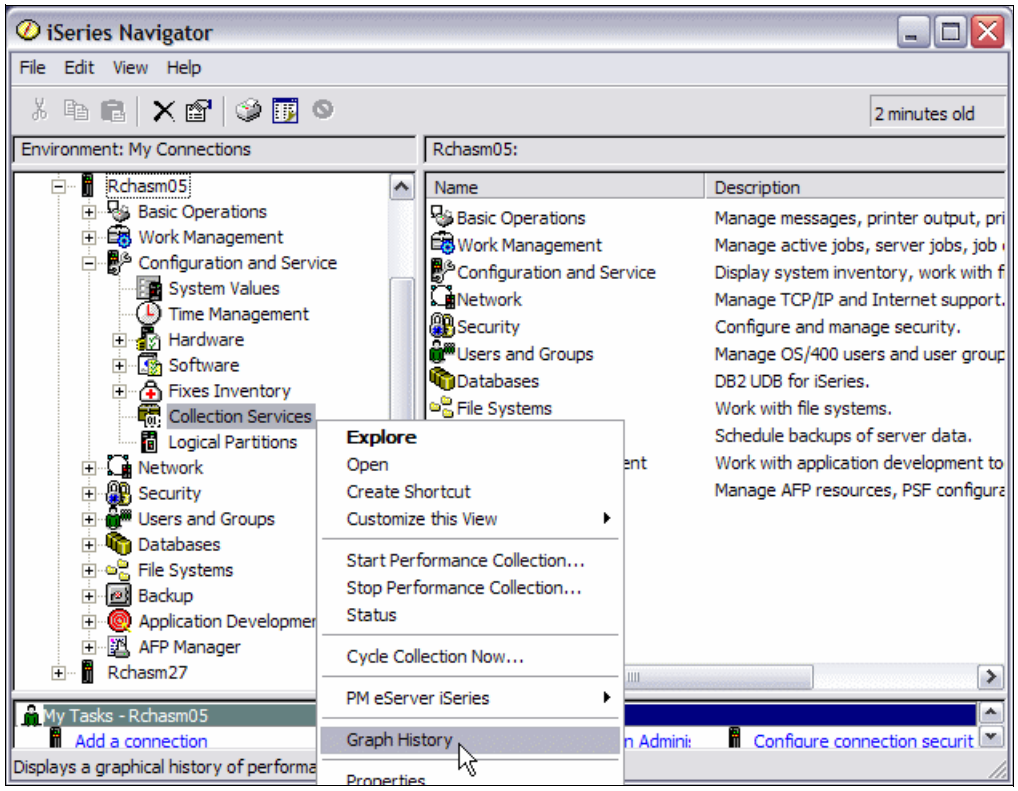


Figure 7-10 Selecting Graph History

Then the Graph History window opens. When you click the drop-down list for Metric, you see several options to help you find the jobs using CPU as shown in Figure 7-11. In this example, we select the option **CPU Utilization (Average)**. You also need to specify the time frame for which you want to see graph history.

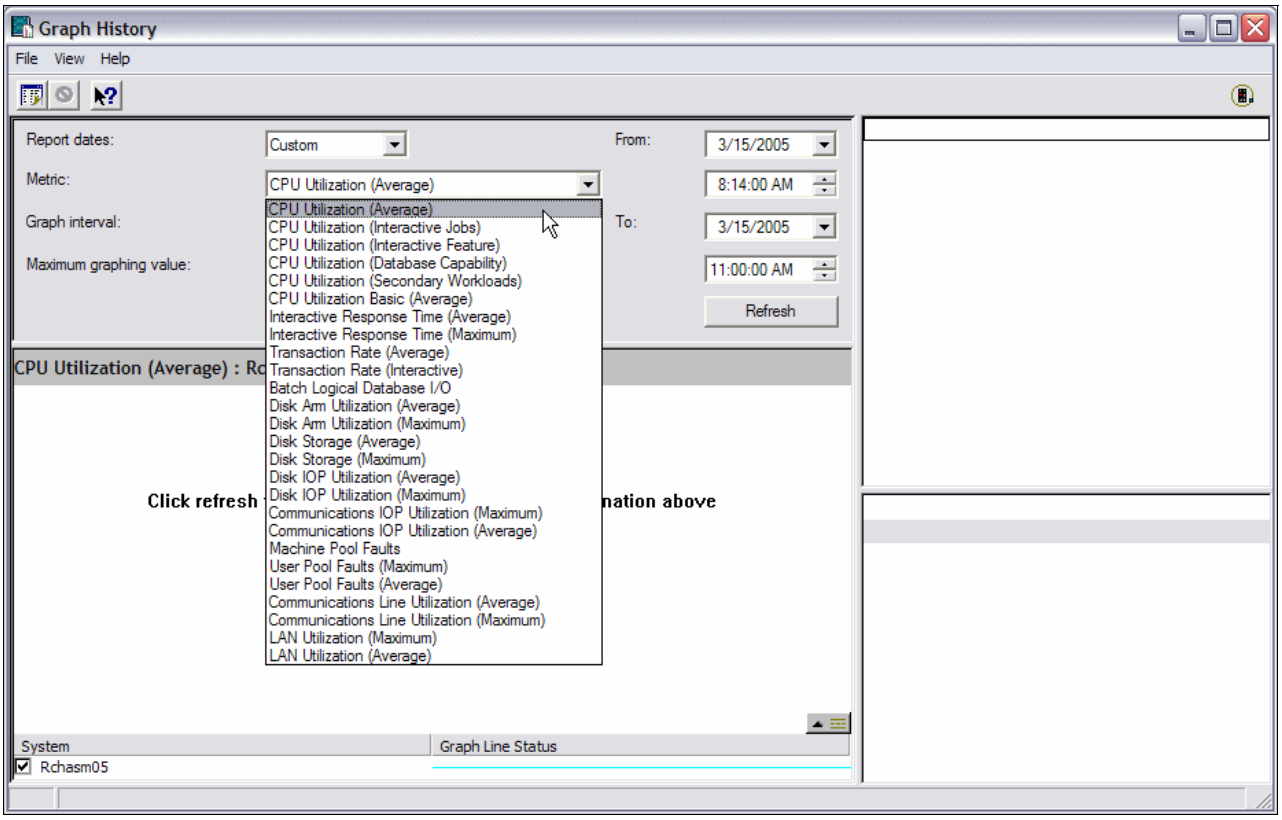


Figure 7-11 Graph History metric options

After you launch a graph history, a window opens that shows a series of graphed collection points. These collection points on the graph line are identified by three different graphics that correspond to the three levels of data that are available:

- A *square collection point* represents data that includes both the detailed information and properties information.
- A *triangular collection point* represents summarized data that contains detailed information.
- A *circular collection point* represents data that contains no detailed information or properties information.

Figure 7-12 shows information after a point on the graph was selected to show the graph in the upper right corner giving the job names. You can select any of the bars on the graph to view information about the job in the box underneath it. In the example shown, the bar turns black after being selected, and the first three lines of information in the box underneath the graph are the job name, user name, and job number. This information is required to run the queries in Chapter 6, “Querying the performance data of the Database Monitor” on page 133, to indicate whether the jobs were running SQL, and if so, the SQL statement that was run.

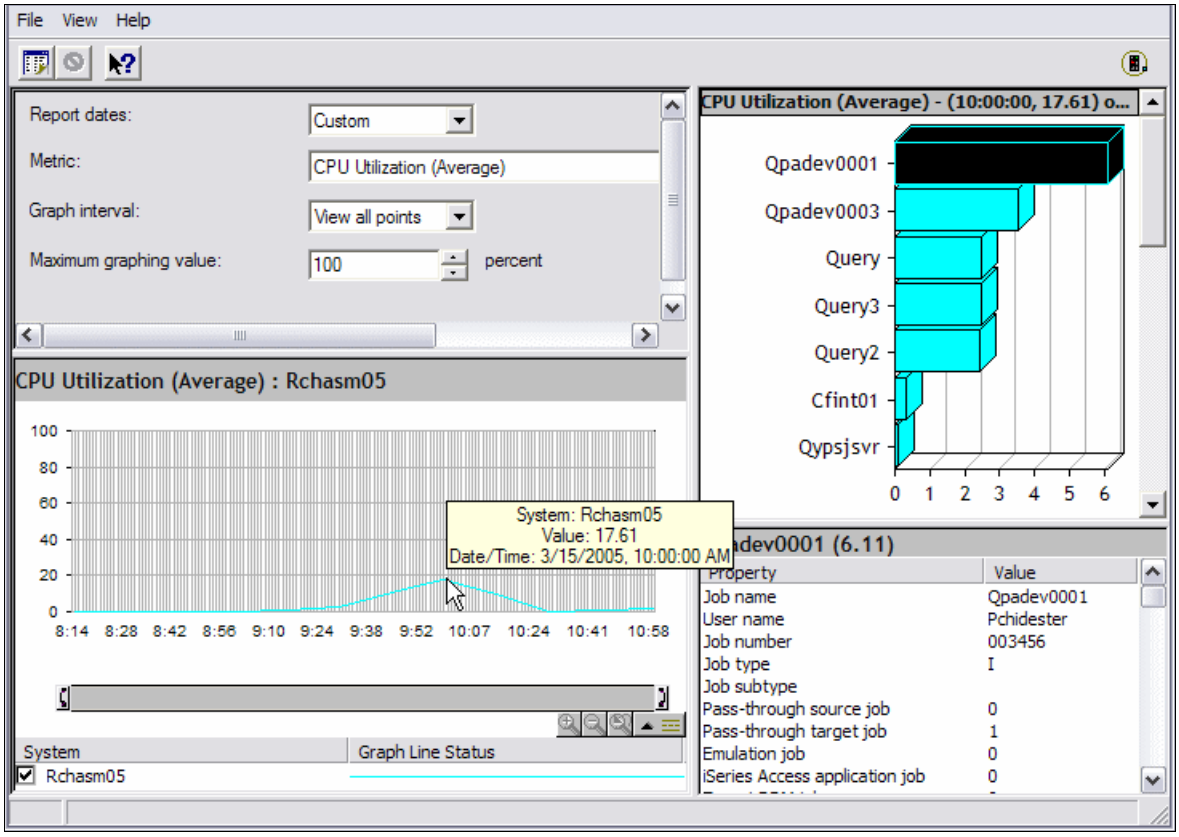


Figure 7-12 Displaying job using CPU with Graph History data

For more information about using Graph History see *iSeries Performance Version 5 Release 3* on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/rzahx/rzahx.pdf

7.2.3 Finding jobs using CPU with Management Central System Monitors

The system monitors display the data stored in the collection objects that are generated and maintained by Collection Services. The system monitors display data as it is collected, for up to one hour. To view longer periods of data, use Graph history as explained in 7.2.2, “Finding jobs using CPU with iSeries Navigator Graph History” on page 189. You can change the frequency of the data collection in the monitor properties, which overrides the settings in Collection Services.

To set up a system monitor, you must define the metrics that you want to track and the action that the monitor should take when the metrics reach the specified levels. To define a system monitor that looks for jobs using CPU, complete the following steps:

1. In iSeries Navigator, expand **Management Central** → **Monitors**. Right-click **System Monitor** and select **New Monitor**.
2. On the **General** page, enter a name and description for the monitor.
3. On the **Metrics** page, and enter the following values:
   a. From the list of Available Metrics, select **CPU Utilization (Average)** and click **Add**. CPU Utilization Basic (Average) is now listed under Metrics to monitor, and the bottom portion of the window shows the settings for this metric.

b. Add two other Available Metrics: CPU Utilization (Interactive Feature) and CPU Utilization (Database Capability).

c. For Collection interval, specify how often you want to collect this data. This value overrides the Collection Services setting. For this example, we specify 30 seconds.

d. To change the scale for the vertical axis of the monitor's graph for this metric, change the Maximum graphing value. To change the scale for the horizontal axis of the graph for this metric, change the value for Display time.

4. On the Systems and Groups page, select your systems and groups. You can click Browse to select the endpoint system on which you want to run the monitor.

5. Expand **Management Central** → **Monitors** and click **System**. You should now see the monitor displayed with a status of *Stopped*.

6. To start the monitor, right-click it and select **Start**.

7. To view the graph data, double-click the monitor that you created. Click any square in the graph to see a bar graph in the upper right corner that shows the job names sorted by CPU. To find more job details, click any of the bars on the graph as shown in Figure 7-13.

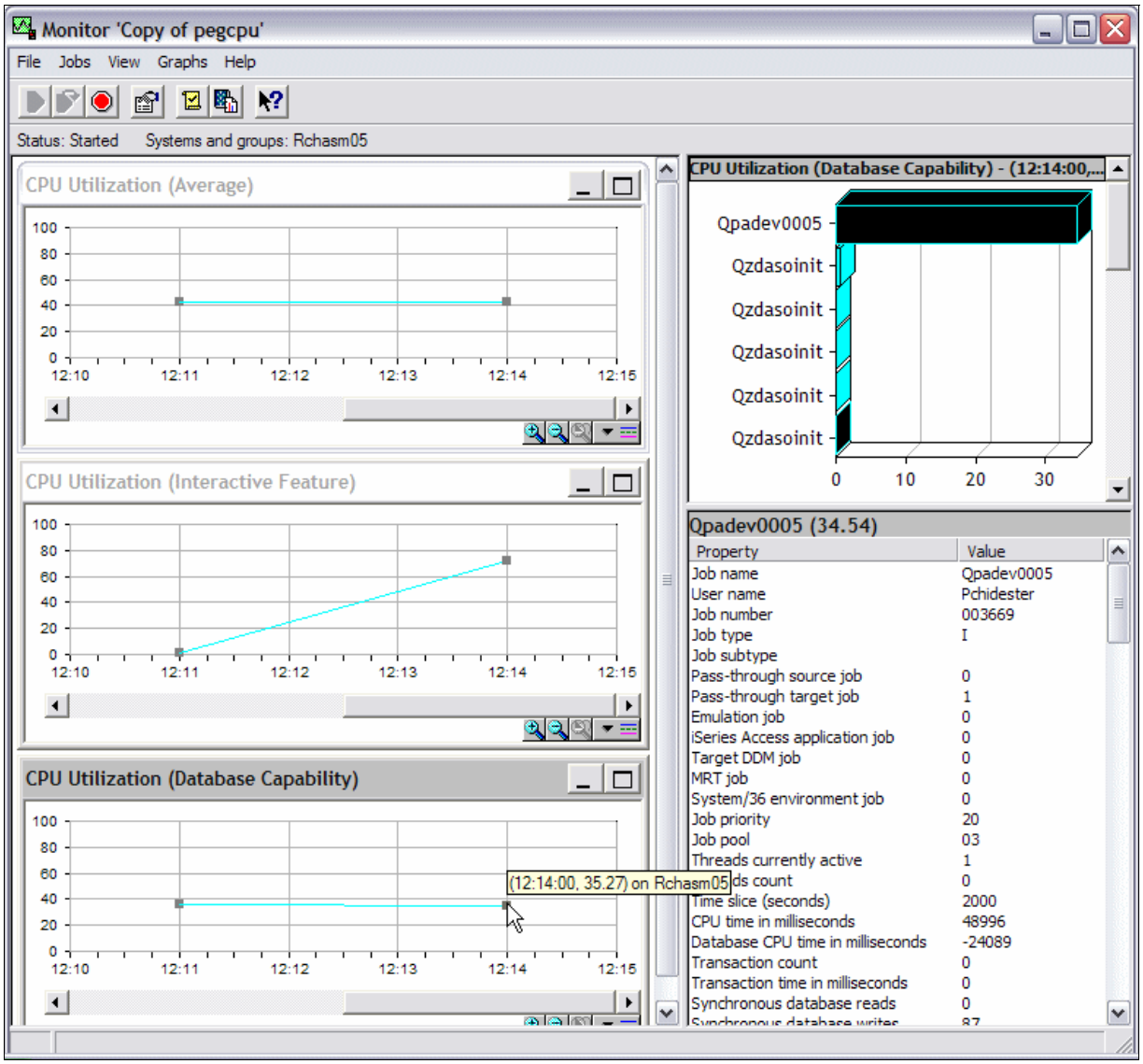


Figure 7-13 System Monitor showing job using CPU

The box below the graph shows the fully qualified job name, consisting of job name, user name, and job number. You can use this information to query a Database Monitor file collecting during this time frame as shown in Chapter 6, “Querying the performance data of the Database Monitor” on page 133.

For more information about using Graph History see *iSeries Performance Version 5 Release 3* on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/rzahx/rzahx.pdf

7.3 Using Collection Services data to find jobs with high disk I/O counts

You can use Collection Services data to look for jobs with high disk I/O counts. To find the jobs with high disk I/O, you are required to have Performance Tools installed. To start looking for jobs that have a high I/O count, use the Component Report from the PERFORM menu. To access the PERFORM menu, enter the following command:

```
GO PERFORM
```

If you want to interactively view the data, select option 7 (Display performance data). Keep in mind your system resources when you choose this option. This job runs interactively and uses CPU and I/O resources. Or you can choose option 3 (Print performance report), which submits a job to batch.

The following sections explain how to find the jobs with high I/O counts either interactively (option 7) or in batch (option 3).

Option 3: Print performance report

When you select option 3, you must specify the library in which you keep your performance data. In most cases, the library is QMPGDATA. You can page up and down until you see a time frame that you want. Type 2 next to the member that you want to use to get a Component Report.

If no members are found and you have started Collection Services, you must create the database files that are needed. See the first bullet point in 7.2.1, “Finding jobs using CPU with the Component Report” on page 183, for information about creating the files.

In the next Select Sections for Report display (Figure 7-3 on page 185), you can press F6 (Print entire report) to select all the sections to be in your report. Since you must often review the storage pool activity and disk activity, we recommend that you use F6. In the display shown in Figure 7-3, you see that we choose option 1 to select Job Workload Activity to find the jobs with high I/O counts.

In the Select Categories for Report display (Figure 7-4 on page 185), we type option 1 to select Time interval. Time interval is usually the best option if you don’t know any other information. If you want more information, you can press F6 to print the entire report, but it is best to narrow the information down to a time frame in which you are interested.

If you selected Time interval on the previous display, you see the Select Time Intervals display (Figure 7-5). You can select intervals that you want to look at based on High Disk Utilization or a time frame.

In the Specify Report Options display (Figure 7-6 on page 186), you specify any report title that you want. You press Enter, and a job is submitted to batch.

You then return to the Print Performance Report - Sample data display and see the following message at the bottom of the display:

```
Job 003675/PCHIDESTER/PRTCPTRPT submitted to job queue QBATCH in library ...
```

This message gives you the submitted job for your report. You can find your report by entering the following command:

```
WRKSBMJOB *JOB
```

In the Work with Submitted Jobs display, your submitted job is called PRTCPTRPT. When the PRTCPTRPT job is in OUTQ status, select option 8 to view the spool file. The report is in spool file QPPTCPTR. Type 5 on the line that precedes the QPPTCPTR file.

Figure 7-14 shows a job that has a high I/O count in comparison to other jobs in the report. The disadvantage of using the printed report versus displaying the data interactively is that the printed Component Report is sorted by the job name, not by disk I/O. Displaying the data interactively allows you to sort on disk I/O. Now that we have isolated a job with relatively high disk I/O, we can look at the Database Monitor data that was running at the same time to investigate what the job was doing. Chapter 6, “Querying the performance data of the Database Monitor” on page 133, gives examples of how to investigate what SQL, if any, that the job was running based on the job name, job user, and job number.

```
                                                  Display Spooled File
 File  . . . . . :   QPPTCPTR
 Control . . . . .
 Find  . . . . . .

*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+....0....
    Library . . : QMPGDATA   System name . . :RCHASM05            Version/Release  :    5/ 3.0  Stopped
  Partition ID  : 000        Feature Code  . :23F5-2434-1520       Int Threshold  . :    .51 %
  Virtual Processors:   2    Processor Units :   2.0
                              T     P         DB
    Job     User Name/ Job    y     t   CPU   Cpb             Tns            -------- Disk I/O --------
    Name    Thread     Number p  Pl y   Util  Util   Tns      /Hour    Rsp     Sync    Async   Logical
---------- ---------- ------ -- -- -- ----- ----- ------- --------- ------- -------- -------- -------- +
QINTER     QSYS       003247 M  02 00   .00    .0       0         0    .000       13        1        0
QPADEV0003 DSQUIRES   003789 P  03 20 28.36  24.0     236       944    .004     4751   267364   847174
QPADEV0004 DSQUIRES   003836 P  03 20   .05    .0     168       828    .051      186       80       63
QPADEV0005 PCHIDESTER 003782 P  03 20   .00    .0      28       112    .051       54       82      789

 F3=Exit   F12=Cancel   F19=Left   F20=Right   F24=More keys
```

Figure 7-14 Example of job having relative high I/O count

Option 7: Display performance data

If you select option 7, you must specify the library in which you keep your performance data. In most cases, the library is QMPGDATA. Next you select the member that you want to investigate, based on the date and time shown. You do this by typing 1 next to the member.

If no members are found and you started Collection Services, you must create the database files needed. See 7.2.1, “Finding jobs using CPU with the Component Report” on page 183, for information about how to create the files.

After you select the member, you can page up and down until you see a time frame you are looking for or until you see a high disk utilization that concerns you. Then you type 1 next to the interval or intervals that you want to examine.

After you select the intervals, you see the Display Performance Data display (Figure 7-8 on page 188). From this display, press F6 to view all jobs. You can use F22 to sort by disk I/O as shown in Figure 7-15. If the disk activity is unusually high and Database Monitor data was gathered during this interval, then the Database Monitor data can be queried as explained in Chapter 6, “Querying the performance data of the Database Monitor” on page 133, to isolate what the job is doing.

```
                                Display Jobs

Elapsed time  . . :   00:15:00          Member  . . . . . :   Q075000002
                                          Library . . . . :     QMPGDATA

Type options, press Enter.
  5=Display job detail   6=Wait detail

                                          Job      CPU      Tns      Avg     Disk
Option  Job         User        Number    Type    Util     Count    Rsp      I/O
        QPADEV0003  DSQUIRES    003789    PTH    28.36       236     .0   272115
        QUERY       DSQUIRES    003834    BCH     2.95         0     .0    42815
        BHAUSER_04  BHAUSER     003760    PTH     1.11       126     .9    40255
        QUERY       DSQUIRES    003835    BCH     2.46         0     .0    38044
        PRTCPTRPT   PCHIDESTER  003837    BCH      .16         0     .0     1748
        QZDASOINIT  QUSER       003738    BCH      .26         0     .0     1274
        QSYSCOMM1   QSYS        003220    SYS      .00         0     .0     1113
        QZDASOINIT  QUSER       003822    BCH      .14         0     .0     1066
        QDBSRVXR2   QSYS        003219    SYS      .00         0     .0      843
        QDBFSTCCOL  QSYS        003221    SYS      .02         0     .0      667
                                                                          More...
F3=Exit           F12=Cancel   F15=Sort by job   F16=Sort by job type
F19=Sort by CPU   F24=More keys
```

Figure 7-15 Job having a high disk I/O count

You can find additional information about Collection Services data and using the tools in “Performance Tools Reports” in the Information Center at*:*

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzahx/rzahxreportperftools.htm

Also refer to *iSeries Performance Version 5 Release 3*, which is available on the Web at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/rzahx/rzahx.pdf

8

Analyzing database performance data with Visual Explain

The launch of Visual Explain with iSeries Navigator Version 4, Release 5, Modification 0 (V4R5M0) in DB2 Universal Database for iSeries was of great interest to database administrators working in an iSeries server environment. This feature has been described as a quantum leap forward in database tuning for query optimization. Visual Explain provides an easy-to-understand graphical interface that represents the optimizer implementation of the query.

For the first time, you can see, in graphic detail, how the optimizer has implemented the query. You can even see all of the facts and figures that the optimizer used to make its decisions. Best of all, the information is presented in one place, in color, with easy-to-follow displays. There is no more jumping between multiple windows, trying to determine what is happening. Even better, if you currently have iSeries Navigator, you already have Visual Explain.

With all of this in mind, is such a richly featured product complicated to use? As long as you are familiar with database tuning, you will enjoy using Visual Explain and want to learn more.

This chapter answers the following questions:

- Where do I find Visual Explain?
- How do I use it?
- What can it be used for?
- Will it tune my Structured Query Language (SQL) queries?
- How can we integrate Visual Explain with the Database Monitors?

8.1 What is Visual Explain

Visual Explain provides a graphical representation of the optimizer implementation of a query request. The query request is broken down into individual components with icons that represent each unique component. Visual Explain also includes information about the database objects that are considered and chosen by the query optimizer. Visual Explain's detailed representation of the query implementation makes it easier to understand where the greatest cost is being incurred.

Visual Explain shows the job run environment details and the levels of database parallelism that were used to process the query. It also shows the access plan in diagram form, which allows you to zoom to any part of the diagram for further details.

If query performance is an issue, Visual Explain provides information that can help you to determine whether you need to:

- Rewrite or alter the SQL statement
- Change the query attributes or environment settings
- Create new indexes

Best of all, you do not have to run the query to find this information. Visual Explain has a modeling option that allows you to explain the query without running it. That means that you can try any of the changes suggested and see how they are likely to work, before you decide whether to implement them.

You can also use Visual Explain to:

- View the statistics that were used at the time of optimization
- Determine whether an index was used to access a table

  If an index was not used, Visual Explain can help you determine which columns might benefit from being indexed.
- View the effects of performing various tuning techniques by comparing the before and after versions of the query graph
- Obtain information about each operation in the query graph, including the total estimated cost and number of rows retrieved
- View the debug messages issued by the query optimizer during the query execution

Visual Explain is an advanced tool to assist you with the task of enhancing query performance, although it does not actually do this task for you. You still need to understand the process of query optimization and the different access plans that you can implement.

Visual Explain is a perfect match with the Database Monitors.

8.2 Finding Visual Explain

Visual Explain is a component of iSeries Navigator and is available under the Databases icon. To locate the Databases icon, you must establish a session on your selected iSeries server using the iSeries Navigator icon.

From the SQL Script Center, you can access to Visual Explain directly, either from the menu or from the toolbar as explained in 8.3.1, "The SQL Script Center" on page 199.

Another way to access Visual Explain is through the SQL Performance Monitors. SQL Performance Monitor is used to create Database Monitor data and to analyze the monitor data with predefined reports.

Visual Explain works with the monitor data that is collected by SQL Performance Monitor on that system or by the Start Database Monitor (STRDBMON) command. Visual Explain can also analyze Database Monitor data that is collected on other systems after that data is restored on the iSeries server.

8.3 Using Visual Explain with the SQL Script Center

The Run SQL Script window (SQL Script Center) provides a direct route to Visual Explain. The window is used to enter, validate, and execute SQL commands and scripts and to provide an interface with i5/OS through the use of CL commands.

8.3.1 The SQL Script Center

To access the SQL Script Center, in iSeries Navigator, expand **Databases**. Then select any database, right-click, and select **Run SQL Scripts**. The Run SQL Script window (Figure 8-1) opens with the toolbar. Reading from left to right, there are icons to create, open, and save SQL scripts, followed by icons to cut, copy, paste, and insert generated SQL (from V5R1) statements within scripts.

The *hour glass icons* indicate to run the statements in the Run SQL Scripts window. From left to right, they run all of the statements, run all of the statements from the cursor to the end (From Selected), or run the single statement identified by the cursor position (Selected).

To the right of the hour glasses is a Stop button, which is in the color red when a run is in progress.

There are two Visual Explain icons in the colors blue and green. The left Visual Explain icon (blue) helps to explain the SQL statement. The right Visual Explain icon (green) enables you to run and explain the SQL statement.

Figure 8-1 Toolbar from Run SQL Scripts

Both of these options are also available from the Visual Explain menu (Figure 8-2). You might choose either option to start Visual Explain.

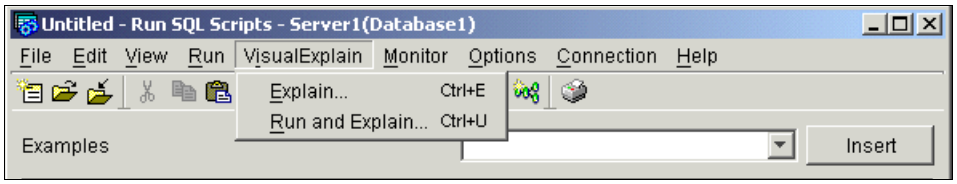


Figure 8-2 SQL Script Center Visual Explain options

The final icon in the toolbar is the Print icon.

You can use SQL Performance Monitors to record SQL statements that are explainable by Visual Explain. We recommend that you obtain access via the SQL Performance Monitors icon, because it provides the full list of monitors.

8.3.2 Explain Only

The Visual Explain Only option (Ctrl + E or the blue toolbar icon) submits the query request to the optimizer and provides a visual explanation of the SQL statement and the access plan that will be used when executing the statement. In addition, it provides a detailed analysis of the results through a series of attributes and values associated with each of the icons. It does not actually run the SQL statement.

To optimize an SQL statement, the optimizer validates the statement, gathers statistics about the SQL statement, and creates an access plan. When you choose the Visual Explain Only option, the optimizer processes the query statement internally with the query time limit set to zero. Therefore, it proceeds through the full validation, optimization, and creation of an access plan and then reports the results in a graphical display.

Note: When you choose Visual Explain Only, Visual Explain might not be able to explain such complex queries as hash join, temp join results, and so on. In this case, you must choose *Run and Explain* for the SQL statements to see the graphical representation.

8.3.3 Run and Explain

The Run and Explain option (Ctrl + U or the green toolbar icon) also submits the query request to the optimizer. It provides a visual explanation of the SQL statement and the access plan used when executing the statement. It provides a detailed analysis of the results through a series of attributes and values associated with each of the icons.

However, it does not set the query time limit to zero and, therefore, continues with the execution of the query. This leads to the display of a results window in addition to the Visual Explain graphics.

Notes:

- Visual Explain might show a representation that is different from the job or environment where the actual statement was run since it might be explained in an environment that has different work management settings.
- If the query is implemented with multiple steps (that is, joined into a temporary file, with grouping performed over it), the Visual Explain Only option cannot provide a valid explanation of the SQL statement. In this case, you must use the Run and Explain option.

8.4 Navigating Visual Explain

The Visual Explain graphics window (Figure 8-3) is presented in two parts. The left side of the display is called the *Query Implementation Graph*. This is the graphical representation of the implementation of the SQL statement and the methods used to access the database. The arrows indicate the order of the steps. Each node of the graph has an icon that represents an operation or values returned from an operation.

The right side of the display has the *Query Attributes and Values*. The display corresponds to the object that has been selected on the graph. Initially, the query attributes and values

correspond to the final results icon. The vertical bar that separates the two sides is adjustable. Each side has its own window and is scrollable.

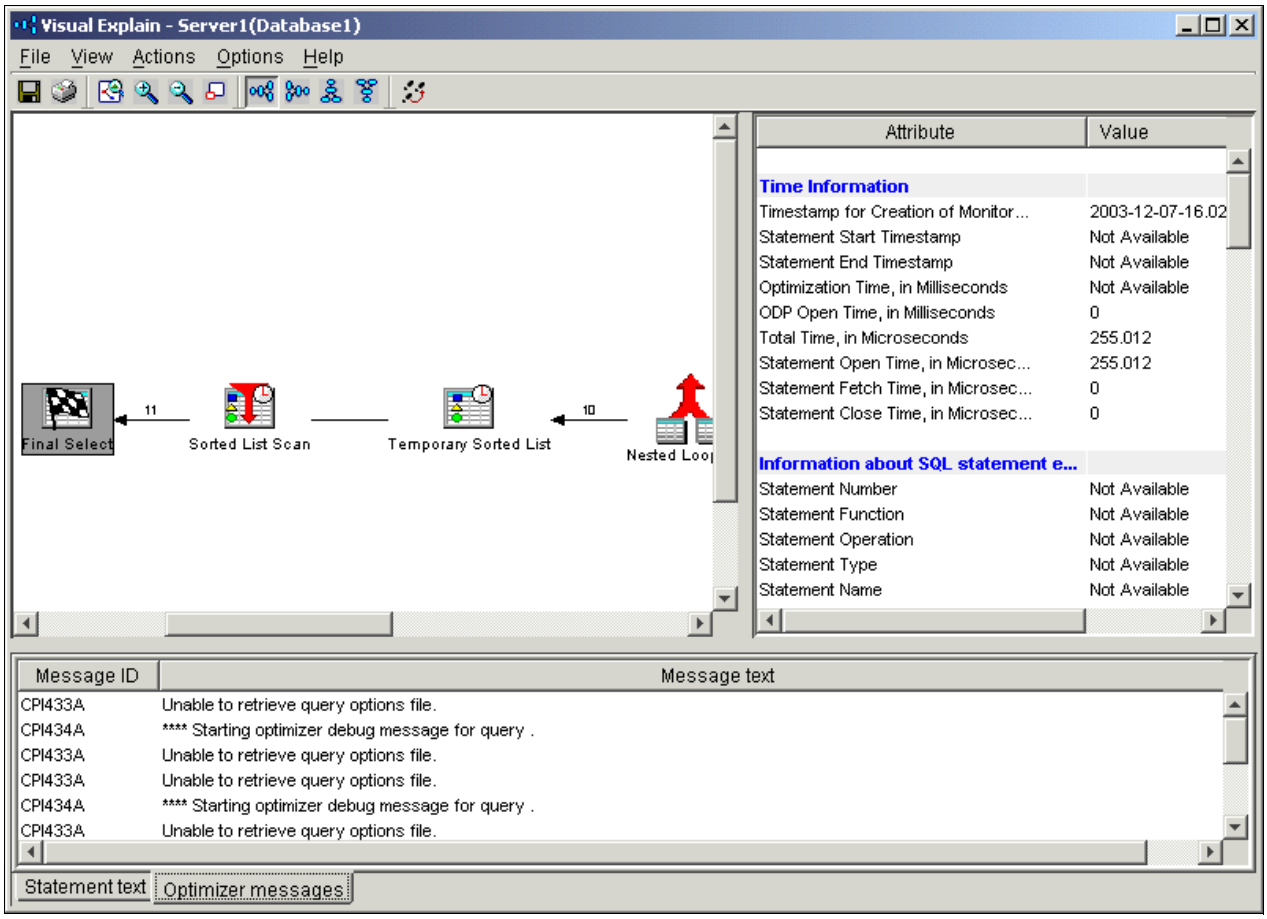


Figure 8-3 Visual Explain Query Implementation Graph and Query Attributes and Values

The default settings cause the display to be presented with the final result icon (a checkered flag) on the left of the display. Each icon on the display has a description and the estimated number of rows to be used as input for each stage of the implementation.

Clicking any of the icons causes the Query Attributes and Values display to change and presents the details that are known to the query for that part of the implementation. You might find it helpful to adjust the display to see more of the attributes and values. Query attributes and values are discussed further in 8.4.5, “Visual Explain query attributes and values” on page 211.

When you right-click any of the icons on the display, an action menu is displayed. The action menu has options to assist with query information. It can provide a shortcut to table information to be shown in a separate window. You can find more details in 8.4.2, “Action menu items” on page 206.

You might find the following action menu items selectively on different icons:

- **Table Description**: Displays table information returned by the Display File Description (DSPFD) command
- **Index Description**: Displays index information returned by the DSPFD command
- **Create Index**: Creates a permanent index on the iSeries server

- **Table Properties**: Displays object properties
- **Index Properties**: Displays object properties
- **Environment Settings**: Displays environment settings used during the processing of this query
- **Additional fly-over panels**: Exist for many of the icons

By moving the mouse pointer over the icon, a window appears with summary information about the specific operation. See Figure 8-4.

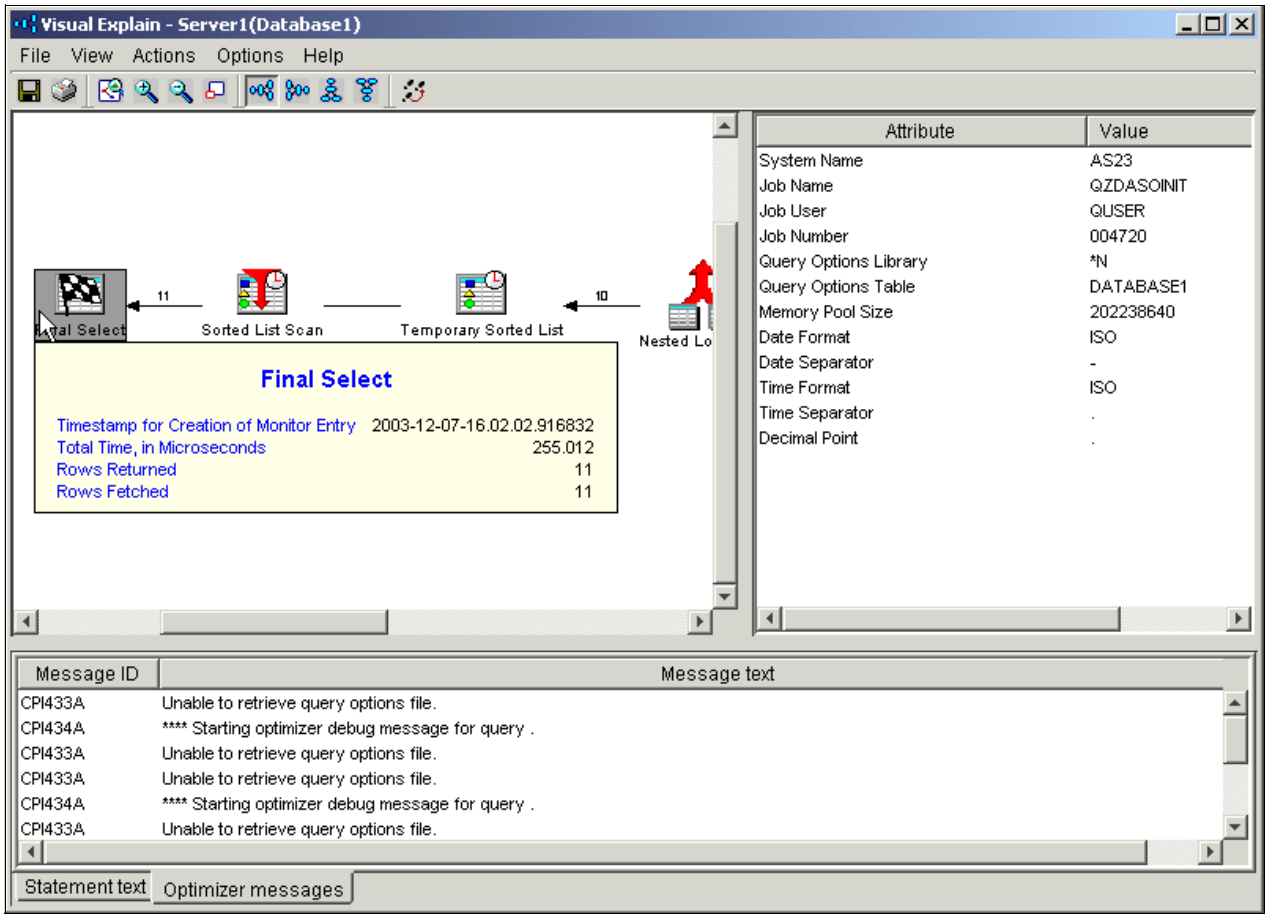


Figure 8-4 Final Select Flyover window

The Visual Explain toolbar (Figure 8-5) helps you to navigate the displays. The first four icons (from left to right after the printer icon) help you to control the sizing of the display. The left-most icon scales the graphics to fit the main window. For many query implementations, this leaves the graphical display too small to be of value. The next two icons allow you to zoom in and out of the graphic image.

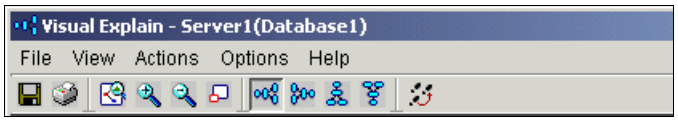


Figure 8-5 Visual Explain toolbar

The fourth icon (Overview) creates an additional window (Figure 8-6) that shows the Visual Explain graphic on a reduced scale. This window has a highlighted area, which represents the part of the image that is currently displayed in the main window.

In the Overview window (Figure 8-6), you can move the cursor into this highlighted area that is shown in the main window. The mouse pointer changes so you can drag the highlighted area to change the section of the overall diagram that is shown in the main window.

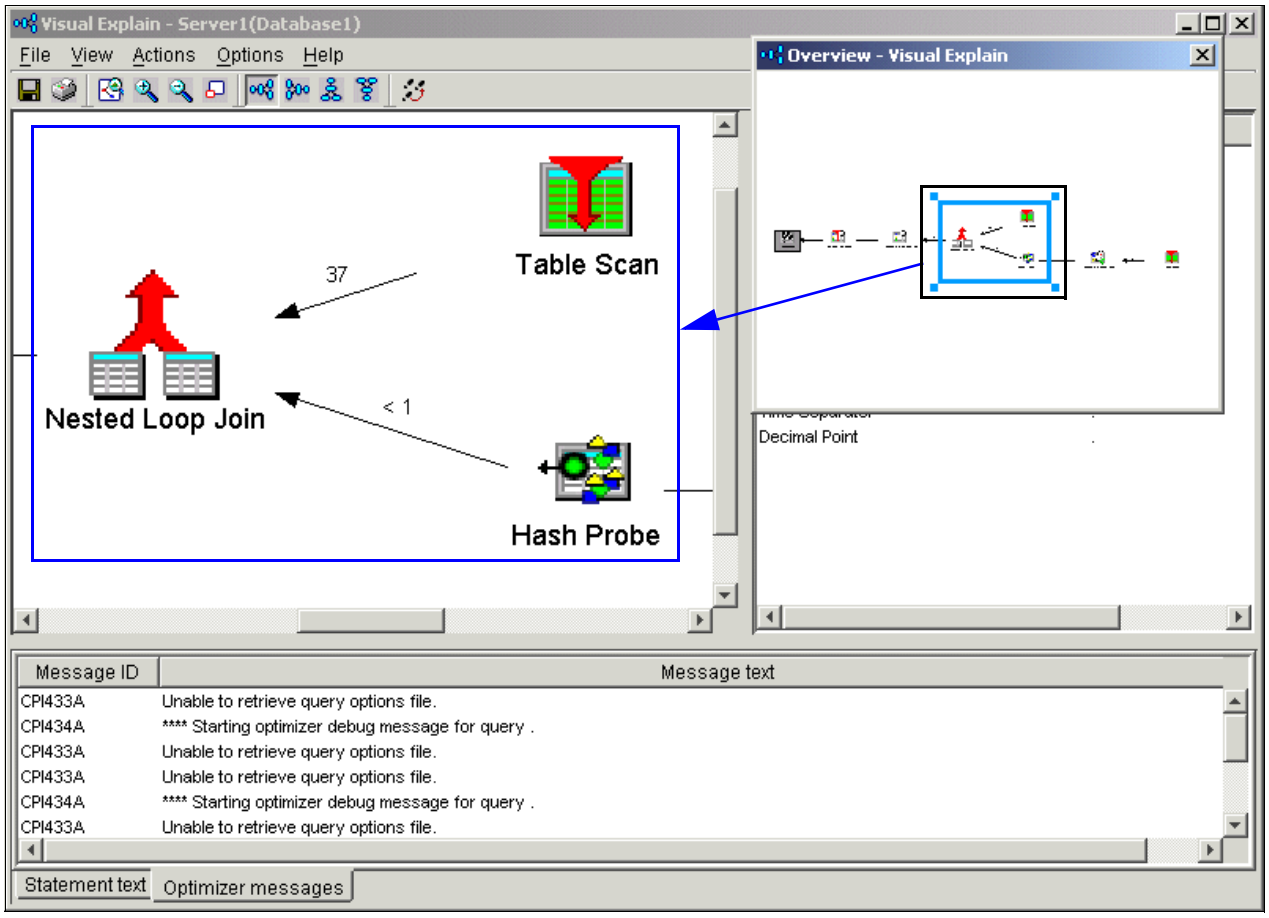


Figure 8-6 Visual Explain Overview window

The default schematic shows the query with the result on the left, working across the display from right to left, to allow you to start at the result and work backward. The remaining four icons on the Visual Explain toolbar allow you to rotate the query implementation image. The icons are:

- Starting from the right, leading to the result on the left (default view)
- Starting from the left, leading to the result on the right
- Starting at the bottom, leading to the result at the top
- Starting from the top, leading to the result at the bottom

Try these icons to see which style of presentation you prefer. Starting in V5R1, a frame at the bottom of the main Visual Explain window was added. In this frame, you can see two tabs. The *Statement Text tab* shows the analyzed SQL statement. Also in V5R1, when Visual Explain is used, it activates the Include Debug Messages in Job Log option and conveniently presents those messages under the *Optimizer Messages tab*.

You can use the last icon (three steps), Statistics and Index Advisor (new in V5R2), to identify missing or stale statistics and to specify how the statistics will be collected (Figure 8-7).

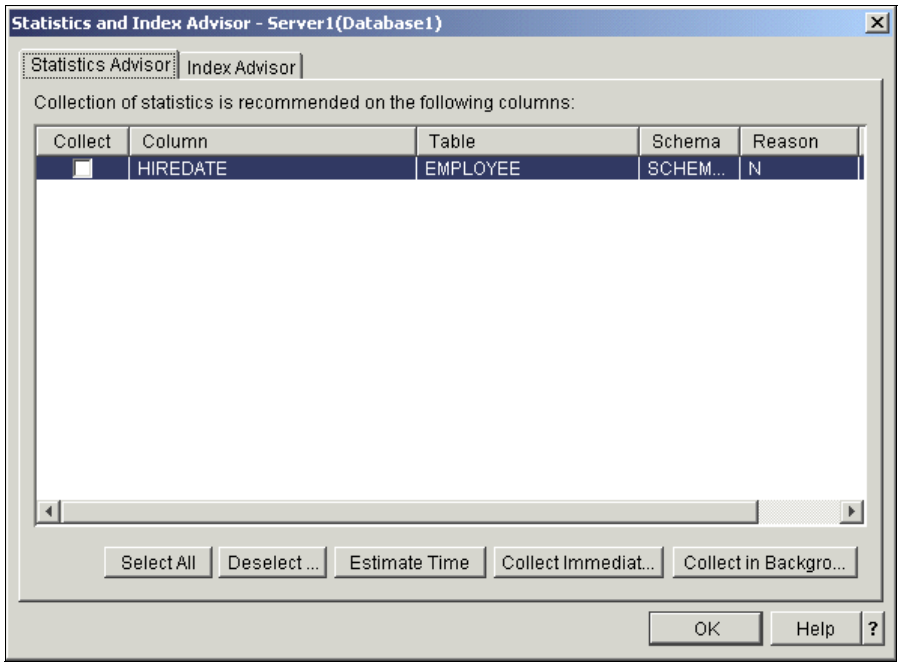


Figure 8-7 Statistics and Index Advisor: Statistics Advisor tab

Additionally, 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 is beneficial, it returns the key columns necessary to create the suggested index. On the Index Advisor tab, you can see the indexes that are recommended for creation as shown in Figure 8-8. You can select the index that you want and click the Create button to create the index selected.

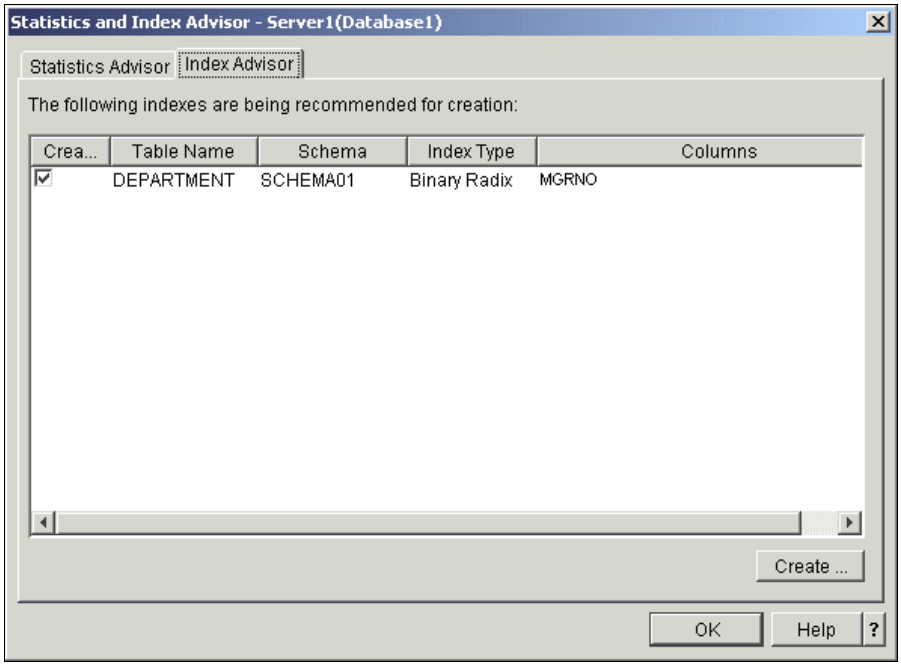


Figure 8-8 Index Advisor in Visual Explain

8.4.1 Menu options

The menu options above the toolbar icons are File, View, Actions, Options, and Help. Starting with V5R1, the ability to either print or save the Visual Explain output as an SQL Performance Monitor file was added. The View options generally replicate the toolbar icons. The additional options are:

- Icon spacing (horizontal or vertical) changes the size of the arrows between the icons.
- Arrow labels allow you to show or hide the estimated number of rows, processing time, or the degree of parallelism that the query is processing at each stage of the implementation.
- Icon labels allow you to show or hide the description of the icons and the object name associated with the icon.
- You can highlight expensive icons by number of returned rows and processing time.
- You can also highlight advised indexes and icons in your graph where the optimizer recommends that you create a permanent index. The icon is highlighted in the graph and the relevant text fields are highlighted in the Icon attributes and values table as shown in Figure 8-9.

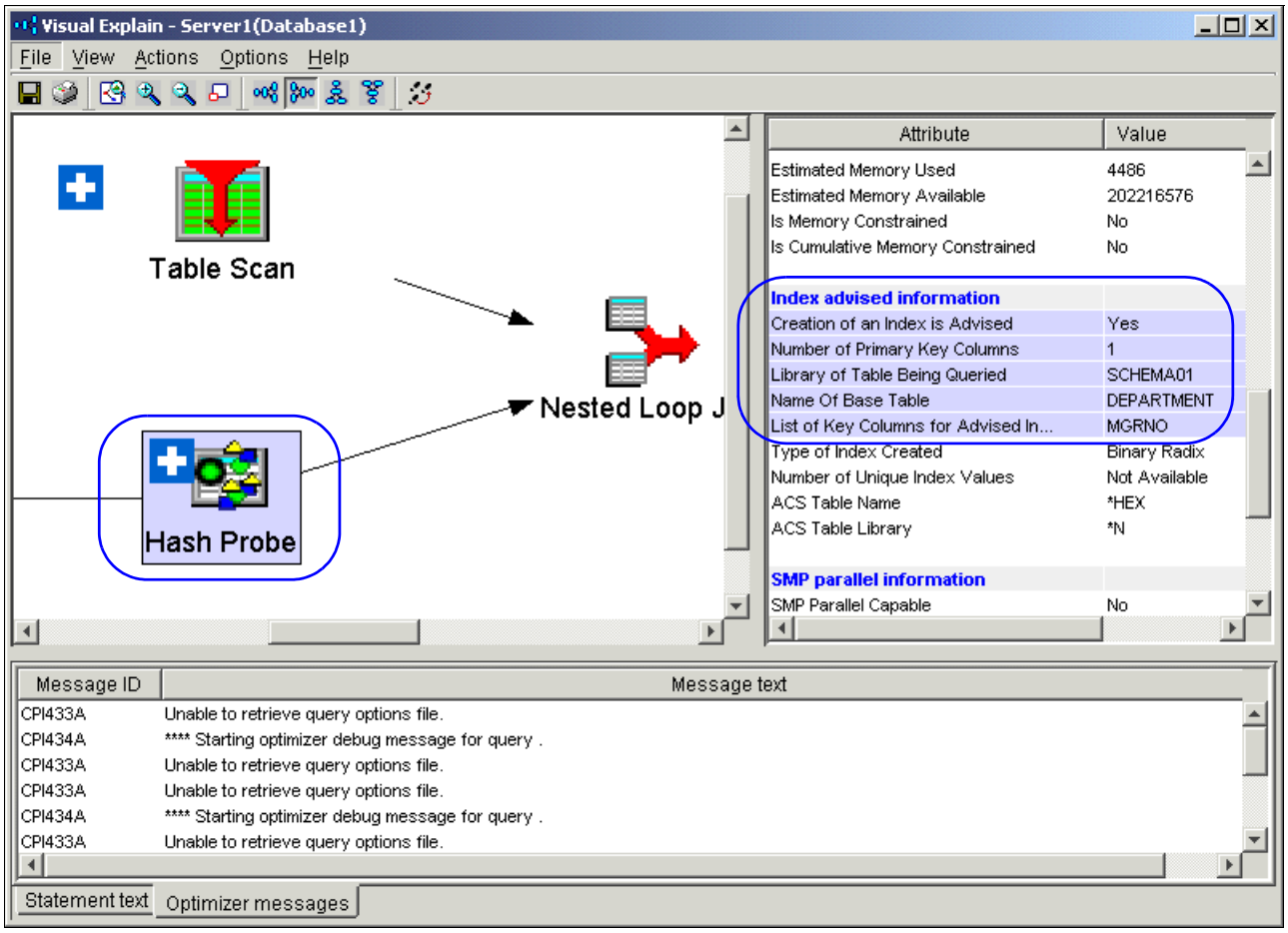


Figure 8-9 Highlighting Index advised information

8.4.2 Action menu items

The Actions menu item replicates the features that are available on the display. When you right-click a query implementation icon, a menu appears that offers further options. These options might include one of more of the items in the following sections.

Table Definition

The Table Definition (Table properties in V5R1) menu item opens the same window that is used to create the table. It enables you to change the definition of the table including columns, constraints, and partitions.

Table Description

The Table Description menu item takes you into the graphical equivalent of the DSPFD command. From here, you can learn more information about the file. The description has several tabs to select to find further information. A limited number of changes can be made from the different tab windows.

Index Description

The Index Description attributes can be accessed to obtain further information about the index. Several changes are allowed to an index from these windows, including access path maintenance settings.

Index Definition

The Index Definition window shows the columns that exist in the table. A sequential number is placed next to the columns that form the index, with an indication of whether the index is ascending or descending. The display also shows the type of index.

Create Index

From the temporary index icon, the Create Index menu item takes you to a window where the attributes of the temporary index are completed as shown in Figure 8-10. Simply click **OK** to create a permanent index.

You need to enter an index and schema name. The type of index is assumed to be binary radix with nonunique keys.

Note: The Create Index menu item is available from any icon where an index is advised (for example, table scan, key positioning, key selection) in addition to the temp index icon. This is one of the easy-to-use features of Visual Explain. It gives you the ability to easily create an index that the optimizer has suggested.

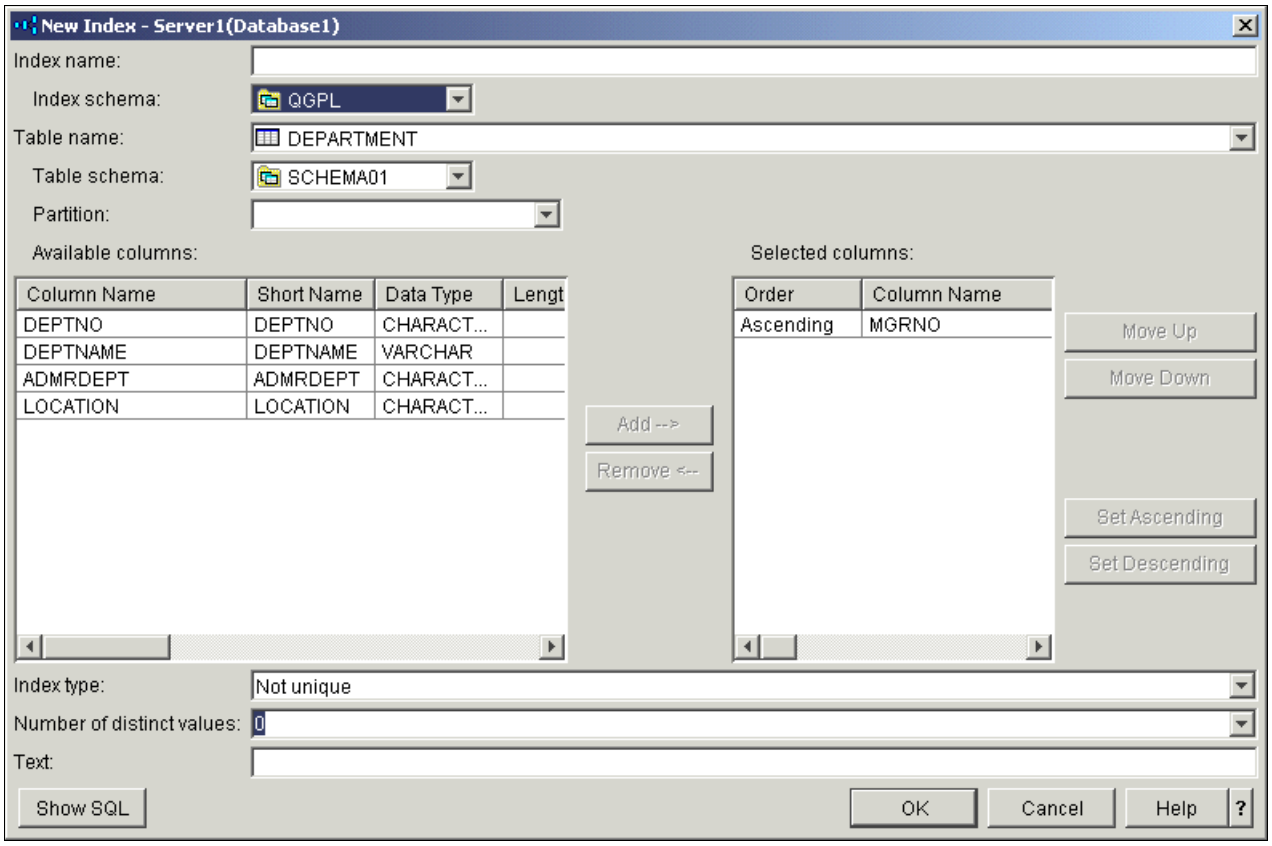


Figure 8-10 Creating an index from Visual Explain

Statistic Data

Use the Statistics Data window (Figure 8-11) to view statistical data for the columns in the selected table. The statistical information can be used by the query optimizer to determine the best access plan for a query. The data presented here can be used to decide whether to collect more statistics, delete existing statistics for a column, or to view more detailed column statistics data.

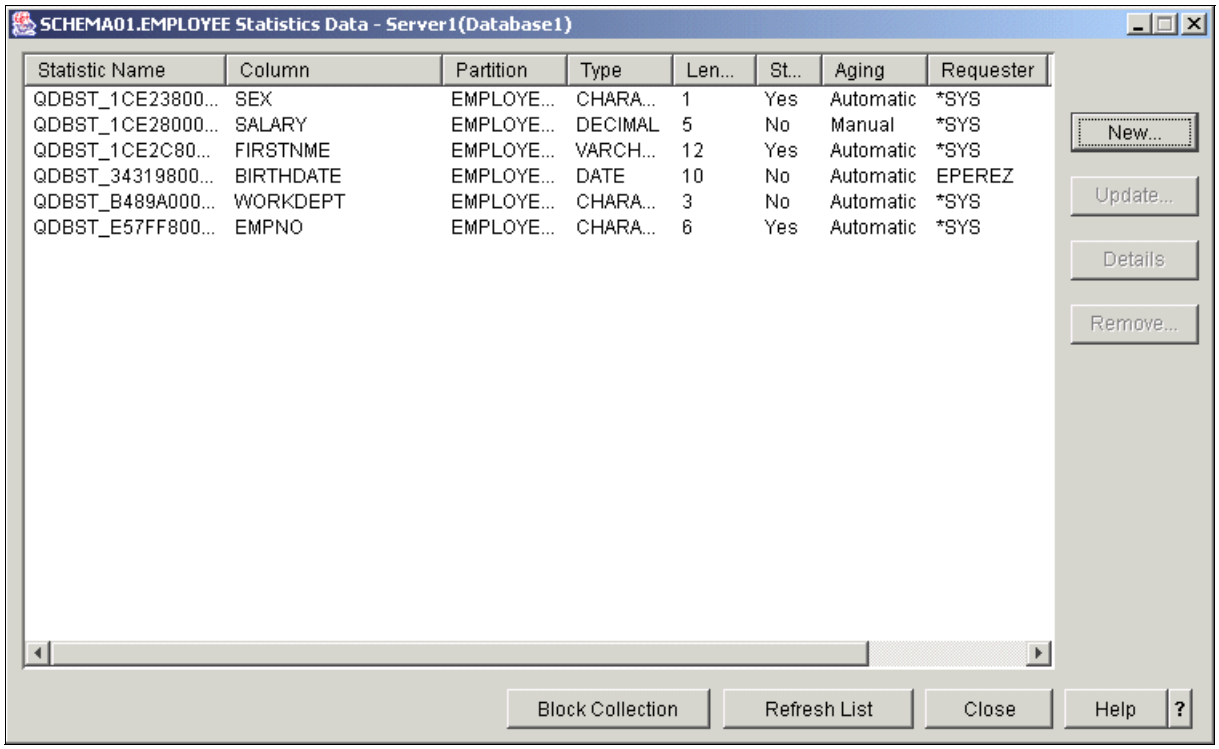


Figure 8-11 Statistic Data window

8.4.3 Controlling the diagram level of detail

You can select how much detail you want to see on the Visual Explain graphs. The menu options enable you to change the level of detail.

Click **Options** → **Graph Detail** → **Basic** to see only the icons that are directly related to the query. Or click **Options** → **Graph Detail** → **Full** to see the icons that are indirectly related to the query, such as table scans performed to build temporary indexes. If you select Full, a white cross is displayed next to some icons where you can right-click the icon and select Expand or Expand to window to view a subgraph of the query operation as shown in Figure 8-12.

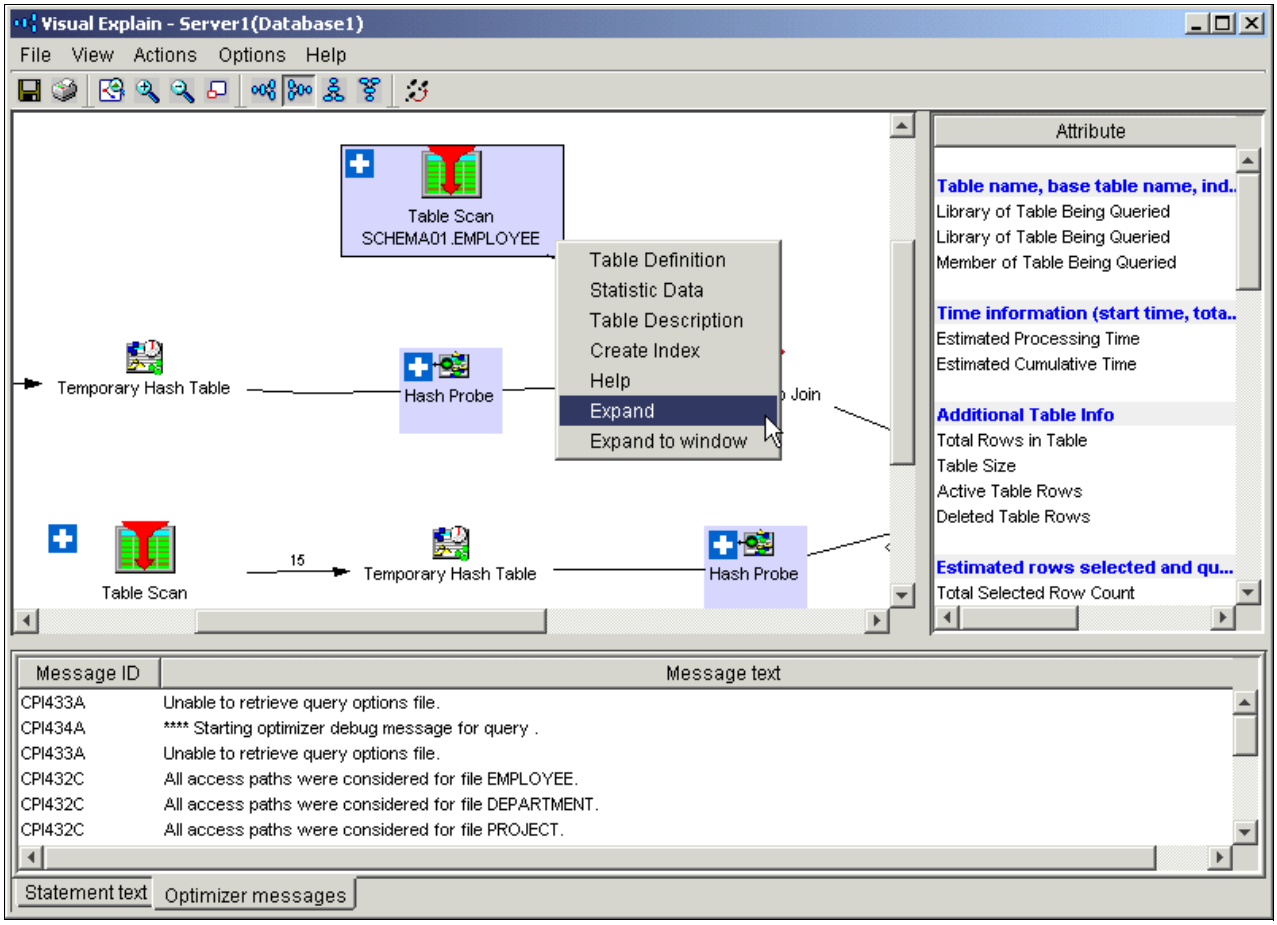


*Figure 8-12 Controlling the diagram level of detail: *Full level (part 1 of 2)*

Figure 8-13 shows a subgraph of the query operation. You can right-click the subgraph and select the Collapse option to see the graph in the original way.

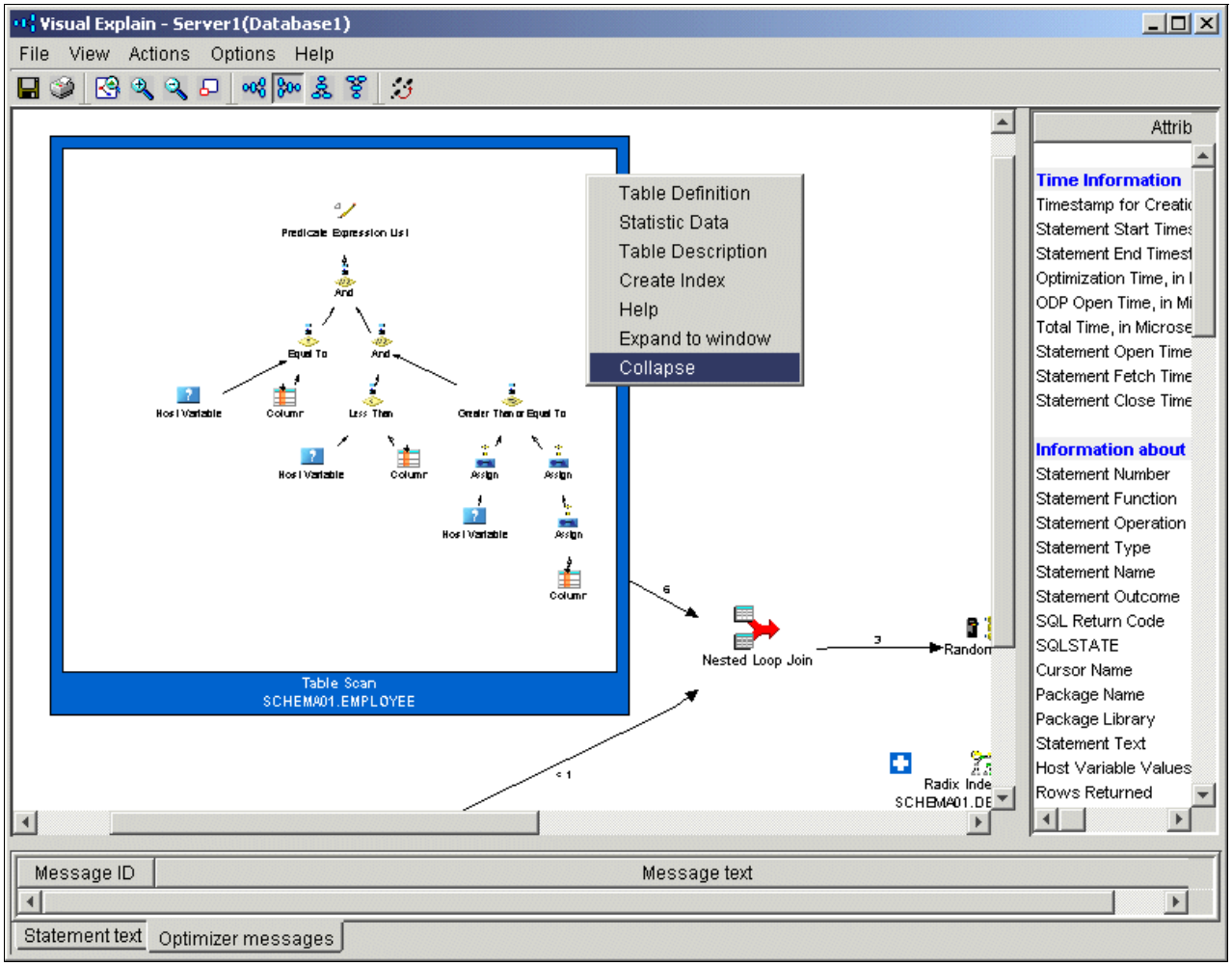


*Figure 8-13 Controlling the diagram level of detail: *Full level (part 2 of 2)*

Most users are satisfied with the *BASIC diagram, while others, with more performance tuning experience, might prefer the *FULL diagram.

Additionally, you can view attributes in Visual Explain in two detail settings: Basic and Full (If Possible). Select the *Basic setting* to retrieve only the basic subset of attributes for each icon. Select the *Full (If Possible) setting* to retrieve a full set of attributes for each icon. The full set of attributes for each icon are shown, unless there is too much data to be returned in Visual Explain. If this happens, basic attributes are retrieved.

8.4.4 Displaying the query environment

The query environment is available as a fast path from the Final Results icon. It shows the work management environment (Figure 8-14) where the query was executed. You can also obtain this information from the Query Attributes and Values displays.

Attribute	Value
System Name	AS23
Job Name	QZDASOINIT
Job User	QUSER
Job Number	004855
Query Options Library	*N
Query Options Table	DATABASE1
Memory Pool Size	151759536
Date Format	ISO
Date Separator	-
Time Format	ISO
Time Separator	.
Decimal Point	.

Figure 8-14 Environment

8.4.5 Visual Explain query attributes and values

The query attributes and values show more information about the optimizer implementation of the query. If you select an icon from the Query Implementation graph, you obtain information about that icon, as well as that part of the query implementation.

We selected a few of the query implementation icons to show you the query attributes and values. This way you can see exactly how much information Visual Explain collects. Prior to Visual Explain, the information was often available, but never in one place.

Table name, base table name, index name

The section in Figure 8-15 shows the name and library or schema of the table being selected.

Attribute	Value
Table name, base table name, index name	
Library of Table Being Queried	EMPLOYEE
Library of Table Being Queried	SCHEMA01
Member of Table Being Queried	EMPLOYEE

Figure 8-15 Table name

Estimated processing time and table info

The estimated processing time (Figure 8-16) shows the time that the optimizer expects to take from this part of the query. The Additional Table Info section shows information about the rows and table size.

Attribute	Value
Time information (start time, total time)	
Estimated Processing Time	1.336E-1
Estimated Cumulative Time	1.336E-1
Additional Table Info	
Total Rows in Table	42
Table Size	4032
Active Table Rows	42
Deleted Table Rows	0

Figure 8-16 Estimated processing time and table information

Estimated rows selected and query join info

The estimated rows selected (Figure 8-17) shows the number of rows that the optimizer expects to output from this part of the query. If the query is only explained, it shows an estimate of the number of rows. If it is run and explained, it shows the number of rows that are selected.

Attribute	Value
Estimated rows selected and query join info	
Total Selected Row Count	5.75
Total Row Count	42
Optimize For N Rows	4.645
Optimize For N Sets	1
Selectivity	1.37E-1
Cumulative Selectivity	1.37E-1
Fetch N Rows	ALL

Figure 8-17 Estimated rows selected and query join info

Estimated Cost Information About the Plan Performed

This section (Figure 8-18) indicates whether the query is CPU or I/O bound. Queries can be CPU-intensive or I/O-intensive. When a query's constraint resource is the CPU, it is called *CPU bound*. When a query's constraint resource is the I/O, it is called *I/O bound*. A query that is either CPU or I/O bound gives you the opportunity to review the query attributes being used when the query was processing. If a symmetric multiprocessor (SMP) is installed on a multiprocessor system, you should review the DEGREE parameter to ensure that you are using the systems resources effectively.

Attribute	Value
Estimated Cost Information About The Plan Performed	
Plan Name	Table Scan
Estimated Processing Time	1.336E-1
CPU Or IO Bound	CPU Bound
IO Cost	0.0
Cpu Cost	1.336E-1
IO Count	0.0
PreLoad Relation	No
Estimated Memory Used	3925
Estimated Memory Available	151759536
Is Memory Constrained	No
Is Cumulative Memory Constrained	No

Figure 8-18 Estimated Cost Information

Information about the index scan performed

This display provides essential information about the index that was used for the query, including the reason for using the index, how the index is being used, and static index attributes. It also specifies the access method or methods used such as Index Scan - Key positioning, Index Scan - Key Selection, and Index Only Access. To find a description about the different reason codes, refer to the manual *DB2 UDB for iSeries Database Performance and Query Optimization* for V5R3, which is available in the Information Center at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzajq/rzajqmst.htm`

SMP parallel information

The SMP information (Figure 8-19) shows the degree of parallelism that occurred on this particular step. It might appear for more than one icon, because multiple steps can be processed with differing degrees of parallelism. This information is relevant only when the DB2 SMP licensed feature is installed.

SMP parallel information	
Parallel Degree Requested	0.0
Max Capable SMP Parallel Degree	1
Parallel Pre Fetch Capable	Yes
Statement Text	CREATE HASHTABLE(SELECT *DEP...
Hash Table Node Name	Node_244

Figure 8-19 SMP parallel information

Index advised information

The Index advised section (Figure 8-20) tells you whether the query optimizer is advising the creation of a permanent index. If an index is being advised, the number and names of the columns to create the index are suggested. This is the same information that is returned by the CPI432F optimizer message. If the Highlight Index Advised option is set, advised index information, such as base table name, library, and involved columns, are easily identifiable.

Attribute	Value
Index advised information	
Creation of an Index is Advised	Yes
Number of Primary Key Columns	1
Library of Table Being Queried	SCHEMA01
Name Of Base Table	EMPLOYEE
List of Key Columns for Advised Index	SEX
Type of Index Created	Binary Radix
Number of Unique Index Values	Not Available
ACS Table Name	*HEX
ACS Table Library	*N

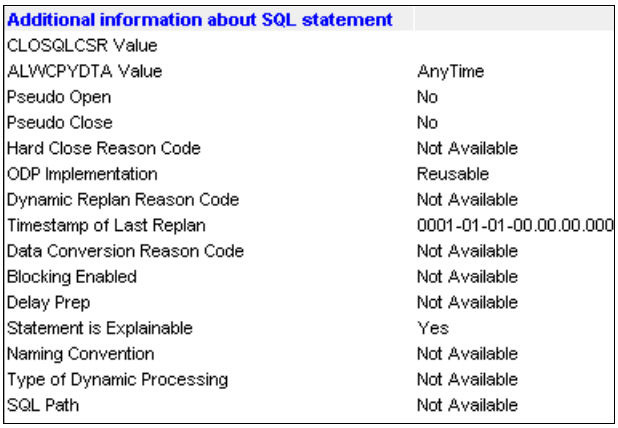
Figure 8-20 Index advised

It is possible for the query optimizer to not use the suggested index, if one is created. This suggestion is generated if the optimizer determines that a new index might improve the performance of the selected data by one microsecond.

Additional information about SQL statement

The display in Figure 8-21 shows information about the SQL environment that was used when the statement was captured. The SQL environment parameters can impact query performance. Many of these settings are taken from the Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) driver settings.

Statement is Explainable specifies whether the SQL statement can be explained by the Visual Explain tool.



Additional information about SQL statement	
CLOSQLCSR Value	
ALWCPYDTA Value	AnyTime
Pseudo Open	No
Pseudo Close	No
Hard Close Reason Code	Not Available
ODP Implementation	Reusable
Dynamic Replan Reason Code	Not Available
Timestamp of Last Replan	0001-01-01-00.00.00.000
Data Conversion Reason Code	Not Available
Blocking Enabled	Not Available
Delay Prep	Not Available
Statement is Explainable	Yes
Naming Convention	Not Available
Type of Dynamic Processing	Not Available
SQL Path	Not Available

Figure 8-21 Additional information

8.5 Using Visual Explain with Database Monitor data

Performance Monitor data is query information that has been recorded by one of the DB2 Universal Database for iSeries Performance Monitors into a database table that can be analyzed later. Multiple Performance Monitors might run on the iSeries at the same time. They can either record information for individual jobs or for the entire system. Each one is individually named and controlled. Any given job can be monitored by a maximum of one system monitor and one job monitor.

You can start Performance Monitors in iSeries Navigator or by using a CL command. With iSeries Navigator, the SQL Performance Monitors component is used to collect Database Monitor data. If you want to use Visual Explain with the data collected with an SQL Performance Monitor, then you must choose the Detailed Monitor collection when setting up the SLQ Performance Monitor in iSeries Navigator.

If you intend to use Visual Explain on the Database Monitor data collected with the CL commands, the data must be imported into iSeries Navigator as detailed data.

Using Visual Explain

In iSeries Navigator, click **Databases** and expand the database that you want to use. Click **SQL Performance Monitors** to obtain a list of the SQL Performance Monitors that are currently on the system.

Right-click the name that you want, and select **List Explainable Statements**. An *explainable statement* (Figure 8-22) is an SQL statement that can be explained by Visual Explain. Because Visual Explain does not process all SQL statements, some statements might not be selected.

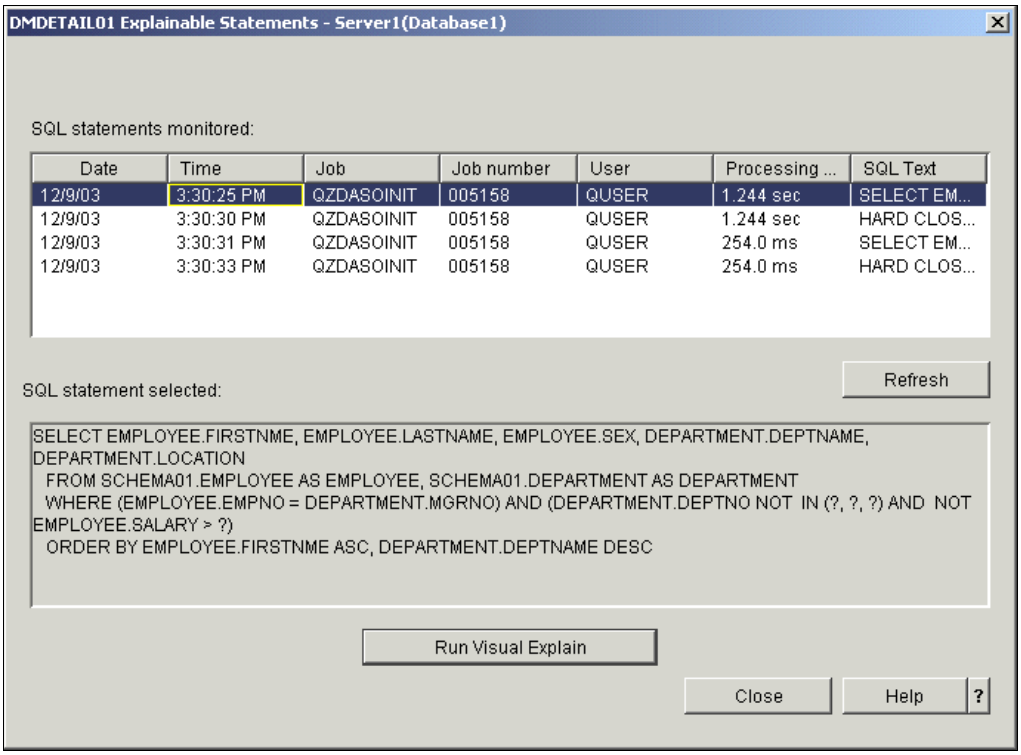


Figure 8-22 SQL explainable statements

The explainable SQL statements that were optimized by the job are now listed. If you were monitoring an SQL Script window, this is the SQL statement that was entered.

Note: Query optimizer information is generated only for an SQL statement or query request when an open data path (ODP) is created. When an SQL or query request is implemented with a reusable ODP, then the query optimizer is not invoked. Therefore, there is no feedback from the query optimizer in terms of monitor data or debug messages. Also, the statement is not explainable in Visual Explain. The only technique for analyzing the implementation of a statement in reusable ODP mode is to look for an earlier execution of that statement when an ODP was created for that statement.

To use Visual Explain on any of the statements, select the statement from the display. The full SQL statement appears in the lower part of the display for verification. Click **Run Visual Explain** (Figure 8-22) to analyze the statement and prepare a graphical representation of the query.

Exit the Visual Explain window and the Explainable Statements window when you have completed your analysis. You might either retain the performance data or remove it from the system at this time, depending on your requirements.

8.6 Using Visual Explain with imported data

You can import Database Monitor data into Visual Explain and then use the tool to help with diagnosing problems further. Visual Explain can be used against current active jobs and against data collected in Performance Monitors either by iSeries Navigator or using the STRDBMON command.

To import a Database Monitor from another system or the same system, you can use iSeries Navigator. Select the system where the data is held. Click **Databases** → **your relational database**. Right-click **SQL Performance Monitors** and select **Import** as shown in Figure 8-23.

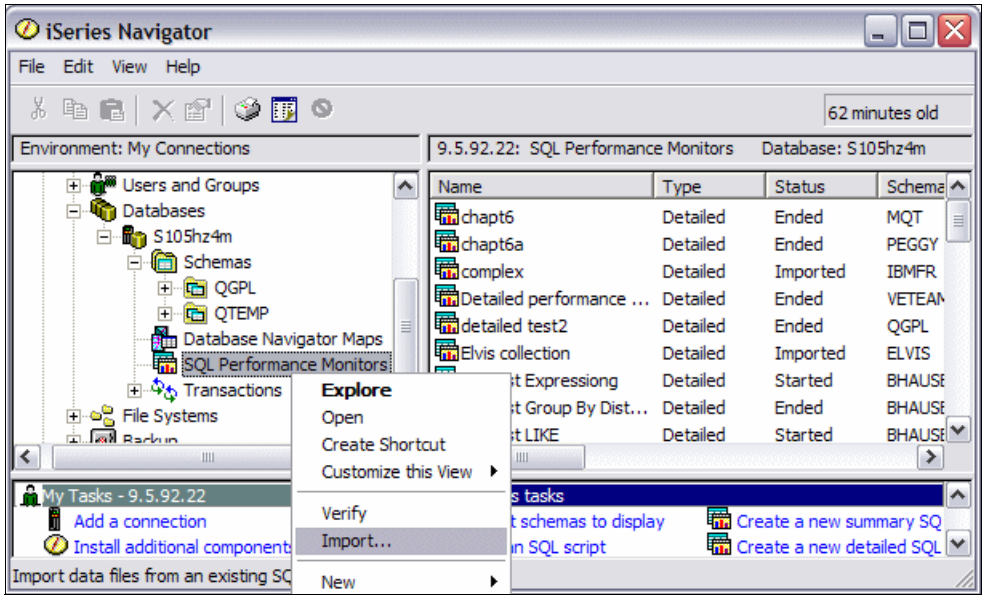


Figure 8-23 Selecting to import a Database Monitor

In the Import SQL Performance Monitor Files window (Figure 8-24), specify the name of the monitor as it will be known in iSeries Navigator and specify the file name and schema in which it resides. For Type of monitor, select either Summary or Detailed depending on how you collected the Database Monitor data.

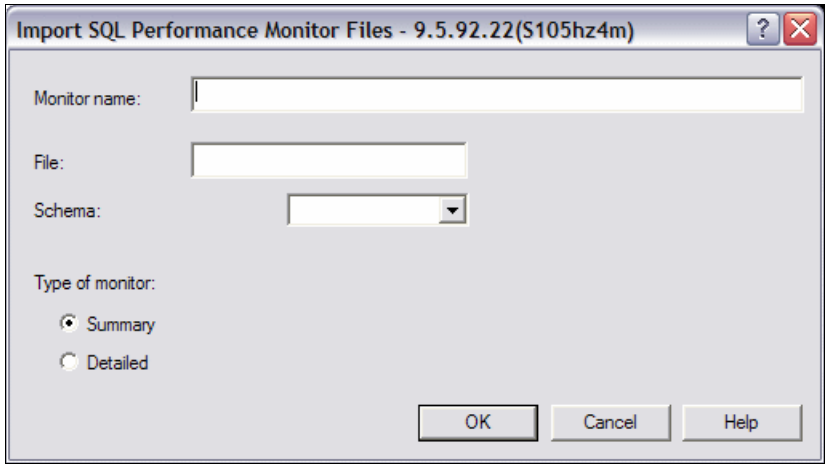


Figure 8-24 Import SQL Performance Monitor Files window

After the monitor is imported, it is displayed in the right pane of the iSeries Navigator window as shown in Figure 8-25.

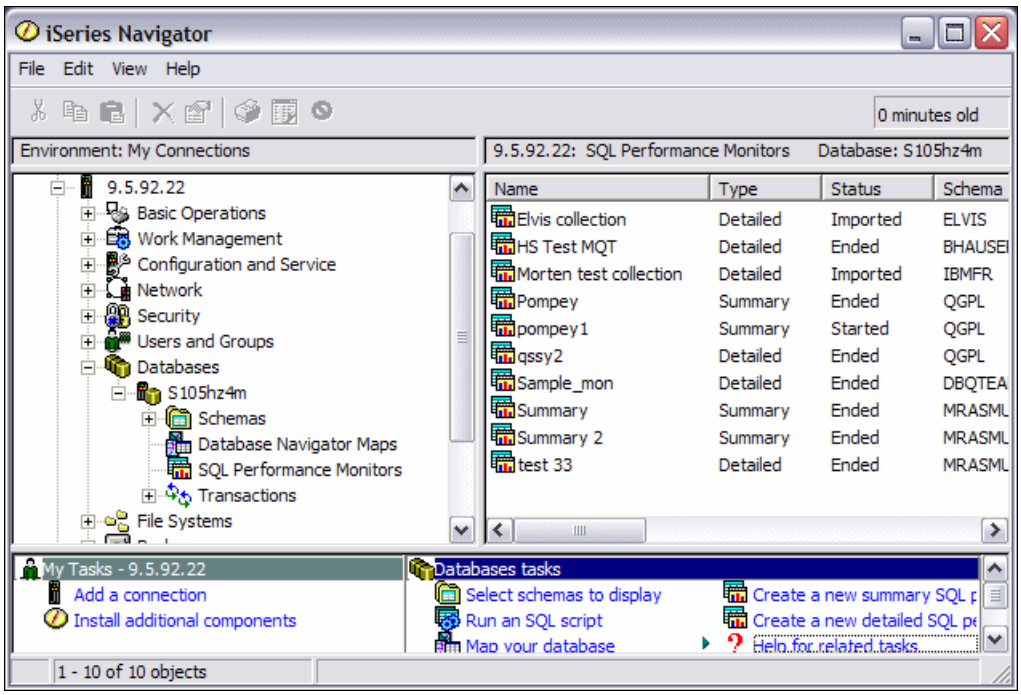


Figure 8-25 SQL Performance Monitors window

After you import the monitor, you can either choose either List Explainable Statements or Analyze results.

8.6.1 List Explainable Statements

When you see the SQL Performance Monitor of interest in the iSeries Navigator window, right-click and select **List Explainable Statements**. A window opens that shows the list of explainable statements (see Figure 8-26). You can select an SQL statement by clicking the required row. You can sort on the columns to help you look for statements of interest, like long running statements as shown in Figure 8-26.

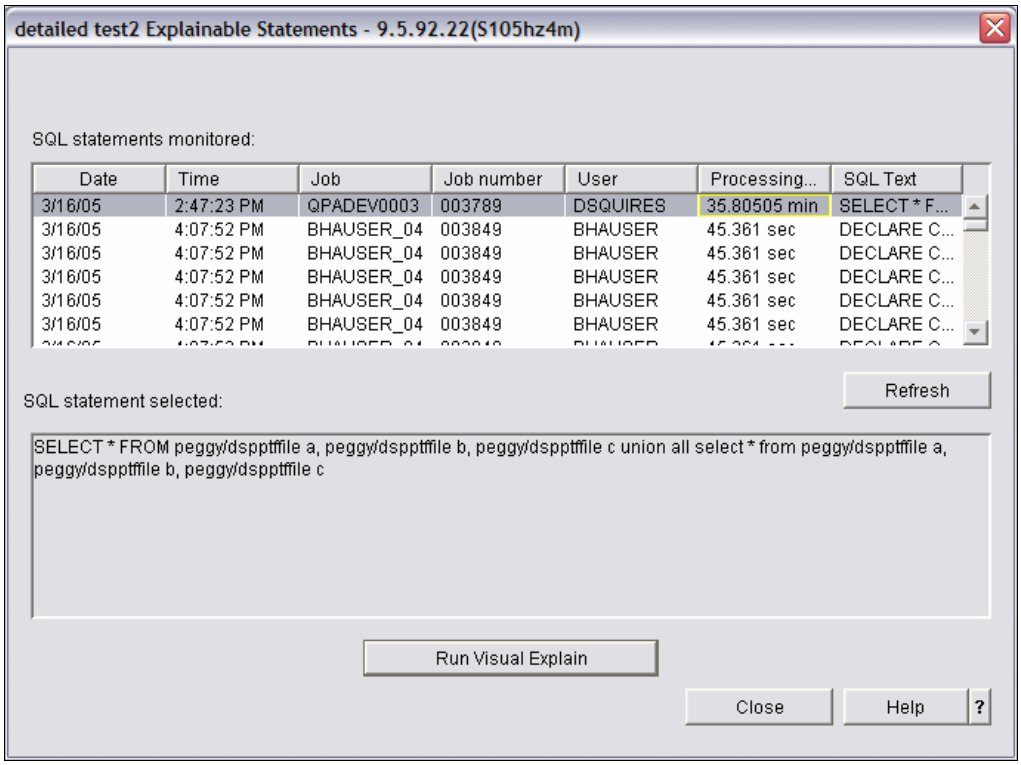


Figure 8-26 List Explainable Statements sorted by most expensive

The SQL statement appears in the bottom half of the window. To obtain the visual explanation, click the **Run Visual Explain** button. After you select an SQL statement for analysis, you can use Visual Explain to understand the implementation the optimizer chose, the time it took to run, any indexes advised, the environmental variables that were used, and the resources that were available at the time the job ran as shown in Figure 8-27.

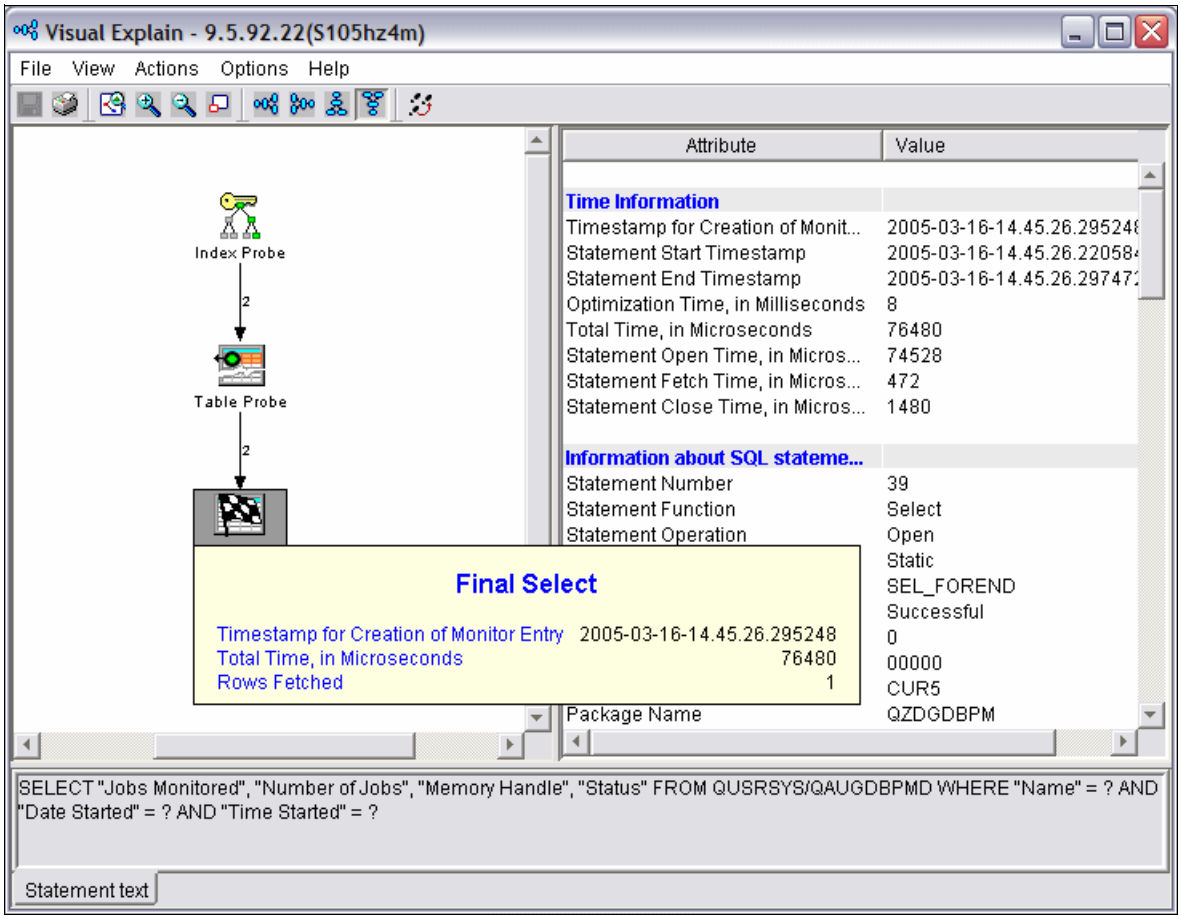


Figure 8-27 Final Select Visual Explain window

8.7 Non-SQL interface considerations

Obviously, the SQL Performance Monitor can capture implementation information for any SQL-based interface. Therefore, any SQL-based request can be analyzed with Visual Explain. SQL-based interfaces range from embedded SQL to Query Manager reports to ODBC and JDBC.

Some query interfaces on the iSeries servers are not SQL-based and, therefore, are not supported by Visual Explain. The interfaces not supported by Visual Explain include:

- Native database access from a high level language, such as Cobol, RPG, and so on
- Query
- OPNQRYF command
- OS/400 Create Query API (QQQQRY)

The query optimizer creates an access plan for all queries that run on the iSeries server. Most queries use the SQL interface. They generate an SQL statement, either directly (SQL Script

window, STRSQL command, SQL in high-level language (HLL) programs) or indirectly (Query Monitor/400).

Other queries do not generate identifiable SQL statements (Query, OPNQRYF command) and cannot be used with Visual Explain via the SQL Performance Monitor. In this instance, the name SQL, as part of the SQL Performance Monitor, is significant.

The statements that generate SQL and that can be used with the Visual Explain via the SQL Performance Monitor include:

- SQL statements from the SQL Script Center
- SQL statements from the Start SQL (STRSQL) command
- SQL statements processed by the Run SQL Statement (RUNSQLSTM) command
- SQL statements embedded into a high level language program (Cobol, Java, or RPG)
- SQL statements processed through an ODBC or JDBC interface

The statements that do not generate SQL and, therefore, that cannot be used with Visual Explain via the SQL Performance Monitor include:

- Native database access from a high level language, for example, Cobol, RPG, and so on
- Query
- Open Query File (OPNQRYF) command
- OS/400 Create Query API (QQQQRY)

8.8 The Visual Explain icons

Table 8-1 lists the icons that you might find on the Visual Explain query implementation chart.

Table 8-1 Visual Explain icons

	The *Final Select* icon displays the original text and summary information about how the query was implemented.
	The *Table* icon indicates that a table was accessed. See the Select icon for more details.
	The *Table Scan* icon indicates that all rows in the table were paged in, and selection criteria was applied against each row. Only those rows that meet the selection criteria were retrieved. To obtain the result in a particular sequence, you must specify the ORDER BY clause.
	The *Table Probe* icon indicates that data from the table must be processed and selected for this query. The table is probed using a key derived from the ordinal number or relative record number associated with each row in the table. The ordinal number is used to calculate the pages of data that need to be retrieved and brought into main memory to satisfy this probe request. The ordinal number used for the probe operation was provided by another data access method previously processed for this query.
	The *Index* icon indicates that an index object was used to process this query.
	The *Index Scan* icon indicates that the entire index will be scanned, which causes all of the entries in the index that are to be paged into main memory to be processed. Any selection criteria whose predicates match the key columns of the index can then be applied against the index entries. Only those key entries that match the specified key selection are used to select rows from the corresponding table data.

	The *Index Probe* icon indicates that the selection criteria whose predicates matched the leading key columns of the index were used to probe directly into the index. The probe selection minimizes the number of key entries that must be processed and paged into main memory. Additional key selection can be applied against the non-leading key columns of the index to further reduce the number of selected key entries. Only key entries that match the specified probe and key selection are used to select rows from the corresponding table data.
	The *Parallel Table Scan* icon indicates that a table scan access method was used and multiple tasks were used to fill the rows in parallel. The table was partitioned, and each task was given a portion of the table to use.
	The *Skip Sequential Table Scan* icon indicates that a bitmap was used to determine which rows were selected. No CPU processing was done on non-selected rows, and I/O was minimized by bringing in only those pages that contained rows to be selected. This icon is usually related to the Dynamic Bitmap or Bitmap Merge icons.
	The *Skip Sequential Parallel Scan* icon indicates that a skip sequential table scan access method was used and multiple tasks were used to fill the rows in parallel. The table was partitioned, and each task was given a portion of the table to use.
	The *Derived Column Selection* icon indicates that a column in the row selected had to be mapped or derived before selection criteria could be applied against the row. Derived column selection is the slowest selection method.
	The *Parallel Derived Column Selection* icon indicates that derived field selection was performed, and the processing was accomplished using multiple tasks. The table was partitioned, and each task was given a portion of the table to use.
	The *Index Key Positioning* icon indicates that only entries of the index that match a specified range of key values were “paged in”. The range of key values was determined by the selection criteria whose predicates matched the key columns of the index. Only selected key entries were used to select rows from the corresponding table data.
	The *Parallel Index Key Positioning* icon indicates that multiple tasks were used to perform the key positioning in parallel. The range of key values was determined by the selection criteria, whose predicates matched the key columns of the index. Only selected key entries were used to select rows from the corresponding table data.
	The *Index Key Selection* icon indicates that all entries of the index were paged in. Any selection criteria whose predicates match the key columns of the index was applied against the index entries. Only selected key entries were used to select rows from the table data.
	The *Parallel Index Key Selection* icon indicates that multiple tasks were used to perform key selection in parallel. The table was partitioned, and each task was given a portion of the table to use.
	The *Encoded-vector Index* icon indicates that access was provided to a database file by assigning codes to distinct key values, and then representing these values in an array (vector). Because of their compact size and relative simplicity, encoded-vector indexes (EVIs) provide for faster scans.
	The *Parallel Encoded-vector Index* icon indicates that multiple tasks were used to perform the EVI selection in parallel. This allows for faster scans that can be more easily processed in parallel.
	The *Encoded-vector Index Scan* icon indicates that the entire EVI will be scanned causing all of the distinct values represented in the index to be processed. Any selection criteria, whose predicates match the key columns of the EVI can then be applied against the distinct values represented in the index. Only those distinct values that match the specified key selection are then used to process the vector and generate either a temporary row number list or temporary row number bitmap.

	The *Encoded-vector Index Probe* icon indicates that the selection criteria whose predicates matched the leading key columns of the EVI were used to probe directly into the distinct values represented in the index. Only those distinct values that match the specified probe selection are then used to process the vector and generate either a temporary row number list or temporary row number bitmap.
	The *Sort Sequence* icon indicates that selected rows were sorted using a sort algorithm.
	The *Grouping* icon indicates that selected rows were grouped or summarized. Therefore, duplicate rows within a group were eliminated.
	The *Nested Loop Join* icon indicates that queried tables were joined together using a nested loop join implementation. Values from the primary file were joined to the secondary file by using an index whose key columns matched the specified join columns. This icon is usually after the method icons used on the underlying tables (that is, index scan-key selection and index scan-key positioning).
	The *Hash Join* icon indicates that a temporary hash table was created. The tables queried were joined together using a hash join implementation where a hash table was created for each secondary table. Therefore, matching values were hashed to the same hash table entry.
	The *Temporary Index* icon indicates that a temporary index was created, because the query either requires an index and one does not exist, or the creation of an index will improve performance of the query.
	The *Temporary Hash Table* icon indicates that a temporary hash table was created to perform hash processing.
	The *Temporary Table* icon indicates that a temporary table was required to either contain the intermediate results of the query, or the queried table could not be queried as it currently exists and a temporary table was created to replace it.
	The *Dynamic Bitmap* icon indicates that a bitmap was dynamically generated from an existing index. It was then used to determine which rows were to be retrieved from the table. To improve performance, dynamic bitmaps can be used in conjunction with a table scan access method for skip sequential or with either the index key position or key selection.
	The *Bitmap Merge* icon indicates that multiple bitmaps were merged or combined to form a final bitmap. The merging of the bitmaps simulates boolean logic (AND/OR selection).
	The *DISTINCT* icon indicates that duplicate rows in the result were prevented. You can specify that you do not want any duplicates by using the DISTINCT keyword, followed by the selected column names.
	The *UNION Merge* icon indicates that the results of multiple subselects were merged or combined into a single result.
	The *Subquery Merge* icon indicates that the nested SELECT was processed for each row (WHERE clause) or group of rows (HAVING clause) selected in the outer level SELECT. This is also referred to as a *correlated subquery*.
	The *Hash Table Scan* icon indicates that the entire temporary hash table will be scanned and all of the entries contained with the hash table will be processed. A hash table scan is generally considered when optimizer is considering a plan that requires the data values to be collated together but the sequence of the data is not required. The use of a hash table scan allows the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary hash table.

The *Hash Table Probe* icon indicates that the selection criteria that match the key columns used to create the temporary hash table will be probed to find all of the matching values stored within the hash table. A hash table probe is generally considered when determining the implementation for a secondary table of a join. The use of a hash table probe allows 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 probe is that the data structure of the temporary hash table usually causes the table data to remain resident within main memory after creation, reducing paging on the subsequent probe operation.

The *Temporary Distinct Hash Table* icon indicates that a temporary distinct hash table was created in order to perform hash processing. A distinct hash table is a data structure that is identical to the temporary hash table, except all duplicate data is compressed out of the temporary being created. The resulting hash table can then be used to perform distinct or aggregate operations for the query.

The *Distinct Hash Table Scan* icon indicates that the entire temporary distinct hash table will be scanned and all of the entries contained with the hash table will be processed. A distinct hash table scan is generally considered when optimizer is considering a plan that requires the data values to be collated together and all duplicate removed but the sequence of the data is not required. The use of a distinct hash table scan allows the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary distinct hash table. An additional benefit of using a distinct hash table scan is that the data structure of the temporary distinct hash table usually causes the table data within the distinct hash table to remain resident within main memory after creation. This benefit reduces the paging on the subsequent scan operations.

The *Distinct Hash Table Probe* icon indicates that the selection criteria that match the key columns used to create the temporary distinct hash table will be probed to find all of the matching values stored within the hash table. The use of a distinct hash table probe allows the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary distinct hash table. An additional benefit of using a distinct hash table probe is that the data structure of the temporary distinct hash table usually causes the table data to remain resident within main memory after creation. This benefit reduces the paging on the subsequent probe operation.

The *Temporary Sorted List* icon indicates that a temporary sorted list was created in order to perform a sequencing operation. A sorted list is a data structure where the table data is collated and sorted based upon the value of a column or columns referred to as the sort key. The sorted list can then be used to return the data in a specified sequence or to perform probe operations using the sort key to quickly retrieve all of the table data that matches a particular sort key.

The *Sorted List Scan* icon indicates that the entire temporary sorted list will be scanned and all of the entries contained with the sorted list will be processed. A sorted list scan is generally considered when optimizer is considering a plan that requires the data values to be sequenced based upon the sort key of the sorted list. The use of a sorted list scan allows the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary sorted list. An additional benefit of using a sorted list scan is that the data structure of the temporary sorted list usually causes the table data within the sorted list to remain resident within main memory after creation. This benefit reduces the paging on the subsequent scan operations.

The *Sorted List Probe* icon indicates that the selection criteria that match the key columns used to create the temporary sorted list is probed to find all of the matching values stored within the sorted list. A sorted list probe is generally considered when determining the implementation for a secondary table of a join when either the join condition uses an operator other than equal or a temporary hash table is not allowed in this query environment. The use of a sorted list probe allows the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary sorted list. An additional benefit of using a sorted list probe is that the data structure of the temporary sorted list usually causes the table data to remain resident within main memory after creation. This benefit reduces the paging on the subsequent probe operation.

The *Temporary List* icon indicates that a temporary list was created. The temporary list was required to either contain the intermediate results of the query, or the queried table could not be queried as it currently exists and a temporary list was created to replace it. The list is an unsorted data structure with no key. The data contained within the list can only be retrieved by a scan operation.

The *List Scan* icon indicates that the entire temporary list will be scanned and all of the entries will be processed.

The *Temporary Row Number List* icon indicates that a temporary row number list was created in order to help process any selection criteria. A row number list is a data structure used to represent the selected rows from a table that matches any specified selection criteria. Since the selected rows are represented by a sorted list of row numbers, multiple lists can be merged and combined to allow for complex selection and processing to be performed without having any paging occur against the table itself.

The *Row Number List Scan* icon indicates that the entire row number list will be scanned and all of the entries will be processed. Scanning a row number list can provide large amounts of savings for the table data associated with the temporary row number list. Since 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.

The *Row Number List Probe* icon indicates that a row number list was used to verify that a row from a previous operation in the query matches the selection criteria used to create the temporary row number list. The use of a row number list probe allows the optimizer to generate a plan that can process the rows in the table in any manner regardless of any specified selection criteria. As the rows are processed, the ordinal number from the row is used to probe into the row number list to determine if that row matches the selection criteria. This is generally found when an index is used to satisfy the ORDER BY from a query and a separate viable index exists to process the selection criteria.

The *Bitmap Scan* icon indicates that the entire bitmap will be scanned and all of the entries that represent selected rows will be processed. Scanning a bitmap can provide large amounts of savings for the table data associated with the temporary bitmap. Since the data structure of the temporary bitmap mirrors the row number layout of the table data, the bitmap can be used to efficiently schedule paging of the table for all selected rows.

The *Bitmap Probe* icon indicates that a bitmap was used to verify that a row from a previous operation in the query matches the selection criteria used to create the temporary bitmap. The use of a bitmap probe allows the optimizer to generate a plan that can process the rows in the table in any manner regardless of any specified selection criteria. As the rows are processed, the ordinal number from the row is used to probe into the bitmap to determine if that row matches the selection criteria. This is generally found when an index is used to satisfy the ORDER BY from a query and a separate viable index exists to process the selection criteria.

The *Index Scan* icon indicates that the entire temporary index will be scanned causing all of the entries in the index to be paged into main memory to be processed. Any selection criteria whose predicates match the key columns of the index can then be applied against the index entries. Only those key entries that match the specified key selection are used to select rows from the corresponding table data.

The *Index Probe* icon indicates that the selection criteria whose predicates matched the leading key columns of the index were used to probe directly into the temporary index. The probe selection minimizes the number of key entries that must be processed and paged into main memory. Additional key selection can be applied against the non-leading key columns of the temporary index to further reduce the number of selected key entries. Only those key entries that matched the specified probe and key selection are used to select rows from the corresponding table data.

The *Temporary Correlated Hash Table* icon indicates that a temporary correlated hash table was created in order to perform hash processing. A hash table is a data structure where the table data is collated based upon the value of a column or columns referred to as the *hash key*. The hash table can then be used to perform probe operation using the hash key to quickly retrieve all of the table data that matches a particular hash value. Because this is a correlated hash table, the hash table needs to be rebuilt prior to any scan or probe operations being performed.

The *Correlated Hash Table Scan* icon indicates that the entire temporary hash table will be scanned and all of the entries contained with the hash table will be processed. A correlated hash table scan is generally considered when optimizer is considering a plan that requires the data values to be collated together but the sequence of the data is not required. In addition, the some of the values used to create the correlated hash table can change from one scan to another. The use of a correlated hash table scan allows the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary correlated hash table. An additional benefit of using a correlated hash table scan is that the data structure of the temporary correlated hash table usually causes the table data within the hash table to remain resident within main memory after creation. This benefit reduces the paging on the subsequent scan operations.

The *Correlated Hash Table Probe* icon indicates that the selection criteria that match the key columns used to create the temporary correlated hash table will be probed to find all of the matching values stored within the hash table. A correlated hash table probe is generally considered when determining the implementation for a secondary table of a join. The use of a hash table probe allows the optimizer to generate a plan that can take advantage of any non-join selection while creating the temporary correlated hash table. An additional benefit of using a correlated hash table probe is that the data structure of the temporary correlated hash table usually causes the table data to remain resident within main memory after creation. This benefit reduces paging on the subsequent probe operation.

The *Temporary Correlated List* icon indicates that a temporary correlated list was created. The temporary correlated list was required to either contain the intermediate results of the query, or the queried table could not be queried as it currently exists and a temporary correlated list was created to replace it. The list is an unsorted data structure with no key that must be rebuilt prior to any scan operation being performed.

The *Correlated List Scan* icon indicates that the entire temporary list will be scanned and all of the entries will be processed.

The *Temporary Buffer* icon indicates that a temporary buffer was created to store the intermediate rows of an operation. The temporary buffer is generally considered at a serialization point within a query to help facilitate operations such as parallelism. The buffer is an unsorted data structure, but it differs from other temporary data structures in that the buffer does not have to be fully populated in order allow its results to be processed.

	The *Buffer Scan* icon indicates that the entire temporary buffer will be scanned and all of the entries will be processed.
	The *Table Random Pre-Fetch* icon indicates that the pages required for the table probe operation will be requested synchronously in the background prior to the actual table probe operation being performed. The system attempts to manage the paging for the table probe to maintain that all of the pages of data necessary to perform the table probe operation stay resident within main memory until they are processed. The amount of pre-fetch paging that is performed by this data access method is dynamically controlled by the system based upon memory consumption and the rate at which rows continue to be processed.
	The *Table Clustered Pre-Fetch* icon indicates that the pages required for the table probe operation will be requested synchronously in the background prior to the actual table probe operation being performed. The system attempts to manage the paging for the table probe to maintain that all of the pages of data necessary to perform the table probe operation stay resident within main memory until they are processed. The amount of pre-fetch paging that is performed by this data access method is dynamically controlled by the system based upon memory consumption and the rate at which rows continue to be processed.
	The *Index Random Pre-Fetch* icon indicates that the pages required for the index probe operation will be requested synchronously in the background prior to the actual index probe operation being performed. The system attempts to manage the paging for the index probe to maintain that all of the pages of data necessary to perform the index probe operation stay resident within main memory until they are processed. The amount of pre-fetch paging that is performed by this data access method is dynamically controlled by the system based upon memory consumption and the rate at which rows continue to be processed.
	The *Logic* icon indicates that the query needed to perform an operation or test against the data in order to generate the selected rows.
	The *Fetch N Rows* icon indicates that a limit was placed upon the number of selected rows. The fetch n rows access method can either be used to implement a user specified limit on the selected rows or it can be combined with other access methods by the optimizer to satisfy complex implementation plans.
	The *Lock Row for Update* icon indicates that an update lock was acquired for the associated table data in order to perform an update or delete operation. To minimize contention between queries, the optimizer attempts to place the lock row for update operation such that the lock is not acquired and held for a long duration.
	The *User-defined table* function icon indicates that a user-defined function that returns a table was used. A table function can be referenced in an SQL FROM clause in the same way that a table or view can be referenced.
	The *Select* icon indicates a point in the query where multiple results are brought together into a single result set. For example, if a query is the union of two different select statements, at the point before the union occurs, the Select icon indicates the points where the select statements finished and the union is about to occur. This icon also represents the default information for an operation that is unknown or not defined elsewhere with Visual Explain. It can help to represent tables or insert, update and delete operations for a query. The summary information for this icon contains any available information to help describe the operation being performed and what the icon represents.
	The *Incomplete Information* icon indicates that a query could not be displayed due to incomplete information.

Part 3

Additional tips

In this part, we provide additional tips to help prevent database performance problems. We present tips regarding indexing strategy and optimizing your SQL statements.

This part encompasses Chapter 9, “Tips to prevent further database performance problems” on page 229.

9

Tips to prevent further database performance problems

It is important to analyze database performance problems and to fix them. For programmers, it is much more important to know how to prevent those problems from the beginning and how to achieve the best performance.

In this chapter, we help you to understand how the optimizer decides for the optimal index, so that you can predict which indexes are necessary. We also provide further tips and information about:

- Optimizing SQL statements
- Coding techniques to avoid with SQL
- Enhancements to the Reorganize Physical File Member (RGZPFM) command

9.1 Indexing strategy

The database has two types of permanent objects, tables and indexes. Tables and indexes include information about the object's structure, size, and attributes. In addition, tables and indexes contain statistical information about the number of distinct values in a column and the distribution of those values in the table. The DB2 Universal Database for iSeries optimizer uses this information to determine how to best access the requested data for a given query request.

Since the iSeries optimizer uses cost-based optimization, the more information given about the rows and columns in the database, the optimizer is better able to create the best possible (least costly and 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 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 concerned with trying to find the rows of interest. A proper indexing strategy assists the optimizer and database engine with this task.

Note: Although you cannot specify indexes in an SQL statement, the optimizer uses them to implement the best access plan to get access to the requested data. You can only specify tables and views in SQL statements.

9.1.1 Access methods

As it stated previously, the database has two types of permanent objects (tables and indexes). There are several methods or algorithms that can be used with this objects:

- Table scan

  With a tables scan, all rows of a tables are processed regardless of the selectivity of the query. Deleted records are examined even though none are selected.

- 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.

- Index scan

  With an index scan, all keys from the index are processed. The resulting rows are sequenced based upon the key columns. This characteristic is used to satisfy a portion of the query request (such as ordering or grouping).

- Index probe

  An index probe reads like an index scan in a keyed sequence, but the requested rows are first identified by a probe operation.

9.1.2 Guidelines for a perfect index

Typically you 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, which gives the optimizer the ability to choose 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:

- Place equal predicates first. Predicates using the equal (=) operator generally eliminate the largest number of nonparticipating rows and should therefore be first in the index.
- If all of the predicates have an equal operator, then order the columns as follows:
  a. Selection predicates + join predicates
  b. Join predicates + selection predicates
  c. Selection predicates + group by columns
  d. Selection predicates + order by columns
- Always place the most selective columns as the first key in the index.

Note: Indexes consume system resources, so it up to you to find a balance between query performance and system (index) maintenance.

The query shown in Example 9-1 uses the ITEMS table and finds all the customers who returned orders at year end 2000 that were shipped via air. We illustrate the perfect indexes for this query.

Example 9-1 One table query

```
SELECT CUSTOMER, CUSTOMER_NUMBER, ITEM_NUMBER
       FROM ITEMS
       WHERE     "YEAR"     = 2000
             AND "QUARTER" =    4
             AND RETURNFLAG = 'R'
             AND SHIPMODE   = 'AIR'
       ORDER BY  CUSTOMER_NUMBER, ITEM_NUMBER
```

The query has four local selection predicates and two ORDER BY columns.

Following the guidelines, the perfect index places the key columns first that cover the equal predicates (“YEAR”, “QUARTER”, RETURNFLAG, SHIPMODE), followed by the ORDER BY columns CUSTOMER_NUMBER, ITEM_NUMBER as specified in Example 9-2.

Example 9-2 Perfect index for the one table query example

```
CREATE INDEX MySchema/ItemIdx01
          ON MySchema/Items
         ("YEAR", "QUARTER", ReturnFlag, ShipMode,
          Customer_Number,   Item_Number)
```

Encoded-vector index guidelines

An encoded-vector index (EVI) cannot be used for grouping or ordering and has a limited use in joins. Single key EVIs can be used to create bitmaps or relative record number (RRN) lists that can be used in combination with binary radix tree indexes. You might use this technique when the local selection contains AND or OR conditions and a single index does not contain all the proper key columns or a single index cannot meet all of the conditions.

If you look at the query in Example 9-1, you see four local selection predicates and two ORDER BY columns. Following the EVI guidelines, single key indexes are created with key columns covering the equal predicates as shown in Example 9-3.

Example 9-3 EVIs for the one table query example

```
CREATE ENCODED VECTOR INDEX MySchema/ItemsEVI_Year
    ON MySchema/Items ("YEAR");

CREATE ENCODED VECTOR INDEX MySchema/ItemsEVI_Quarter
    ON MySchema/Items ("QUARTER");

CREATE ENCODED VECTOR INDEX MySchema/ItemsEVI_ReturnFlag
    ON MySchema/Items (RETURNFLAG);

CREATE ENCODED VECTOR INDEX MySchema/ItemsEVI_ShipMode
    ON MySchema/Items (SHIPMODE);
```

9.1.3 Additional indexing tips

Keep in mind that indexes are used by the optimizer for the optimization phase or the implementation phase of the query. For example, you might create an index that you do not see is being used by the optimizer for the implementation phase, but it might have been used for the optimization phase. There are some additional considerations to remember where the optimizer might not use an index:

- Avoid NULL capable columns if expecting to use index only access. When any key in the index is NULL capable, the index-only access method cannot be used with the Classic Query Engine (CQE).
- Avoid derived expressions in local selection. Access via an index might not be used for predicates that have derived values. Or, a temporary index is created to provide key values and attributes that match the derivative. For example, if a query includes one of the following predicates, the optimizer considers that predicate to be a derived value and might not use an index for local selection:

  ```
  T1.ShipDate > (CURRENT DATE – 10 DAYS)
  UPPER(T1.CustomerName) = "SMITH"
  ```

- Index access is not used for predicates where both operands come from the same table. For example, if the WHERE clause contains the following snippet, the optimizer does not use an index to retrieve the data since it must access the same row for both operands.

  ```
  T1.ShipDate > T1.OrderDate
  ```

- Consider *index only access (IOA)*. If all of the columns used in the query are represented in the index as key columns, the optimizer can request index only access. With IOA, DB2 Universal Database for iSeries does not have to retrieve any data from the actual table. All of the information required to implement the query is available in the index. This might eliminate the random access to the table and drastically improve query performance.
- Use the most selective columns as keys in the index. Give preference to columns used in equal comparisons.
- For key columns that are unique, use a unique constraint.
- Make sure that statistics exist for the most and least selective columns for the query.

9.1.4 Index Advisor

The optimizer has several feedback mechanisms to help you identify the need of an index. To help you to identify the perfect index for a given query request, you can use the following methods to find the requested index:

- STRDBG (Start Debug) CL command
- Running and analyzing an SQL statement using Visual Explain Index Advisor in iSeries Navigator

STRDBG CL command

You can use the STRDBG CL command to analyze the feedback of the optimizer on the green-screen interface. Before you start SQL by using the Start SQL (STRSQL) CL command or before you call your program containing embedded SQL statement, you must enter the STRDBG command. With this command, all SQL messages are registered in the job log.

If an access path, either SQL index or keyed logical file is found, you see the appropriate message in the job log (see Figure 9-1).

```
Query options retrieved file QAQQINI in library QUSRSYS.
All access paths were considered for file ORDHDR.
Access path of file ORDHDRIO2 was used by query.
Query options used to build the OS/400 query access plan.
```

Figure 9-1 Job log for an executed SQL statement where an access path is available

If no access path is available, you might find an access path suggestion. Figure 9-2 shows the job log for an SQL statement, where an access path is suggested.

```
Query options retrieved file QAQQINI in library QUSRSYS.
Arrival sequence access was used for file ORDHDR.
Access path suggestion for file ORDHDR.
Query options used to build the OS/400 query access plan.
```

Figure 9-2 Job log for an executed SQL statement, where no access path is available

If you look at the detailed message for the access path suggestion (position the cursor on the message and press F1), you see the recommended key fields as in Figure 9-3.

```
                        Additional Message Information

Message ID . . . . . :   CPI432F       Severity . . . . . :   00
Message type . . . . . :   Information
Date sent  . . . . . :   03/15/05      Time sent  . . . . . :   11:02:37

Message . . . . :   Access path suggestion for file ORDHDR.
Cause . . . . . :   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 ORDHDR of file ORDHDR in library
  MYSCHEMA.
    In the list of key fields that follow, the query optimizer is recommending
  the first 1 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:
    ORDER_DATE.
    If file ORDHDR in library MYSCHEMA is a logical file then the access
  path should be built over member ORDHDR of physical file ORDHDR in library
  MYSCHEMA.
```

Figure 9-3 Access path suggestion: Detailed information

Running and analyzing an SQL statement using Visual Explain Index Advisor in iSeries Navigator

If you execute an SQL script or use Performance Monitors, you can use Visual Explain to analyze your SQL statements. Visual Explain has a feature, called *Index Advisor*, that might recommend the required indexes and offers a way to create them easily.

To execute and analyze an SQL statement:

1. Start iSeries Navigator.
2. Under My Connections, click **your connection** → **Databases**.
3. Select **your database** and, in the Database tasks area in the bottom right pane, select **Run an SQL Script**. See Figure 9-4.

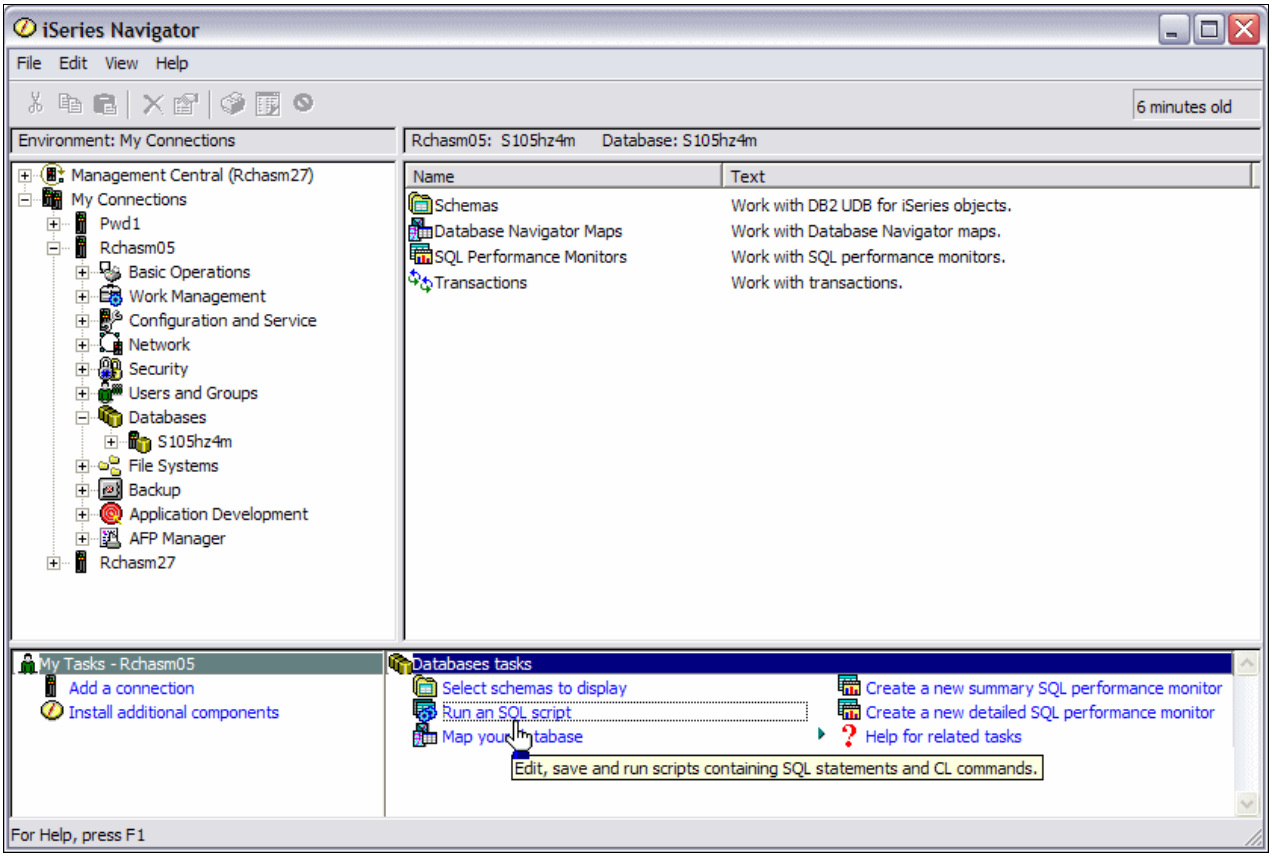


Figure 9-4 Selecting Run an SQL script

4. In the Run an SQL script window, type your SQL statement and highlight it. From the toolbar, click the **Visual Explain** option and select either **Explain** or **Run Explain**.

   Figure 9-5 illustrates these steps to start Visual Explain for a specified SQL statement.

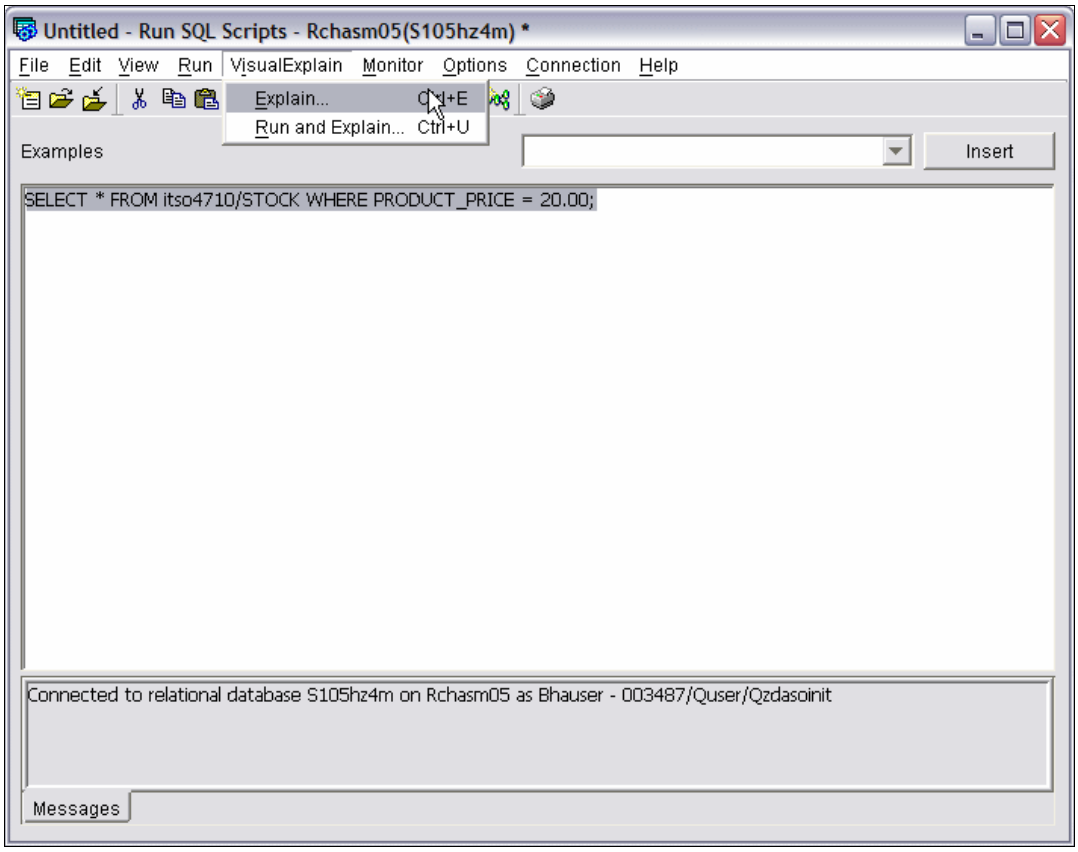


Figure 9-5 Start Visual Explain

5. The window in Figure 9-6 shows the Visual Explain presentation of the SQL statement. Note the joblog messages in the bottom pane of the window. In this example, you see that a table scan is performed, but an index is suggested. To see which index is recommended, click **Actions** and select **Advisor**.

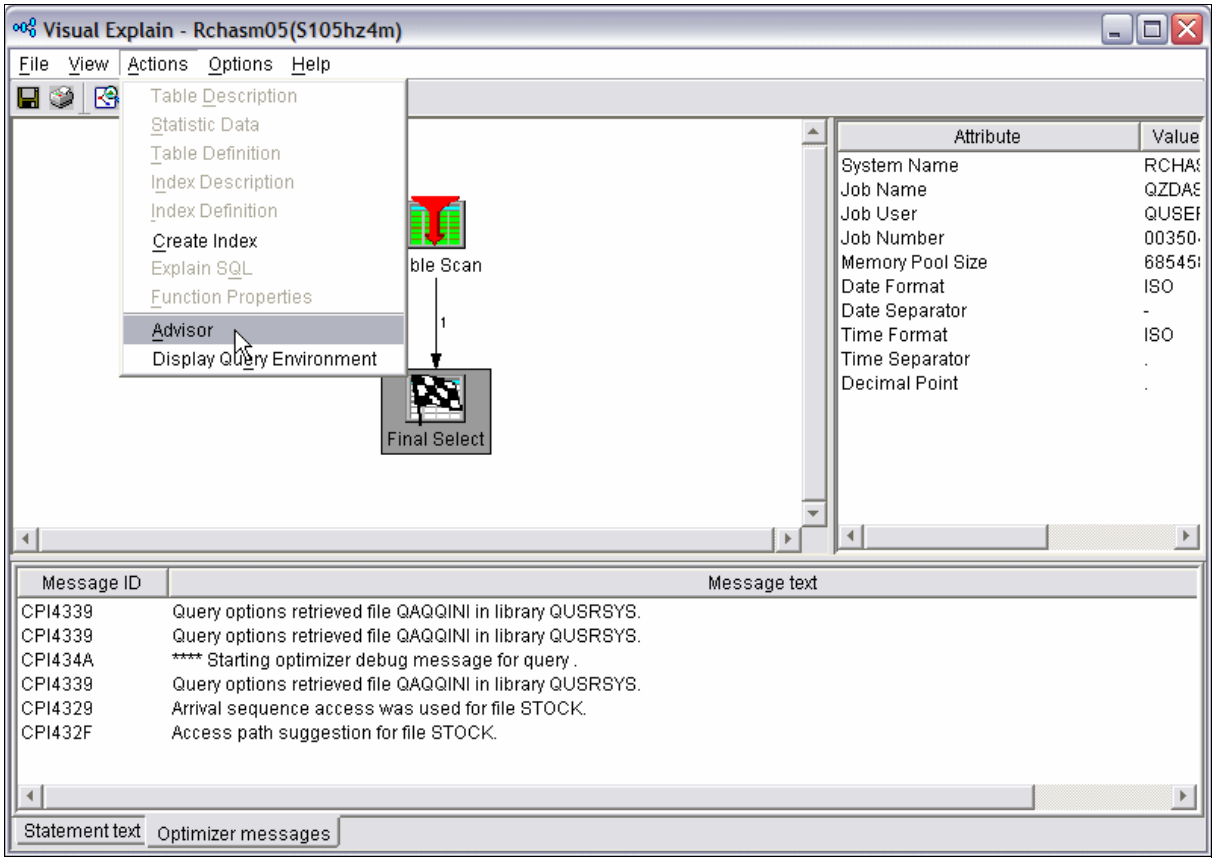


Figure 9-6 Starting the Index Advisor

The Index and Statistics Advisor window (Figure 9-7) shows you all the recommended indexes and statistic columns. If you click **Create**, you can easily create the suggested indexes. All key columns are listed in the desired sequence.

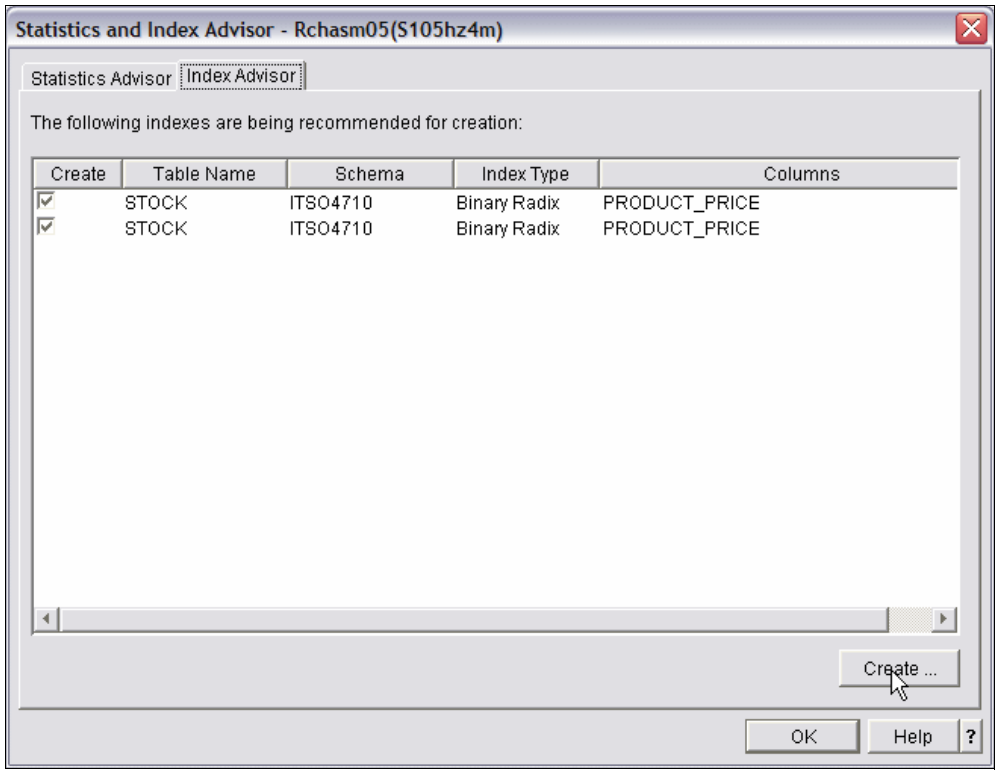


Figure 9-7 Index Advisor showing the recommended indexes

9.2 Optimization of your SQL statements

It is not possible to force the optimizer to use a particular index, but you can affect the optimizer's decision by coding the SQL select statement in different ways. In the following section, we show you some ways that you can affect the optimizer.

9.2.1 Avoid using logical files in your select statements

It is a common misunderstanding that, by specifying logical files in SQL statements, the query optimizer can be forced to choose this index. This is not true. The specified index can be selected if it meets all the requirements for the best access path. But the query optimizer can choose any other index or decide even on doing a table scan as well.

If you specify a keyed logical file in an SQL select statement, the optimizer takes only the column selection, join information, and select/omit clauses, and rewrites the query. For each of these joined tables, a specific index is determined.

There is another point to avoid specifying logical files in an SQL statement. A select statement that contains logical files is rerouted and executed by the CQE and does not benefit from the enhancements of the new SQL Query Engine (SQE). The cost of rerouting might cause an overhead of up to 10 to 15% in the query optimization time.

We illustrate this with an example. Suppose that we have a base table ORDHDR that contains all order header information. In addition to a primary key constraint, two logical files are created.

Example 9-4 show the DDS description of the keyed logical file ORDHDRL1, with the key fields ORHDTE = Order Date and ORHNBR = Order Number.

Example 9-4 Logical file ORDHDRL1

```
A          R ORDHDRF                   PFILE(ORDHDR)
 *
A          K ORHDTE
A          K ORHNBR
```

Example 9-5 shows the DDS description of the second logical file ORDHDRL2, with the key fields ORHDLY = Delivery Date in descending order, CUSNBR = Customer Number, and ORHNBR = Order Number.

Example 9-5 Logical file ORDHDRL2

```
A          R ORDHDRF                   PFILE(ORDHDR)
 *
A          K ORHDLY                    DESCEND
A          K CUSNBR
A          K ORHNBR
```

Now we want to select all orders with the order date 11 March 2005. In the first case, we perform the select using the logical file ORDHDRL2, and in the second case, we use the physical file.

Figure 9-8 shows the Visual Explain for both statements. In both cases, the access path of the keyed logical file is used. But in the first case, an Index Scan - Key Positioning is performed that refers to the CQE, while in the second case, an index probe in combination with a table probe is executed that refers to SQE.

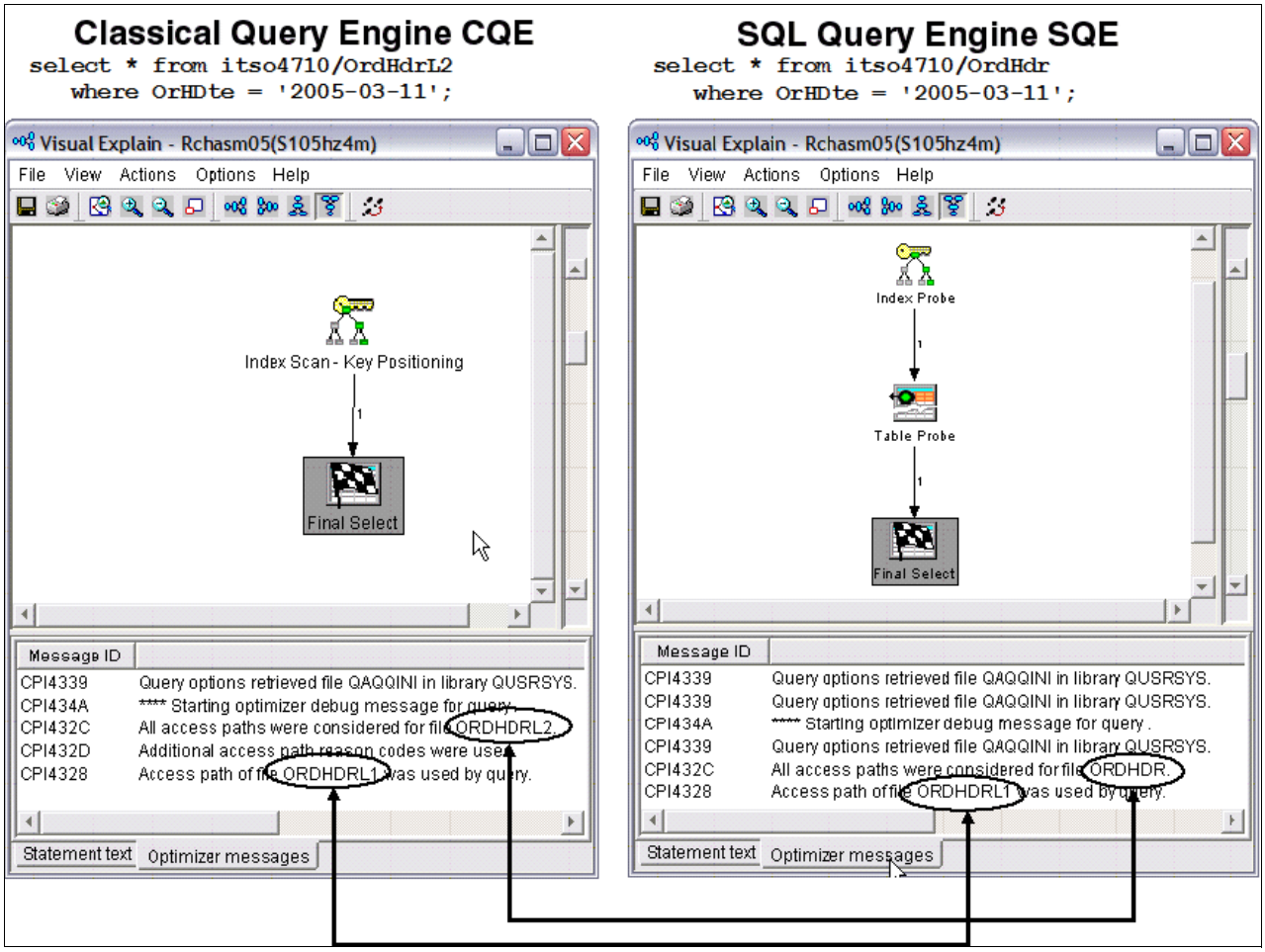


Figure 9-8 Comparing the use of logical and physical files in select statements

9.2.2 Avoid using SELECT * in your select statements

When reading a record/row with native I/O, all fields/columns and field/column values are always moved into memory. If you have tables with a lot of columns and you only need information from a few, a lot of unnecessary information must be loaded. With SQL, you can select only the columns that you need to satisfy the data request.

If you specify the required columns in your select statements, the optimizer might perform index only access. With IOA, DB2 Universal Database for iSeries does not have to retrieve any data from the actual table. All of the information required to implement the query is available in the index. This might eliminate the random access to the table and drastically improve query performance.

9.2.3 Avoid using the relative record number to access your data

For most tables, it is easy to define a primary or unique key. In some cases, such as for transaction tables, it is almost impossible to determine a unique key. With native I/O, the required records are often read using the relative record number. For native I/O, this is the fastest access method.

In SQL, the relative record number can be determined by using the scalar function RRN(). The relative record number is not a defined row in the database table. Therefore, it is not possible to create an index over the relative record number. If you try to read a row using the relative record number, a table scan is always performed.

For tables where you cannot specify a unique key, you must create an additional column to hold a unique value. SQL provides methods, such as identity columns or columns with ROWID data type, where a unique value is automatically generated if a new row is inserted. Identity columns or columns with ROWID data type are ideal for primary or unique key constraints. However, it is also possible to create an index over these columns.

Figure 9-9 shows the selection of a specific row using the SQL scalar function RRN().

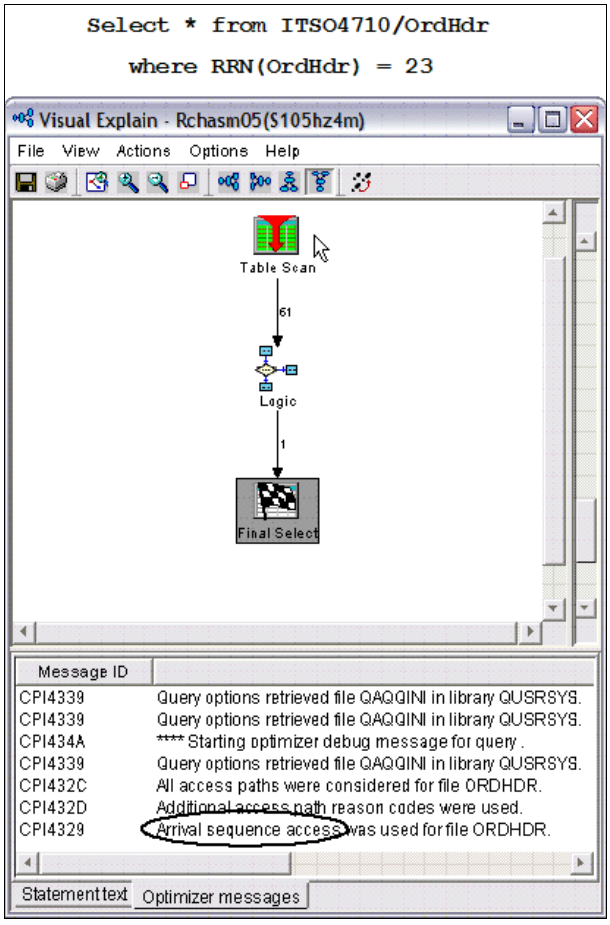


Figure 9-9 Row selection by using the SQL scalar function RRN()

9.2.4 Avoid numeric data type conversion

When a column value and a host variable (or constant value) are being compared, try to specify the same data types and attributes. A query that uses the CQE 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 you have different numeric definitions between the column and host variable or constant value, the queries that use the SQE can use the existing indexes.

If different numeric definitions exist, and no index is available, neither of the two query engines suggests an index.

To avoid problems for columns and constants being compared, use:

- The same data type
- The same scale, if applicable
- The same precision, if applicable

Figure 9-10 shows an SQL statement that selects all orders with an order total greater than 200. The order_total column is defined as decimal 9.2. In the WHERE clause, we used a precision of 4. We executed the query twice. On the first one, we specified a logical file. This way, the query was forced to use CQE. The second one was executed by SQE. Executing this query with CQE, a table scan is performed. With SQE, an index is found and used.

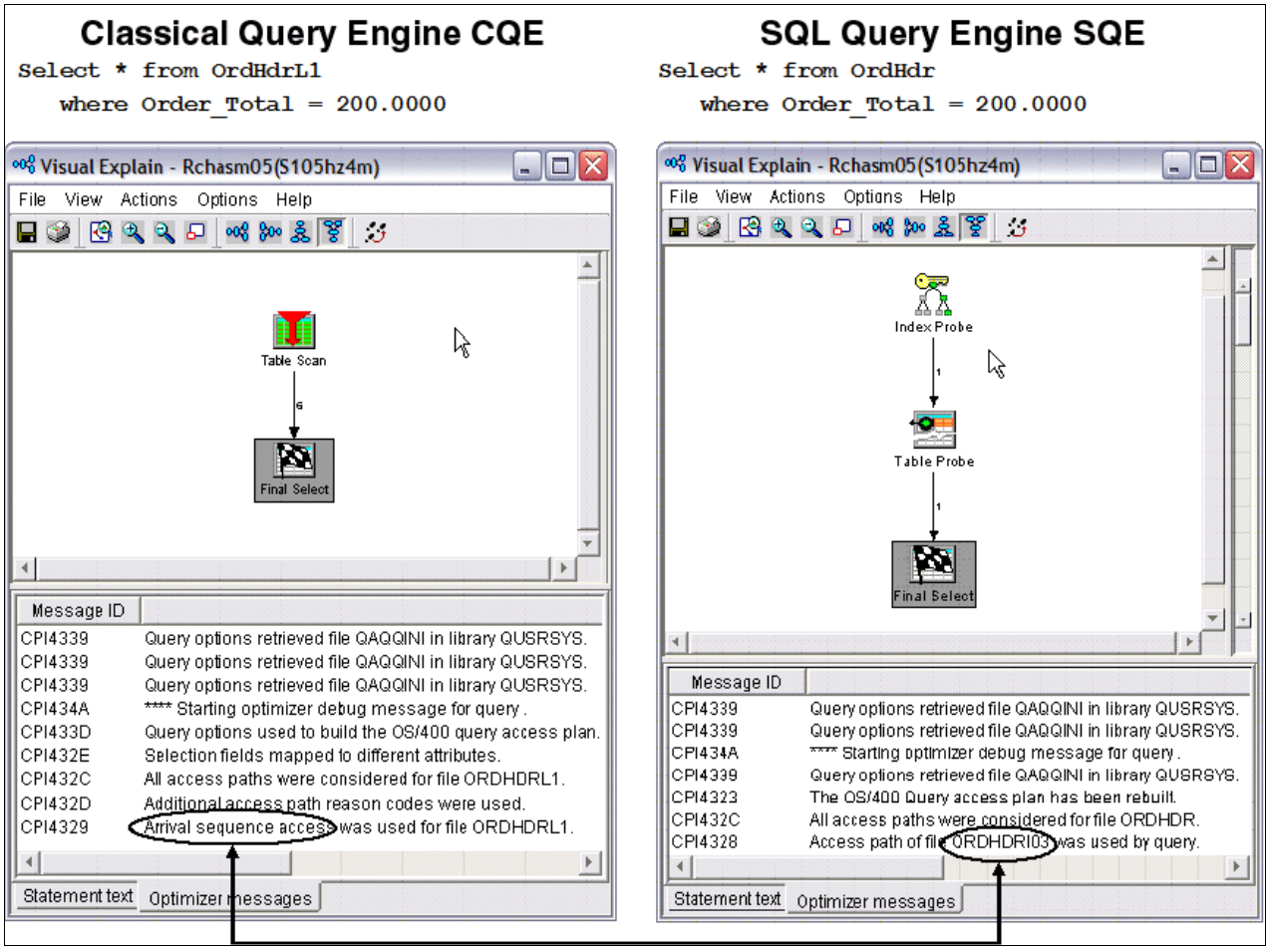


Figure 9-10 Row selection using different numeric data types in the where clause

With SQE, an index can be used if the numeric value is casted to another numeric format, such as Integer or Float. With CQE, a table scan is performed.

9.2.5 Avoid numeric 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 might not cause an index over the column to become unusable, it prevents any estimates and possibly the use of index scan-key positioning on the index.

While using CQE, a table scan might be performed. SQE can use the index but must do an additional relative record scan.

Example 9-6 shows part of an RPG program that selects the orders for next seven days. The new date is calculated in a numeric expression.

Example 9-6 Using numeric expressions in embedded SQL

```
D CountOrder      S              5I 0
 *-------------------------------------------------------------------------------------------
C/EXEC SQL
C+ Select Count(Order_Number)
C+   into :CountOrder
C+   from  OrdHdr
C+   Where Order_Delivery = Current_Date + 7 Days
C/End-Exec

C     CountOrder    Dsply
C                   Return
```

Instead of a numeric expression in the SQL statement, Example 9-7 shows that a host variable is defined and the new date is filled from the RPG program.

Example 9-7 Using host variables instead of numeric expressions in SQL

```
D NextWeek        S               D
D CountOrder      S              5I 0
 *-------------------------------------------------------------------------------------------
C                   Eval      NextWeek    = %Date() + %Days(7)
C/EXEC SQL
C+ Select Count(Order_Number)
C+   into :CountOrder
C+   from  OrdHdr
C+   Where Order_Delivery = :NextWeek
C/End-Exec

C     CountOrder    Dsply
C                   Return
```

Figure 9-11 shows the Visual Explain for both SQL Statements. Notice the table scan when CQE executes the query. Using a host variable instead, an index can be used.

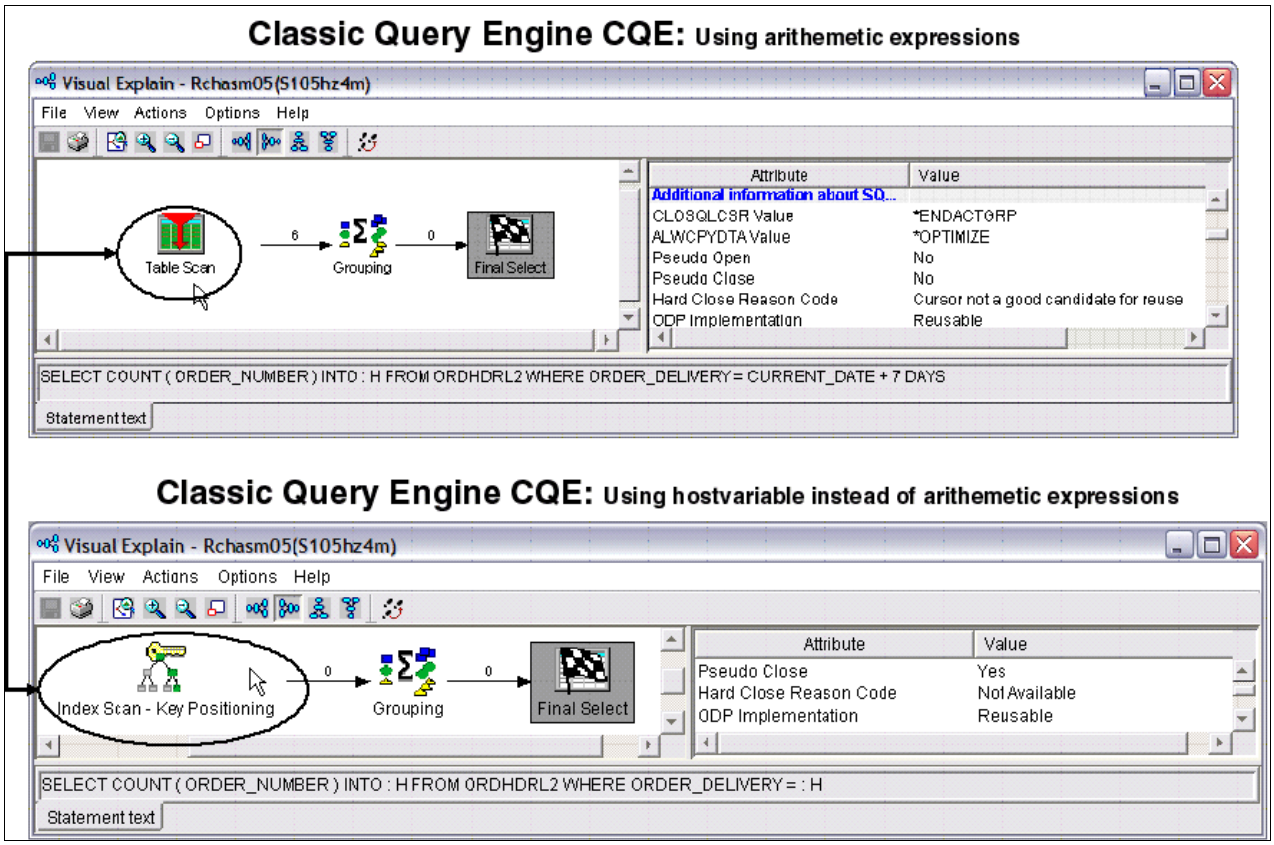


Figure 9-11 Using numeric expressions or host variables in embedded SQL with CQE

When SQE executes the query, in both cases, indexes can be used. But with numeric expressions, an additional relative record scan is necessary. Figure 9-12 shows the Visual Explain for both SQL statements when executed by SQE.

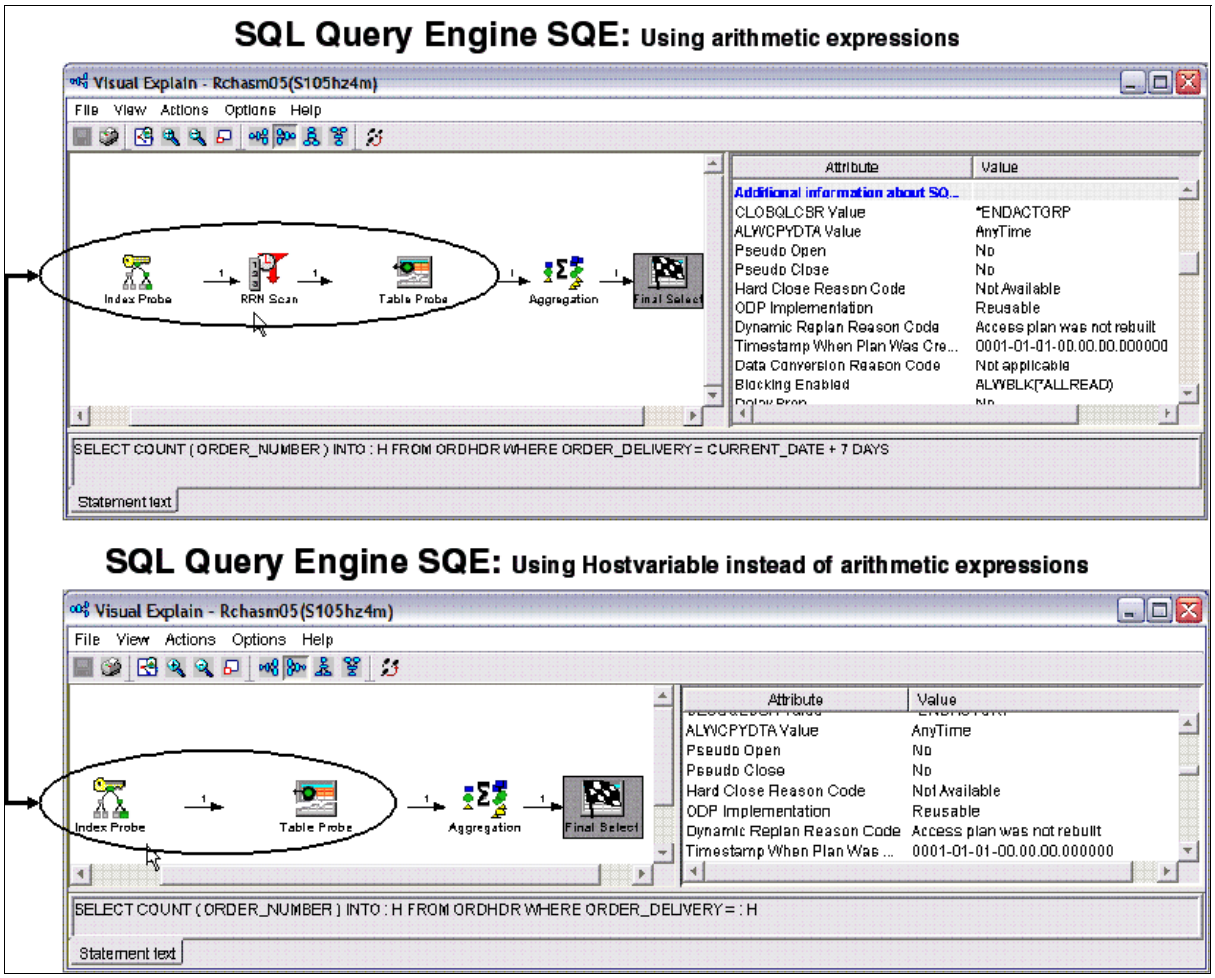


Figure 9-12 Using numeric expressions or host variables in embedded SQL with SQE

9.2.6 Using the LIKE predicate

The percent sign (%) and the underline (_), when used in the pattern of a LIKE predicate, specify a character string that is similar to the column value of rows that 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. However, when used at the beginning of a character string, the % and _ can prevent DB2 Universal Database for iSeries from using any indexes that might be defined in the appropriate column to limit the number of rows scanned using index scan-key positioning.

Figure 9-13 shows the difference when using the LIKE predicate with the % sign in the first position of the character expression.

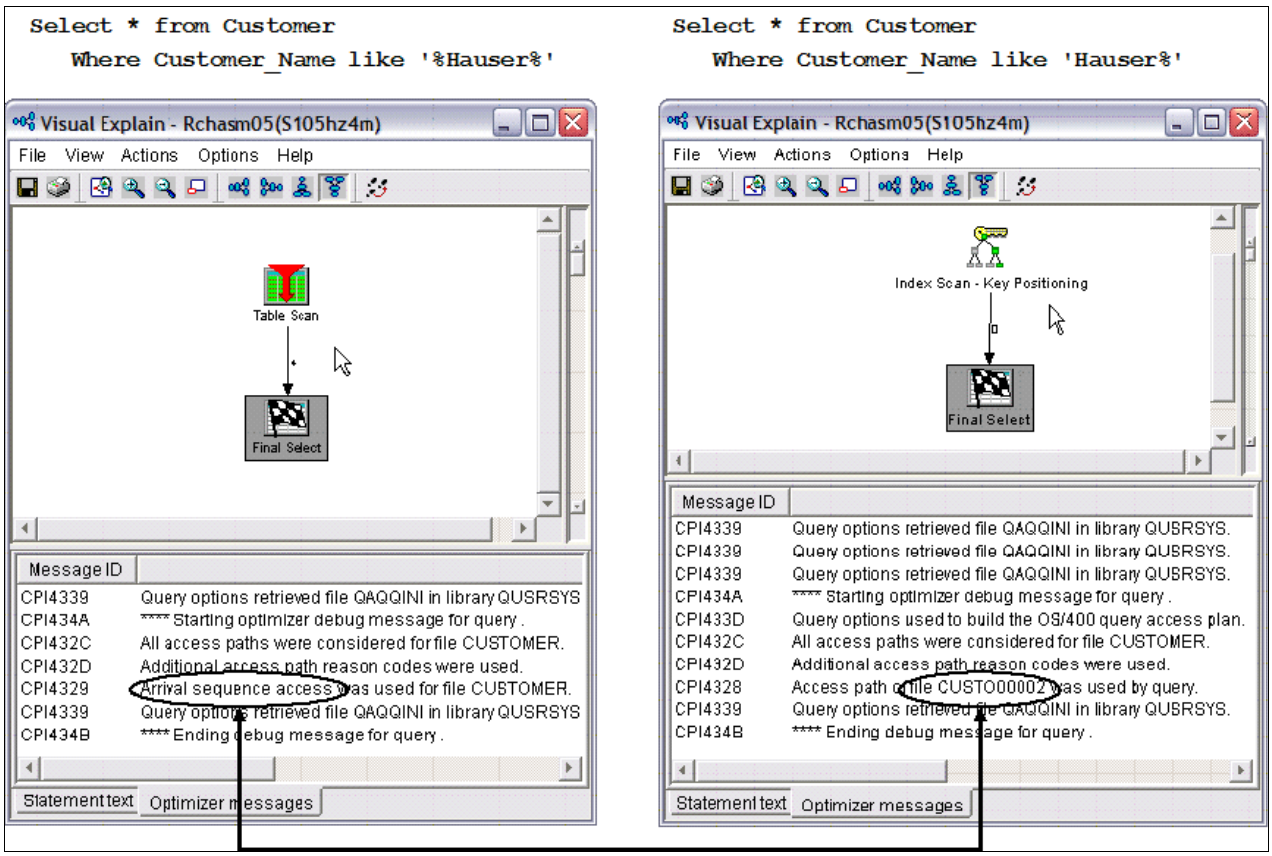


Figure 9-13 Using the LIKE predicate in SQL select statements

Note: In V5R3M0, queries that contain the LIKE predicate to select rows are rerouted to the CQE and cannot benefit from the advantages of the new SQE.

If you search for the first character and want to use the SQE instead of the CQE, you can use the scalar function substring on the left sign of the equal sign. If you have to search for additional characters in the string, you can additionally use the scalar function POSSTR. By splitting the LIKE predicate into several scalar function, you can affect the query optimizer to use the SQE.

Figure 9-14 shows Visual Explain for two select statements that lead to the same result. In the first case, we use the LIKE predicate. In the second case, we use the scalar functions SUBSTR() and POSSTR() instead.

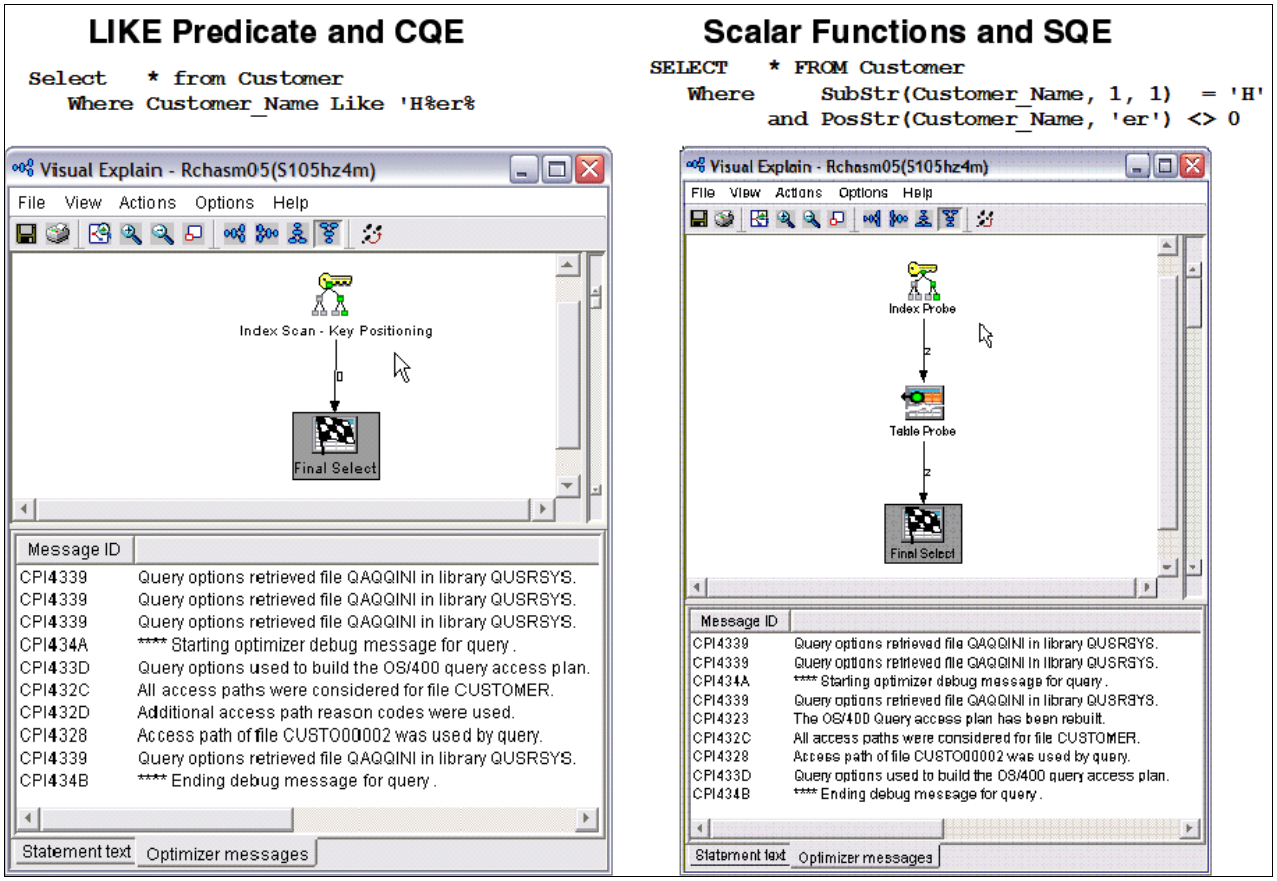


Figure 9-14 Replacing the LIKE predicate with SQL scalar functions

9.2.7 Avoid scalar functions in the WHERE clause

If a scalar function is used in the WHERE clause, the optimizer might not use an appropriate existing index. In some cases, you can affect the optimizer by rewriting your query in a different manner. For example, if you must select all the orders for a specific year or month, use a range and not the scalar functions YEAR or MONTH.

In the following example, all order totals for the year 2005 are cumulated by month. We first used the scalar function YEAR to select the desired year in the WHERE clause. In the second test, we used a date range instead, using the BETWEEN predicate. While in the first example a table scan is performed, in the second example an index is used.

Figure 9-15 shows the Visual Explain diagrams for both SQL statements.

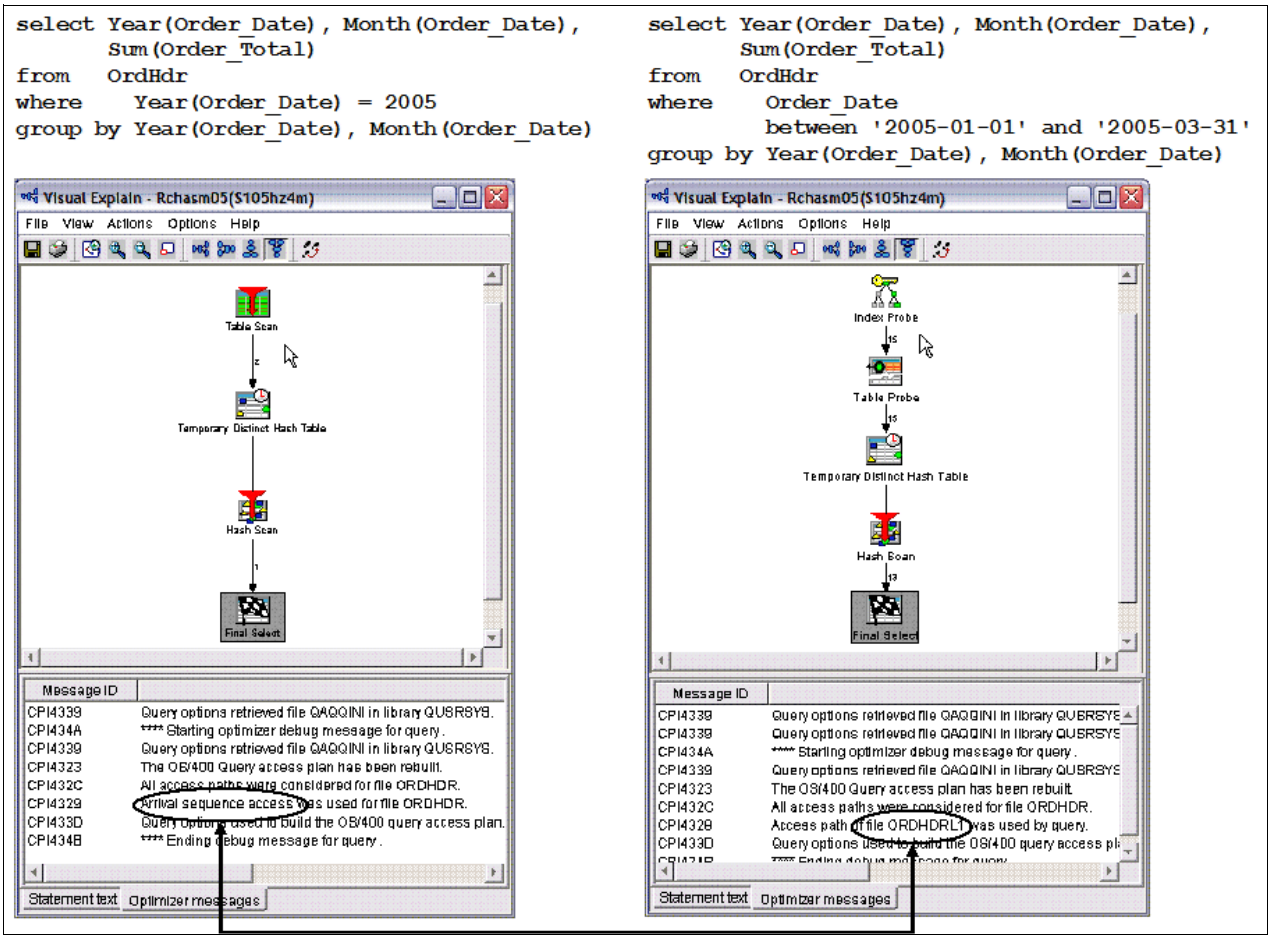


Figure 9-15 Using ranges instead of scalar functions

9.3 Reorganizing your database

With release V5R3M0, there are several enhancements that can help to analyze and reorganize the database objects, tables, and indexes for better performance:

- Index Evaluator
- Improved reorganization support for tables and physical files
- Fast Delete support

9.3.1 Index Evaluator

Because indexes cannot directly be used in SQL select statements, it is difficult to determine which one are obsolete. An index is always created with access path maintenance *IMMED. Therefore, a vast number of indexes can drop down the performance. It is important that you regularly check the obsolete indexes and remove them.

Index Evaluator is a new function that makes it easier to analyze the indexes (and keyed logical files) that are helping your SQL and query performance and the indexes that are not really being used at all. For each index defined over the table, the evaluator returns data about the last time an index was used in the implementation of a query and provides statistics about the table data to the query optimizer. In addition, it returns a count for the number of times the query optimizer used an index object.

In Figure 9-16, you see the output of the Index Evaluator. To initiate it, in iSeries Navigator, right-click a **table object** and select **Indexes**.

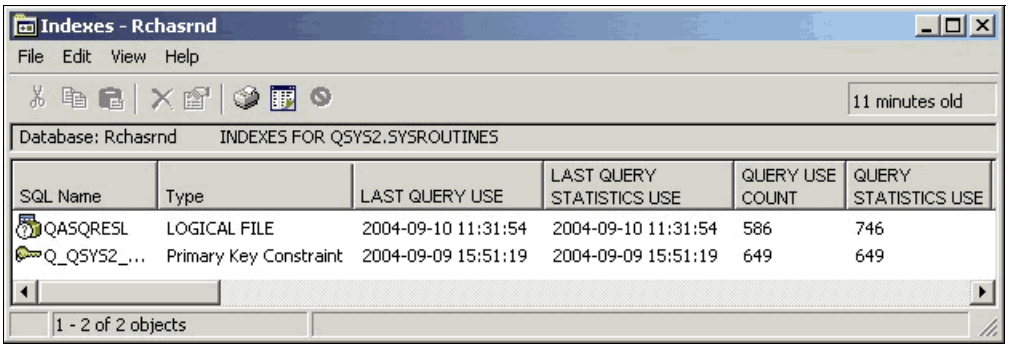


Figure 9-16 Index Evaluator

This new capability requires iSeries Access for Windows V5R3 Service Pack 2 on the client and the PTFs SI12938 and SI12873 installed on the server. The index usage information does not contain any data about queries that were run before these PTFs are applied. Therefore, don't decide on the value of an index before you let the majority of your reports and queries run.

9.3.2 Improved reorganization support for tables and physical files

In the past, many companies could not use the benefits of reorganized tables. To reorganize tables, they must be allocated exclusively. This is a problem if the tables are heavily used and the company works 24/7.

Contrary to the previous releases, you can run the reorganization support online with options that allow read-level, or even update-level, access to rows in the table while the reorganization is running.

In V5R3M0, the following new capabilities are added to the reorganize physical file support:

- Suspended reorganize support
- Online reorganize
- Index rebuild options

Suspended reorganize support

With V5R3, reorganize support can be suspended and resumed later. This is useful for large tables that you might not have reorganized in the past because you didn't have a long enough window to take the table offline to run the reorganization.

This capability lets you start the reorganize support during a time that suits your business needs and then suspend the support when that window expires. You effectively get an incremental reorganization of your table. Then when you have another window available, you can resume the reorganize support from where it left off last time, or you can start from scratch if you made significant changes to the table since you suspended the reorganization. Even this incremental reorganization might increase the efficiency of accessing your table.

Online reorganize

You can also choose whether users can access a table during reorganization (that is, online reorganize). Online reorganize is beneficial to those who need to reorganize a table but simply can't allocate the table exclusively. This gives you two options for allowing user access to the table while reorganize support is running: allow users read-level access or allow them read- and update-level access. Depending on your business needs, these two options can help keep your business running while improving table-access performance.

Keep in mind that reorganization takes longer if you allow users to access the table. Read access allows you to keep running queries or reports while the table is being reorganized. If you choose to allow update access, the reorganize support might be unable to put the rows in the exact key order.

Index rebuilt options

You can choose when (or if) to rebuild indexes built on the table you are reorganizing. This also has implications on the length of time required for the reorganization. You can choose to have the indexes maintained during the reorganization (so a rebuild isn't necessary), after the reorganization or separate from the reorganization.

Having the indexes maintained during the reorganization provides faster access for query users because changes to the index are made simultaneously with the data reorganization. Having the indexes rebuilt at the end provides faster reorganization. However, queries that typically use the indexes are substantially impaired until the indexes are rebuilt, and any applications that depend on the indexes attempt to rebuild the indexes during open. Rebuilding the indexes separate from the reorganization allows the reorganization to be done faster, but the indexes are not rebuilt as part of the reorganization itself. They are rebuilt in the background by a separate system process. All of these new capabilities are available on the Reorganize Physical File Member (RGZPFM) command.

9.3.3 Fast Delete support

As developers move from native I/O to embedded SQL, they often wonder why a Clear Physical File Member (CLRPFM) command is faster than the SQL equivalent of DELETE FROM table. The reason is that the SQL DELETE statement deletes a single row at a time. In i5/OS V5R3, DB2 Universal Database for iSeries has been enhanced with new techniques to speed up processing when every row in the table is deleted.

If the DELETE statement is not run under commitment control and no WHERE clause is specified, then DB2 Universal Database for iSeries uses the CLRPFM operation underneath the covers.

If the Delete is performed with commitment control and no WHERE clause is specified, then DB2 Universal Database for iSeries can use a new method that is faster than the old delete one row at a time approach. The deletions can still be committed or rolled back, but individual journal entries for each row are not recorded in the journal. This technique is only used if *all* of the following statements are *true*:

- The target table is not a view.
- A significant number of rows is being deleted.
- The job issuing the DELETE statement does not have an open cursor on the file (not including pseudo-closed SQL cursors).
- No other job has a lock on the table.
- The table does not have an active delete trigger.
- The table is not the parent in a referential constraint with a CASCADE, SET NULL, or SET DEFAULT delete rule.
- The user issuing the DELETE statement has *OBJMGT or *OBJALTER system authority on the table in addition to the DELETE privilege.

A

Tools to check a performance problem

When a performance problem is occurring, it is important to understand what is happening with the system, even when you know that you are having a problem with SQL queries. There are several commands that you can use to see a high level view of what the system is doing.

Use the tools in this appendix to help you in looking at the big picture. They will help you to try to determine if SQL queries are causing performance problems on the system or if system tuning needs to be done to prevent SQL query performance problems.

WRKACTJOB command

The Work with Active Jobs (WRKACTJOB) command allows you to see what jobs are using system resources. You can sort on any column. Position the cursor over the column that you want to examine and press F16 to sort it. You can sort by CPU% to find the job or jobs that are using most of the CPU.

Figure A-1 shows the Work with Active Jobs display. You can see that the top three jobs are using the majority of CPU. Refresh this display often to see if the jobs continue to use CPU or if it was a one-time occurrence. It is important to note the function for the jobs that are using CPU. You can find the function by looking in the Function column. See if they are building indexes. If indexes are being built, the function shows IDX-*indexname*, where *indexname* is the name of the index being built.

```
                          Work with Active Jobs                      RCHASCLC
                                                              03/04/05  10:13:32
CPU %:    99.8     Elapsed time:   00:00:04     Active jobs:   1300

Type options, press Enter.
  2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
  8=Work with spooled files   13=Disconnect ...

Opt  Subsystem/Job  User        Type  CPU %  Function        Status
       QCLNSYSLOG   QPGMR       BCH    22.5  CMD-DLTPRB       RUN
       QPADEV001V   KLD         INT    18.4  CMD-WRKMEDIBRM   RUN
       QPADEV0035   TSWEENEY    INT    17.5  CMD-DSPLICKEY    RUN
       Q1PDR        QPM400      BCH     4.3  PGM-Q1PBATCH     RUN
       PRTCPTRPT    EILEENPI    BCH     3.7  PGM-QPTBATCH     RUN
       CRTPFRDTA    QSYS        BCH     3.7  CMD-CRTPFRDTA    RUN
       QPADEV004W   DHUFFMAN    INT     1.4  MNU-MAIN         RUN
       QPADEV0018   HANS        INT     1.4  CMD-WRKPRB       RUN
       QRWTSRVR     QUSER       BCI     1.3                   RUN
                                                                More...
Parameters or command
===>
F3=Exit   F5=Refresh       F7=Find      F10=Restart statistics
F11=Display elapsed data   F12=Cancel   F23=More options   F24=More keys
```

Figure A-1 Work with Active Jobs panel showing jobs using CPU

The Work with Active Jobs display can also show jobs that are using a large amount of I/O (see Figure A-2). To view the I/O display, you enter the WRKACTJOB command and then press F11. Then place your cursor in the AuxIO column and press F16 to sort the column.

In the Work with Active Jobs display, the I/O count shown is only reported after an operation has completed. An example of where the WRKACTJOB I/O count for a job might not match the Work with System Activity (WRKSYSACT) count is when a blocked fetch is done. The WRKSYSACT command shows the actual I/O count, where the WRKACTJOB command does not show the I/O count until the fetch has completed.

In this example, using the WRKACTJOB command, a poor performing SQL statement might appear as though it is performing little to no I/O, but the WRKSYSACT command shows that the job is I/O intensive. For more information about the WRKSYSACT command, see “WRKSYSACT command” on page 255.

It is important to press F11 two more times when looking at the I/O with the WRKACTJOB command to reach the display that shows the function for the jobs. If the function is

IDX-*indexname*, then the job is building an index. Further investigation must be done to determine why the job is building an index.

```
                          Work with Active Jobs                      RCHASCLC
                                                                03/04/05  10:27:54
CPU %:    99.8     Elapsed time:   00:14:25     Active jobs:   1302

Type options, press Enter.
  2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
  8=Work with spooled files   13=Disconnect ...
                                          --------Elapsed---------
Opt  Subsystem/Job  Type  Pool  Pty      CPU  Int    Rsp  AuxIO  CPU %
       PRTSYSRPT    BCH     2   50     62.3               29230    1.5
       QSCSTT0001   BCH     2   25   5743.6               22403     .3
       QSCSTT0004   BCH     2   25   6025.5               22346     .3
       QPADEV0048   INT     3   20      4.0  109    2.7  15138     .0
       QPADEV004W   INT     3   20  36282.0    0     .0   6551     .9
       QPADEV003N   INT     3   20      1.5   37    2.8   3733     .0
       QRWTSRVR     BCI     2   20  33036.7               3601     .5
       QPADEV0018   INT     3   20   2747.9    0     .0   3389     .4
       AMQPCSEA     BCH     2   35     12.0               2434     .0
                                                                   More...
Parameters or command
===>
F3=Exit   F5=Refresh      F7=Find      F10=Restart statistics
F11=Display thread data   F12=Cancel   F23=More options   F24=More keys
```

Figure A-2 Work with Active Jobs panel showing I/O used

WRKSYSACT command

WRKSYSACT command is provided with the Performance Tools. This command is helpful in finding jobs that use system resources. The advantage of using this command over the WRKACTJOB command is that the WRKSYSACT command shows the Licensed Internal Code (LIC) tasks in addition to the active jobs in the system.

The WRKSYSACT command, by default, sorts on CPU utilization as shows in Figure A-3. It is important to note the elapsed time. To watch for jobs using CPU, press F10 often to see if the same jobs stay near the top of the list of jobs using CPU. These are the jobs that you want to determine what they are doing.

One way to determine what a job is doing is to look at the function of the job using the WRKACTJOB command as shown in Figure A-1. You can also use the Work with Jobs (WRKJOB) command to see what the job is doing. For more information about the WRKJOB command, see “WRKJOB command” on page 259. If it is known that the jobs using CPU are also using SQL, then you want to look at Database Monitor data or use other tools to try to capture the performance problem.

```
                        Work with System Activity                   RCHASCLC
                                                              03/04/05  10:24:50
Automatic refresh in seconds  . . . . . . . . . . . . . . . .     5
Elapsed time . . . . . . :   00:00:02    Average CPU util . . . . :      99.9
Number of CPUs . . . . . :       4      Maximum CPU util . . . . . :   101.8
Overall DB CPU util  . . . :     7.7     Minimum CPU util . . . . . :    98.8
                                         Current processing capacity:     3.00
Type options, press Enter.
  1=Monitor job   5=Work with job
                                                          Total  Total   DB
     Job or                                               CPU    Sync  Async   CPU
Opt  Task        User        Number  Thread    Pty   Util   I/O    I/O   Util
     QCLNSYSLOG  QPGMR       640427  00000088   10   24.3      0      0    .0
     QPADEV0035  TSWEENEY    639636  00000002   23   22.4      0      0    .0
     QPADEV001V  KLD         640239  0000008A   20   21.7      0      0    .0
     Q1PDR       QPM400      640481  00000416   50    4.9      0      0   7.1
     CRTPFRDTA   QSYS        640539  00000029   50    4.7      0      0    .0
     PRTCPTRPT   EILEENPI     641086  00000022   50    4.5      0      0    .0
     QPADEV0015  PEGGYCL      641174  00000016    1    1.7      4      0    .0
     QPADEV004W  DHUFFMAN     640056  00000040   23    1.5     28      0    .5
                                                                      More...
F3=Exit   F10=Update list   F11=View 2   F12=Cancel   F19=Automatic refresh
F24=More keys
```

Figure A-3 Work with System Activity panel sorted by CPU

The WRKSYSACT command can also sort on different resources, such as I/O. You can resequence the list by selecting F16. Then you see the Select Type of Sequence display (see Figure A-4).

```
                      Select Type of Sequence

 Type option, press Enter.

   Option  . . . .   2  1. Sequence by CPU
                        2. Sequence by I/O
                        3. Sequence by net storage
                        4. Sequence by allocated storage
                        5. Sequence by deallocated storage
                        6. Sequence by database CPU
                        7. Sequence by total waiting time



 F12=Cancel
```

Figure A-4 Sequence options for the WRKSYSACT command

In this example, we type option 2 to sort by I/O. Then you see the Work with System Activity display shown in Figure A-5. It is important to refresh the display with F10 often to see if the same jobs are doing a lot of I/O. It is also good to notice if the I/O is synchronous or asynchronous. In most cases, asynchronous I/O is more desirable. *Asynchronous disk I/O* means the job can do other work while waiting for disk I/O to complete. *Synchronous disk I/O* is when a job has to wait for disk I/O work to be done before doing other work.

When you know the job that is using I/O, then determine what the job is doing. You can use the WRKACTJOB command to find the job and see what function the job is in at the time as

shown in Figure A-1 on page 254. You can also use the WRKJOB command to see what the job is doing. For more information about the WRKJOB command, see “WRKJOB command” on page 259.

If SQL is being done, then Database Monitor data can determine what is being done. For more information about gathering Database Monitor data, see Chapter 4, “Gathering database SQL performance data” on page 51.

```
                        Work with System Activity                   RCHASCLC
                                                          03/04/05  10:24:50
Automatic refresh in seconds  . . . . . . . . . . . . . . . . . .     5
Elapsed time . . . . . . :   00:00:02    Average CPU util . . . . :      99.9
Number of CPUs . . . . . . :       4     Maximum CPU util . . . . . :   101.8
Overall DB CPU util  . . . :     7.7     Minimum CPU util . . . . . :    98.8
                                         Current processing capacity:     3.00
Type options, press Enter.
  1=Monitor job   5=Work with job
                                                                Total  Total   DB
     Job or                                                      CPU   Sync  Async   CPU
Opt  Task        User        Number  Thread    Pty   Util   I/O    I/O   Util
     SMP00001                                    0    1.0     0   1786    .0
     GLIDDEN51E  QEJBSVR     620898  0000000C    0     .3   173      0    .0
     LTWAS51ND   QEJBSVR     632143  00002D98   26     .4   144      0    .0
     PRTSYSRPT   EILEENPI    641085  00000089   50    1.3    46     91    .0
     QYPSJSVR    QYPSJSVR    624351  0000002C    0     .2   131      0    .0
     GNETSIZ     AJMAC       639853  00000027   50    1.1     3    124    .0
     NODEAGENT   QEJBSVR     621978  00000015    0     .2   125      0    .0
     MICK51EXP   QEJBSVR     636665  000000D2    0     .2   115      0    .0
                                                                        More...
F14=Display jobs only   F15=Display tasks only   F16=Resequence
F24=More keys
```

Figure A-5 Work with System Activity panel sorted by I/O

WRKSYSSTS command

The Work with System Status (WRKSYSSTS) command provides a global view of the system. You can get a better view if you press F21 and select the advanced assistance level. You can press F10 to restart the statistics. It is best to look at the data after a couple of minutes have elapsed. F5 refreshes the display until you see 00:02:00.

Figure A-6 shows a Work with System Status display. When you have a performance problem, check the Work with System Status display to see if your *% CPU used* is higher than you normally run. You can also look at the pools to see if you have abnormally high faulting or paging.

What do you do when % CPU used seems high? One item to check is the *% DB Capability.* If the % DB Capability is also high, then it is an indication that there SQL activity is occurring. Use the WRKSYSACT command to display the jobs using CPU. See “WRKSYSACT command” on page 255 for more information.

- If a high priority job (low number) is using a lot of CPU, greater than 50% for an extended period of time, then the job can cause the entire system to have poor response times. If it is found that one or a few jobs are using the majority of CPU, then ask:
  - Is the priority of the job really appropriate?
  - Is the job running in the correct environment? For example, if the job is interactive, is it be better suited running in batch?
  - What is the job doing?
- If the CPU utilization is high, greater than 80%, and all jobs seem to have an equal but small CPU percent, this can mean that there are too many active jobs on the system.

```
                          Work with System Status                   RCHASCLC
                                                                 03/04/05  13:52:55
% CPU used . . . . . . . :        2.7    System ASP . . . . . . . :     1922 G
% DB capability  . . . . :         .0    % system ASP used  . . . :    88.2414
Elapsed time . . . . . . :   00:02:00    Total aux stg  . . . . . :     2055 G
Jobs in system . . . . . :      14182    Current unprotect used . :     6944 M
% perm addresses . . . . :       .202    Maximum unprotect  . . . :     7476 M
% temp addresses . . . . :       .910

Sys      Pool   Reserved    Max  ----DB-----  --Non-DB---  Act-   Wait-  Act-
Pool    Size M   Size M     Act  Fault Pages  Fault Pages  Wait   Inel   Inel
  1     788.06    227.82  +++++     .0    .0     .9   1.3  18.3     .0     .0
  2    2754.35      2.17   9514     .0    .8   11.4  42.8  2148     .0     .0
  3    4785.78       .00    233     .0    .0    4.1   4.8  32.8     .0     .0
  4      44.91       .00     11     .0    .0     .0    .0    .0     .0     .0
  5      94.20       .00      5     .0    .0     .0    .0    .0     .0     .0
  6       1.25       .00      6     .0    .0     .0    .0    .0     .0     .0
  7      94.20       .00     24     .0    .0     .0    .0    .0     .0     .0
  8      94.20       .00      5     .0    .0     .0    .0    .0     .0     .0
  9     763.33       .17     23     .0    .0     .0    .0  83.5     .0     .0
                                                                      Bottom
===>
F21=Select assistance level
```

Figure A-6 Work with System Status panel

For more information about the WRKSYSSTS command and how to view the data via iSeries Navigator, see *Managing OS/400 with Operations Navigator V5R1 Volume 5: Performance Management*, SG24-6565.

WRKOBJLCK command

Using the Work with Object Lock (WRKOBJLCK) command on user profiles can help narrow down clients that are having performance problems. For example, a user is using an ODBC connection and is complaining about having a performance problem. To find the job that the user is running on the iSeries, enter the following command:

```
WRKOBJLCK OBJ(QSYS/userprofile) OBJTYPE(*USRPRF)
```

In this example, *userprofile* refers to the user's iSeries user ID.

A panel is displayed that shows a list of jobs that the user profile has locked. You can work with each job to see if you can isolate the one that is having the problem. You can look at the call stack and objects locked to see a picture of what the job is doing.

If any program beginning with QSQ is found in the call stack or QDBGETMQO is found, then SQL is being used. Refer to Chapter 3, "Overview of tools to analyze database performance"

on page 31, to learn about other tools that you can use to further analyze the job, after the correct job is found.

WRKJOB command

You can use the Work with Job (WRKJOB) command to determine what a job is doing. It includes options to look at the job. Some of the following options might be helpful to check for jobs that have performance problems.

- Use option 1 to find the pool in which the job is running. The subsystem and subsystem pools are shown:

```
An example is:
Subsystem . . . . . . . . . . . . . . . . . :   QINTER
  Subsystem pool ID . . . . . . . . . . . . :     2
```

  To find the system pool in which the job is running, enter the Work with Subsystems (WRKSBS) command.

  Figure A-7 shows the Work with Subsystems display for our example. As you can see, the *subsystem pool ID* is 2 and the subsystem is QINTER. We go to subsystem QINTER and look under the subsystem pool column of 2, which shows the job is using *system pool 3*.

```
                           Work with Subsystems
                                                          System:   RCHASCLC
Type options, press Enter.
  4=End subsystem   5=Display subsystem description
  8=Work with subsystem jobs

                      Total     -----------Subsystem Pools------------
Opt  Subsystem     Storage (M)   1   2   3   4   5   6   7   8   9  10
     QASE5                .00   2                           8
     QASE51               .00   2   9
     QBATCH              1.25   2                               6
     QCMN                 .00   2
     QCTL                 .00   2
     QEJBAS51             .00   2
     QHTTPSVR             .00   2
     QINTER               .00   2   3   4                           5
     QMQM                 .00   2
     QSERVER              .00   2
                                                                  More...
Parameters or command
===>
F3=Exit   F5=Refresh   F11=Display system data   F12=Cancel
F14=Work with system status
```

Figure A-7 Work with Subsystems panel

- Use option 3 to give the job run attributes, if the job is active. This option is helpful in determining how much CPU and temporary storage are being used in the job. High temporary storage use can be the result of temporary indexes being built.
- Use option 4 to see all the spooled files for the job. Check this to see if any unexpected spooled files are being created.
- Use option 10 to see the job log for the job. It is important to review the job log. Chapter 3, “Overview of tools to analyze database performance” on page 31, explains how to turn on debug messages to capture optimizer messages when running SQL.

- Use option 11 to view the call stack for the job. The example in Figure A-8 shows QDBGETMQO at the bottom of the call stack. QDBGETMQO is when the SQL Query Engine (SQE) is getting rows. SQE is discussed in Chapter 2, “DB2 Universal Database for iSeries performance basics” on page 9.

  If QDBGETM is displayed at the bottom of the call stack, either the Classic Query Engine (CQE) is fetching rows or there is native I/O such as in RPG. It is important to note any user programs that are at the bottom of the call stack. If a user program is found, use the Print SQL Information (PRTSQLINF) command to see if the program contains SQL. For details about PRTSQLINF, see Chapter 3, “Overview of tools to analyze database performance” on page 31.

  Use F10 to monitor whether the programs in the call stack change. If any program beginning with QSQ is found in the call stack or QDBGETMQO is found, then SQL is being used.

```
                           Display Call Stack
                                                          System:   RCHASCLC
Job:   QPADEV0027     User:   PEGGYCL        Number:   647778

Thread:   00000046

       Program
Rqs    or
Lvl    Procedure   Library     Statement        Instruction
       QSQIMAIN    QSQL                           05CA
       QSQISE      QSQL                           0707
       QQUDA       QSYS                           03CD
       QQURA       QSYS                           0087
       QQURB       QSYS                           0677
       QDBGETMQO   QSYS        0000002573

                                                                      Bottom
F3=Exit        F10=Update stack   F11=Display activation group   F12=Cancel
F16=Job menu   F17=Top   F18=Bottom   F22=Display entire name
```

Figure A-8 Display Call Stack panel, when using option 11 from the WRKJOB command

iDoctor for iSeries Job Watcher

For a more in depth analysis of a performance problem, you can use the advanced analysis tool called *iDoctor for iSeries Job Watcher*, which we refer to as Job Watcher. Job Watcher is made up of two parts:

- Tools for collecting data
- Tools for analyzing and viewing the collected data

A typical situation for deciding to use the Job Watcher is for a job that is taking a long time to run but is hardly using any CPU resource and disk I/Os are not particularly excessive. Job Watcher is an excellent tool to help you determine job waits, seizes, and other types of contention. Identifying why a job or multiple jobs or threads are “not doing anything when they should be,” is a primary situation to demonstrate a key set of Job Watcher capabilities.

Job Watcher returns near real-time information about a selected set of jobs, threads, LIC tasks, or all three. It is similar in sampling function to the WRKACTJOB and WRKSYSACT system commands, where each refresh computes delta information for the ending snapshot interval. In Job Watcher, these refreshes can be set to occur automatically, even as frequently

as every 5 seconds. Better yet, Job Watcher harvests the data from the jobs, threads, or tasks being watched in a manner that does not impact other jobs on the system while it is collecting.

Job Watcher collected data includes the following information among other details:

- Standard WRKSYSACT type information
  - CPU
  - DASD I/O breakdown
  - DASD space consumption
  - For jobs or threads, the user profile under which the job or thread is running

    For prestart server jobs that were started under user profile QUSER, you can see the user profile that is currently being serviced by that job/thread, rather than QUSER.
- Expanded details on types of waits and object lock or seize conditions
- Last run SQL statements syntax
- Program or procedure call stack, 1000 levels deep

You can download Job Watcher from the following Web site:

`http://www.ibm.com/eserver/iseries/support/i_dir/idoctor.nsf`

Select downloads from the left pane and then select the appropriate download option. You can sign up for a 45-day trial to use the product.

For further information about using Job Watcher, see the *IBM iDoctor iSeries Job Watcher: Advanced Performance Tool*, SG24-6474.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 264. Note that some of the documents referenced here may be available in softcopy only.

- *Advanced Functions and Administration on DB2 Universal Database for iSeries*, SG24-4249
- *IBM iDoctor iSeries Job Watcher: Advanced Performance Tool*, SG24-6474
- *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503
- *Managing OS/400 with Operations Navigator V5R1 Volume 5: Performance Management*, SG24-6565
- *Preparing for and Tuning the V5R2 SQL Query Engine on DB2 Universal Database for iSeries*, SG24-6598
- *Using AS/400 Database Monitor and Visual Explain To Identify and Tune SQL Queries*, REDP-0502

Other publications

These publications are also relevant as further information sources:

- *DB2 Universal Database for iSeries Database Performance and Query Optimization*

  http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzajq/rzajqmst.htm
- *Star Schema Join Support within DB2 UDB for iSeries - Version 3*

  http://www-1.ibm.com/servers/enable/site/education/abstracts/16fa_abs.html

Online resources

These Web sites are also relevant as further information sources:

- Information Center

  http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp
- iDoctor for iSeries Job Watcher

  http://www.ibm.com/eserver/iseries/support/i_dir/idoctor.nsf

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

SQL Performance Diagnosis on IBM DB2 Universal Database for iSeries

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

SQL Performance Diagnosis

on IBM DB2 Universal Database for iSeries

Discover the tools to identify SQL performance problems

Unleash the capabilities of the SQL Performance Monitors

Learn to query the Database Monitor performance data

The goal of database performance tuning is to minimize the response time of your queries. It is also to optimize your server's resources by minimizing network traffic, disk I/O, and CPU time.

This IBM Redbook helps you to understand the basics of identifying and tuning the performance of Structured Query Language (SQL) statements using IBM DB2 Universal Database for iSeries. DB2 Universal Database for iSeries provides a comprehensive set of tools that help technical analysts tune SQL queries. The SQL Performance Monitors are part of the set of tools that IBM i5/OS provides for assisting in SQL performance analysis since Version 3 Release 6. These monitors help to analyze database performance problems after SQL requests are run.

This redbook also presents tips and techniques based on the SQL Performance Monitors and other tools, such as Visual Explain. You'll find this guidance helpful in gaining the most out of both DB2 Universal Database for iSeries and query optimizer when using SQL.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6654-00 ISBN 0738497487