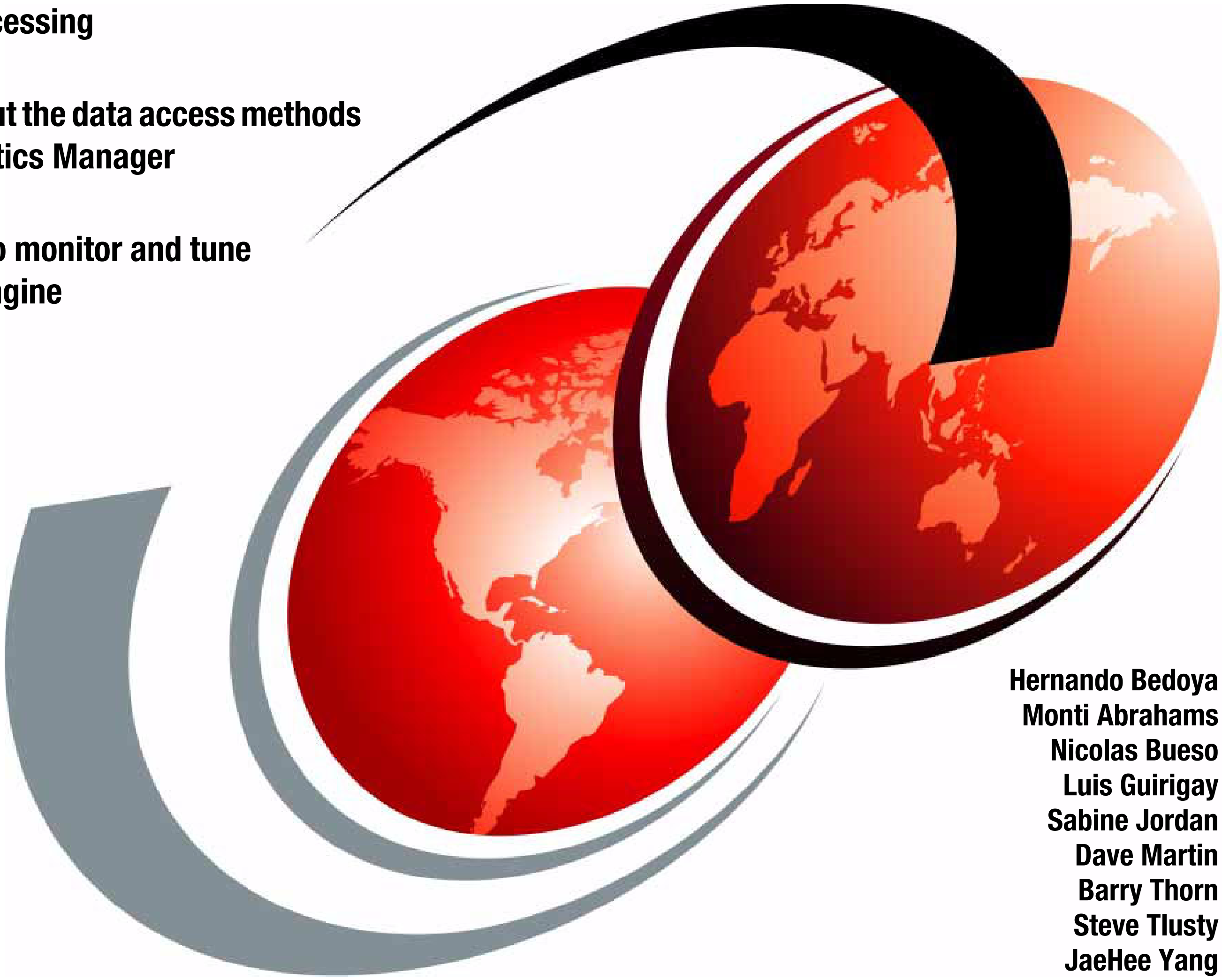
IBM

Preparing for and Tuning the SQL Query Engine on DB2 for i5/OS

Understand the new architecture for query processing

Learn about the data access methods and Statistics Manager

See how to monitor and tune the SQL engine



Hernando Bedoya
Monti Abrahams
Nicolas Bueso
Luis Guirigay
Sabine Jordan
Dave Martin
Barry Thorn
Steve Tlusty
JaeHee Yang

Redbooks

ibm.com/redbooks

International Technical Support Organization

Preparing for and Tuning the SQL Query Engine on DB2 for i5/OS

September 2006

SG24-6598-01

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

Second Edition (September 2006)
This edition applies to Version 5, Release 4 of IBM i5/OS, Program 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 illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

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

- Ascendant®
- AS/400®
- Domino®
- DB2 Universal Database™
- DB2®
- eServer™
- IBM®
- iSeries™
- i5/OS®
- Lotus Workflow™
- Lotus®
- Net.Data®
- OS/400®
- PartnerWorld®
- QuickPlace®
- Redbooks™
- Redbooks (logo) ™
- Sametime®
- System i™
- System i5™
- System/36™
- System/38™
- WebSphere®

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.

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

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

Preface

The IBM® System i™ family, which encompasses the IBM AS/400®, eServer™ iSeries™, eServer i5, and System i5™, has a successful 24-year history of satisfying hundreds of customers' requirements, thanks to its developers. DB2® for i5/OS®, also known as DB2 Universal Database™ for iSeries, is deeply rooted in the architecture and heritage of the AS/400 and its predecessor System/38™.

The database has undergone significant changes over the years to maintain its competitive advantage. However, with an increasing demand for change, IBM has made fundamental changes to the structure of the database to compete successfully in the industry. In doing so, IBM Rochester launched a project to re-engineer and redesign important components of the database. The goal was to form the base for the future development of the product. This made the product easier to maintain, enhance, and provide far better performance. The redesigned components were architected using object-oriented design concepts and then implemented using object-oriented implementation techniques.

The *query optimizer* was one of the key components that was redesigned. This IBM Redbook gives a broad understanding of the architectural changes of the database regarding the query optimizer. It explains the following concepts:

- The architecture of the query optimizer
- The data access methods used by the query optimizer
- The Statistics Manager included in V5R2
- The feedback message changes of the query optimizer
- Some performance measurements
- Changes and enhancements made in V5R4

The objective of this redbook is to help you minimize any SQL or database performance issues when upgrading to OS/400® V5R2 or V5R3 or to IBM i5/OS V5R4. Prior to reading this book, you should have some knowledge of database performance and query optimization.

Important: This edition covers DB2 Universal Database for iSeries on OS/400 V5R2 and V5R3, as well as DB2 for i5/OS V5R4. In general, throughout this book, we use the name *DB2 for i5/OS* to refer to all versions, unless specifically noted.

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 Information Technology (IT) Specialist at the ITSO, in Rochester, MN. He writes extensively and teaches IBM classes worldwide in all areas of DB2 for i5/OS. Before joining the ITSO, he worked in IBM Colombia as an AS/400 IT Specialist doing presales support for the Andean countries. He has 18 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, with an emphasis in database. His areas of expertise are database technology, application development, and data warehousing.

Nicolas Bueso Quan is an iSeries presales senior Certified I/T Specialist at IBM Brazil in Field Technical Sales Support. His main focus includes server consolidation, system management, performance, logical partition (LPAR), and solution design in iSeries. He has worked for IBM for 15 years. He has been working with the System i platform since its introduction, having previously worked on Systems/36 and Systems/38. Nicolas has worked with many of the largest System i customers providing presales support and consulting across Brazil.

Monti Abrahams is a Senior IT Specialist for IBM South Africa, based in Cape Town. He is an IBM Certified Solutions Expert: DB2 Content Manager On Demand and a Certified SAP Technical Consultant with more than 15 years experience in OS/400 and DB2 for i5/OS. Monti provides technical support and consulting to System i clients throughout South Africa and Namibia. He also conducts training courses for IBM IT Education Services. Monti has coauthored several IBM Redbooks™ on various System i topics.

Luis Guirigay is an IT Specialist with Ascendant® Technology LLC, an IBM Premier Business Partner based in Austin, TX. Luis has more than seven years of IT experience in Lotus® Domino® and related products, such as Sametime®, QuickPlace®, and Lotus Workflow™, and two years of experience with the System i platform. He is a Dual IBM Certified Advanced Professional in Application Development and System Administration for Notes and Domino 5, 6, and 7 and an IBM Certified Specialist for Collaborative Solutions. Most recently, Luis has been focusing on projects that involve WebSphere® Portal. He has also coauthored other IBM Redbooks that are soon to be released. He holds a Bachelor of Science degree in Computer Engineering from Rafael Belloso Chacin University, Venezuela.

Sabine Jordan is a Technical Support Specialist with IBM, based in Germany. She has more than 12 years of experience in working with the System i platform. Her areas of expertise include Domino on iSeries, database performance, and porting database applications from other databases to DB2 for i5/OS. She has written papers and given presentations on System i database performance. She has also worked with customers and independent software vendors (ISVs) on their specific performance or porting questions the DB2 for i5/OS area.

Dave Martin is a Certified IT Specialist in Advanced Technical Support (ATS) who started his IBM career in 1969 in St. Louis, Missouri, as a Systems Engineer. From 1979 to 1984, he taught System/34 and System/38 implementation and programming classes to customers and IBM representatives. In 1984, he moved to Dallas, Texas, where he has provided technical support for S/36, S/38, AS/400, and iSeries in the areas of languages, operating systems, systems management, availability and recovery, and most recently Business Intelligence. He is frequently speaks at COMMON User Group and iSeries Technical Conferences.

Barry Thorn is a Certified Consulting IT Specialist with the Enterprise Systems Group of IBM United Kingdom, providing technical support within Europe, Middle East, and Asia (EMEA). He has 32 years of IT experience in IBM, including 14 years of working with the System i family. His areas of expertise include Business Intelligence and data warehouse implementation, database technology, database performance, and database migration. He has written papers and presentations and runs classes on iSeries Business Intelligence, data warehouse implementation, and database. He has also been involved in several projects that require the migration of a database to DB2 on the iSeries server.

Steve Tlusty is a Staff Software Engineer with IBM in Rochester, MN. He has been with IBM for seven years. For the past year, has been working in Enterprise Resource Planning (ERP) development. Before joining the Rochester lab, he was an IT Specialist for IBM Global Services. His projects included supporting manufacturing systems in Rochester, MN, and working with IBM Clients on a variety of Business Intelligence projects.

JaeHee Yang is an iSeries IT Specialist with IBM, based in Korea. She has more than 12 years of experience in the working with the System i family. She has nine years of experience in AS/400 system applications development and two years of experience in database replication between various database. JaeHee currently supports database and applications on the System i platform in Korea and provides general System i technical support.

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

Michael Cain
Kent Milligan
Jarek Miszczyk
PartnerWorld® for Developers, IBM Rochester

Abdo Abdo
Robert Andrews
Robert Bestgen
Curt Boger
Dave Carlson
Robert Driesch
Randy Egan
Michael Faunce
Kevin Kathmann
Chuck King
Doug Mack
Tom McKinley
Paul Nelsestuen
Jerri Ostrem
Kurt Pinnow
Carol Ramler
Tom Schrieber
Jeff Tenner
Ulrich Thiemann
Denise Tompkin
iSeries Development, IBM Rochester

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 will team with IBM technical professionals, Business Partners and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will 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 IBM Redbooks to be as helpful as possible. Send us your comments about this or other IBM 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 Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

1

Overview of DB2 for i5/OS

Important: This edition covers DB2 Universal Database for iSeries on OS/400 V5R2 and V5R3, as well as DB2 for i5/OS V5R4. In general, throughout this book, we use the name *DB2 for i5/OS* to refer to all versions, unless specifically noted.

Data is the key. Quick and reliable access to business data is critical for making crucial business decisions. However, the data needs to be turned into useful information to make these business decisions. It is the job of the database management system (DBMS) to provide and manage this process.

A robust 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. DB2 for i5/OS is well known for being a DBMS that fully incorporates these capabilities and functionality.

In August 2002, IBM introduced the newly designed architecture for DB2 Universal Database for iSeries with the delivery of OS/400 V5R2. This is regarded by many as one of the most important OS/400 release updates since V3R6. This new architecture included additional new functionality for Structured Query Language (SQL) query processing, independent auxiliary storage pools (ASPs), and New Transaction Services (NTS). This laid the foundation for exciting future developments with the ability to respond to new demands quickly and effectively.

In this chapter, we explain the rationale behind the re-engineering of DB2 Universal Database for iSeries. We explore the history and evolution of the database. We discuss modern application development trends, as well as business and technical trends that demand this architecture in DB2 for i5/OS.

Also this this chapter, we discuss the various options considered in redeveloping the database, the path IBM chose, and the reasons behind that choice. Plus, we present an overview of the SQL Query Engine delivered with DB2 Universal Database for iSeries starting in OS/400 V5R2 plus the new benefits added in V5R3 and V5R4.

1.1 The history of DB2 for i5/OS

DB2 for i5/OS is a member of the DB2 family. Integrated into OS/400 and i5/OS, it has its roots in the integrated relational database of the IBM System/38, the predecessor of the AS/400. Applications for this system were written almost exclusively in high-level languages. 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. The SQL objects in DB2 for i5/OS include industry standard objects for relational databases. The objects consist of tables, views, indexes, cursors, triggers, procedures, and schemas. SQL provides an alternative method for accessing data, thereby complementing the existing native file access methods. This helps protect a customer's investment by allowing applications that use native file access and SQL to coexist and access the same data. This coexistence also allows customers and independent software vendors (ISV) to migrate their existing native file access applications to SQL and to write new applications using SQL.

As mentioned earlier, the DB2 for i5/OS engine is integrated into OS/400 and i5/OS. It uses and depends on the operating system to provide such services as locking, security, archive/backup, buffer management, and transaction. The operating system, in turn, uses and relies on the database engine to provide certain services. Figure 1-1 shows a high level overview of the structure of DB2 for i5/OS.

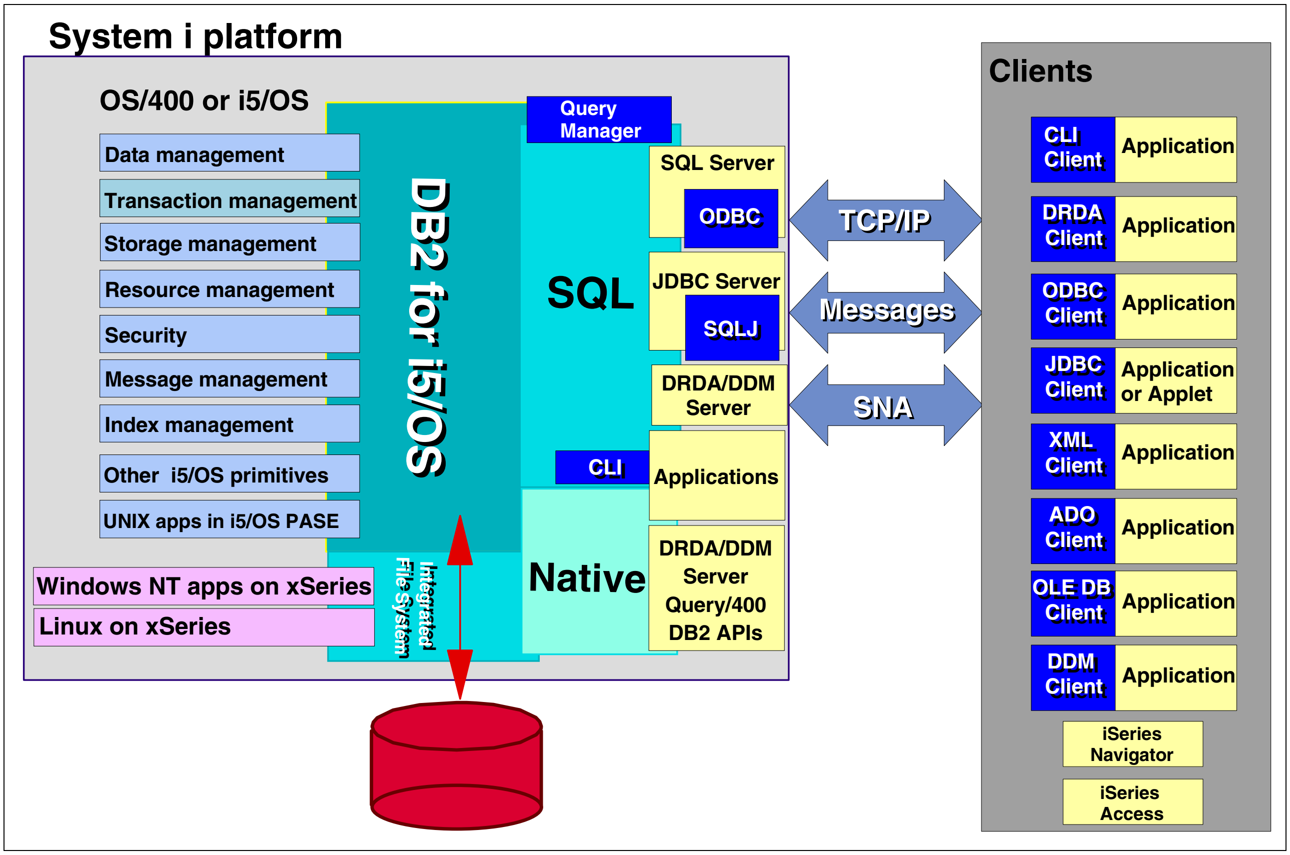


Figure 1-1 The structure of DB2 for i5/OS

The same interfaces that manage the system are employed to manage the database. Additional interfaces that are required to manage the database are provided as natural extensions to the interfaces provided by the operating system. These interfaces make it unnecessary for a full-time database administrator (DBA) to manage the relatively simple database tasks on the System i platform.

Focusing primarily on online transaction processing (OLTP) applications, the database has satisfied customer requirements for well over 20 years. Many of the original System/38 applications, along with applications developed for other systems such as the System/36™, are still in productive use around the world today.

More recently a different breed of applications started to dominate development efforts. These applications are designed to accommodate rapidly changing business needs and processes. Examples of such initiatives include:

- Internet-driven electronic commerce for business-to-business (B2B) and business-to-consumer (B2C) transactions
- Management information reporting to support Business Intelligence (BI) requirements, Customer Relationship Management (CRM), and integrated Supply Chain Management (SCM) solutions
- Enterprise Resource Planning (ERP) and Enterprise Application Solutions (EAS) applications to extend or replace the traditional back-office OLTP applications that are currently supporting the basic business processes

1.2 The ‘new’ application world

The breed of applications referred to earlier has characteristics that include:

- Two or three-tier implementation across multiple, often heterogeneous, servers

  For example, a typical application model today logically, and often physically, separates the functions of a client user interface, application serving, and database serving.

- Database access using SQL

  This is typically complex in nature, making full use of such capabilities as subqueries, unions, and joins at multiple levels, each with their own complex predicates.

- Automatic generation of a dynamic SQL statement from within an application or tool
- Use of dynamic SQL
- Complex transactions with requirements for:
  - Many transactions to use a single connection to the database
  - One transaction to use multiple connections to the database
  - Sophisticated control of transactional integrity to ensure optimum security and ease of recovery for the user
- A potential end-user population of thousands, often using browser technology and requiring excellent scalability to ensure good performance

Figure 1-2 shows an example of an application architecture that is becoming more prevalent in its use. As shown, it is common to find the application spread across more than three tiers.

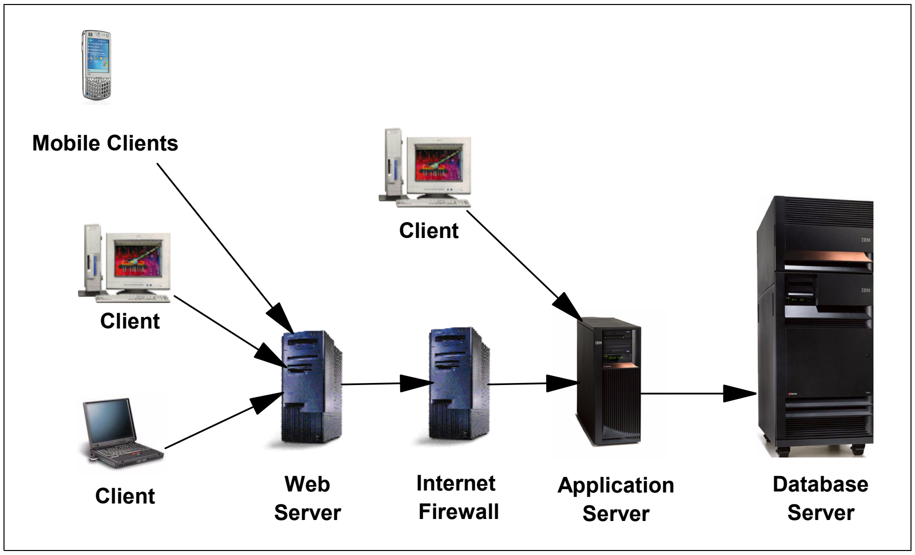


Figure 1-2 The new application world

To keep up with changing application paradigms, while maintaining the DB2 Universal Database for iSeries value proposition, for OS/400 V5R2, the IBM development team explored new methods to deliver increased performance and functionality without increasing development costs.

1.3 The database design

To address the issues and satisfy the demands of the “new” application world, IBM considered the following options for the database design of DB2 Universal Database for iSeries for OS/400 V5R2:

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

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

At that time, acquiring a new database technology would compromise some of 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 on only those aspects of the product for which re-engineering offered the greatest potential, that is the potential that 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 DB2 Universal Database for iSeries self-managing value proposition

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 re-design of the query engine and its components.

1.4 The implementation

It is always an attractive proposition to deliver maximum functionality over the shortest possible time frame. However, this can pose a risk to customers' existing production applications, as well as other DB2 for i5/OS interfaces and other products. With this in mind, IBM decided to deliver these enhancements using a phased implementation approach. This minimizes the risk of disruption to our customers' existing production environments.

As a result of this phase-in approach, two separate query engines coexist in the same system. The redesigned query engine in DB2 for i5/OS is the *SQL Query Engine* (SQE). The existing query engine is referred to as the Classic Query Engine (CQE).

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.

Figure 1-3 shows this staged approach for adding new functionality to the product according to any available version. The OS/400 V5R2 General Availability (GA) release incorporated SQE with specific functionality. An optional PTF, SI07650, was made available after GA of V5R2 to give customers the choice to increase the amount of SQL requests that could be processed and executed by the SQL Query Engine.

Refer to Informational APAR II13486 for the latest information about the SQE functionality and a list of the PTFs that provide this functionality. You can access this Informational APAR and other SQE-related material on the Web at the following address:

http://www.iseries.ibm.com/db2/sqe.html

Important: Only one single interface into the query optimizer exists from the perspective of the user application. It is not possible for a user application to select either SQE or CQE to process the query request. No additional coding is required in the application program.

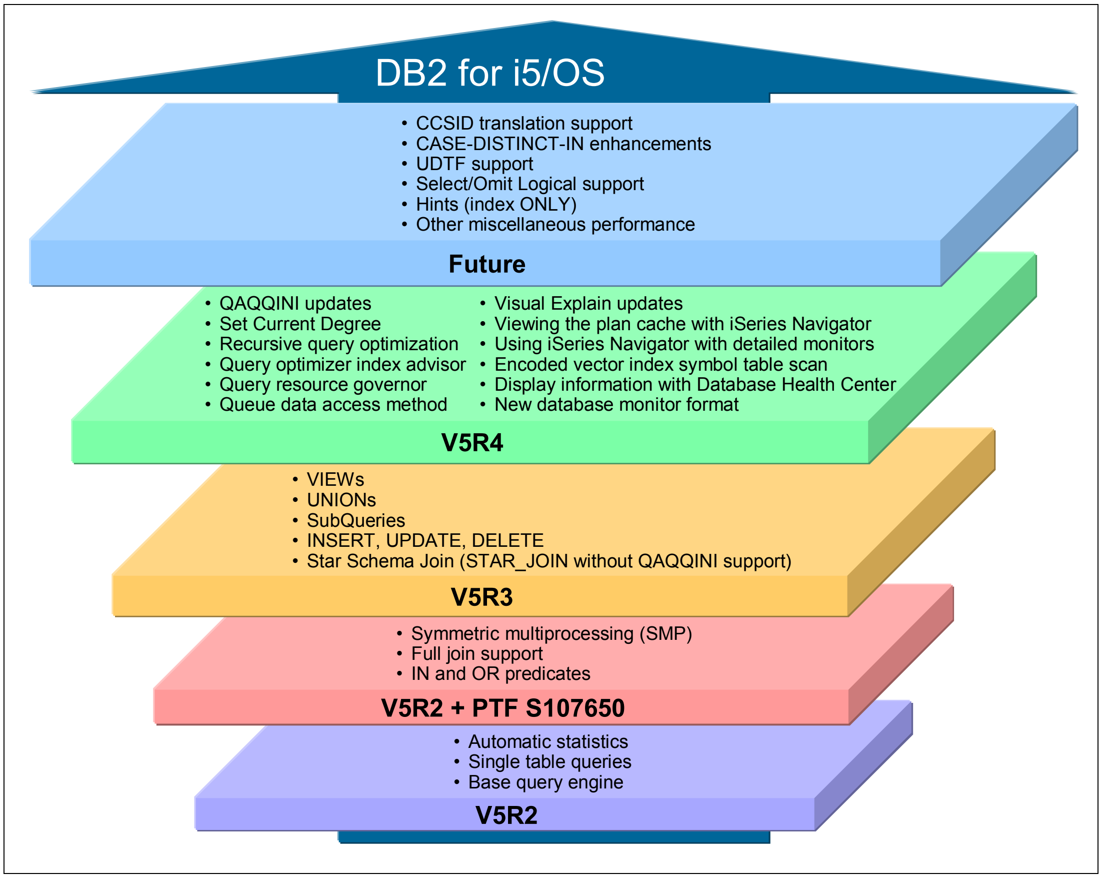

DB2 for i5/OS
• CCSID translation support
• CASE-DISTINCT-IN enhancements
• UDTF support
• Select/Omit Logical support
• Hints (index ONLY)
• Other miscellaneous performance
Future
• QAQQINI updates
• Set Current Degree
• Recursive query optimization
• Query optimizer index advisor
• Query resource governor
• Queue data access method
• Visual Explain updates
• Viewing the plan cache with iSeries Navigator
• Using iSeries Navigator with detailed monitors
• Encoded vector index symbol table scan
• Display information with Database Health Center
• New database monitor format
V5R4
• VIEWs
• UNIONs
• SubQueries
• INSERT, UPDATE, DELETE
• Star Schema Join (STAR_JOIN without QAQQINI support)
V5R3
• Symmetric multiprocessing (SMP)
• Full join support
• IN and OR predicates
V5R2 + PTF S107650
• Automatic statistics
• Single table queries
• Base query engine
V5R2

Figure 1-3 Staged implementation of new functionality

Figure 1-4 shows the supported interfaces for both SQE and CQE starting in OS/400 V5R2 and how they are routed. This routing is performed by a component of the query optimizer known as the *Query Dispatcher* or *dispatcher*. Refer to 2.3.1, "The Query Dispatcher" on page 16, for a detailed explanation of the features and functions of the dispatcher.

Most of the processing in CQE occurs above the Machine Interface (MI). In the case of SQE, the majority of the processing occurs below the MI. This is one of the main features that adds to the efficient way in which SQE processes queries.

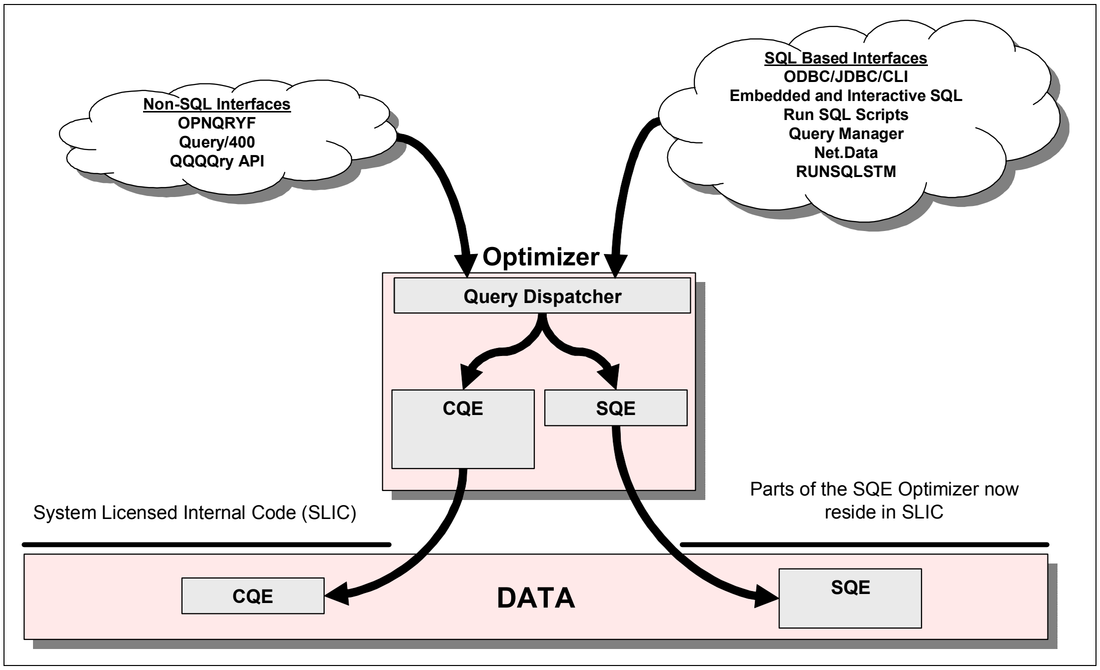


Figure 1-4 Query optimizer supported interfaces

1.5 Design criteria and features of SQE

As mentioned previously, the re-engineering process of DB2 Universal Database for iSeries focussed primarily on the areas that provide the greatest immediate benefits to our customers. At the same time, this process allows for continual enhancements to the product, to maintain the System i competitive edge.

In the following sections, we look at some of the design criteria considered during this re-engineering process. We also examine the features that were ultimately incorporated in the design of DB2 Universal Database for iSeries and the performance benefits from this design.

1.5.1 The objectives

In the design plan of the database engine for DB2 Universal Database for iSeries, the IBM design team was given the challenge to create a database engine that meets the following objectives:

- Make the DB2 SQL Query Engine easier to maintain
- Make the DB2 SQL Query Engine easier to enhance
- Give the DB2 SQL Query Engine far better performance

1.5.2 The SQE design

In line with the design objectives, as of the release of OS/400 V5R2 through i5/OS V5R4, DB2 for i5/OS incorporates the following features:

- A new query optimizer using object-oriented design and programming models, together with enhanced algorithms
- More functions below the MI, including the majority of the SQE Optimizer
- A new part of the query optimizer known as the *Query Dispatcher* or *dispatcher*

  The dispatcher routes the query request to either the SQE or the CQE portion of the optimizer, depending on the type of query interface from which the request originated.
- A new, separate component known as the *Statistics Manager*

  The Statistics Manager provides more sophisticated and more complex database statistics for use in the query access plan.
- New System Licensed Internal Code (SLIC) data access methods, known as the *SQE Primitives*, which are used in the actual implementation of a query
- An additional component of SQE known as the *plan cache*

  The plan cache is a repository that controls and physically stores query access plans as part of the query, without relying upon external objects or processes to provide it with information.

  > **Note:** V5R4 of iSeries Navigator support includes the capabilities to view and analyze the contents of the plan cache.

- Additional and enhanced query feedback and debug information messages through the Database Monitor (DBMON) and Visual Explain interfaces

These features translate into an SQE design that allows for greater flexibility in implementing efficient database functions with improved performance. This has resulted in numerous patent applications for the IBM development team responsible for the redesign of the query engine.

Refer to Chapter 2, “The architecture of DB2 for i5/OS” on page 11, for a more detailed explanation of the features and functionality in DB2 for i5/OS. The same chapter shows how you can benefit from this enhanced technology.

Performance improvements

IBM Rochester has performed extensive benchmarks to test the benefits of SQE in the execution of SQL statements in complex SQL environments. Some of the benchmarks are documented in Chapter 6, “Practical experience” on page 143. In complex SQL environments, we can conclude that, on average, the queries improve two times. Figure 1-5 shows response time improvements measured in an environment with 119 different longer running Business Intelligence type queries.

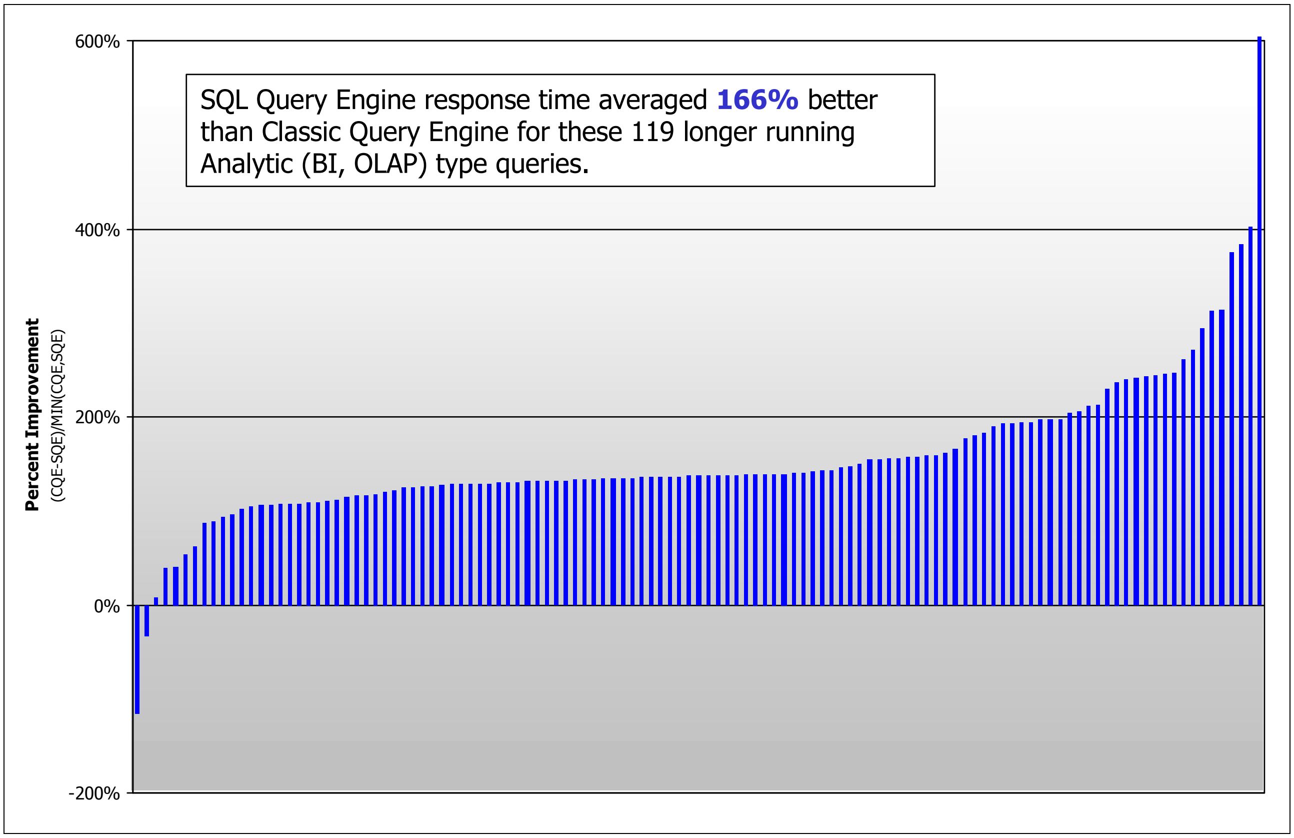


Figure 1-5 Response time improvements in a BI environment, tested in a V5R2 environment

2

The architecture of DB2 for i5/OS

In this chapter, we explain the architecture and features of the Structured Query Language (SQL) Query Engine (SQE) in DB2 for i5/OS starting with OS/400 V5R2.

As mentioned in Chapter 1, “Overview of DB2 for i5/OS” on page 1, two separate query engines coexist in the same system. These engines are the redesigned query engine (SQE) and the existing Classic Query Engine (CQE). Some queries continue to use CQE in OS/400 V5R2 and V5R3 and i5/OS V5R4, while other queries that originate from SQL interfaces can be processed by SQE. In line with the ease-of-use philosophy of the System i platform, the routing decision for processing the query by either CQE or SQE is transparent and under the control of the system. The requesting user or application program cannot control or influence this behavior.

In this chapter, we present a detailed description of the new components and functionality of SQE. We begin with a glance at the different methods in which SQL programs can be implemented on the System i platform.

2.1 SQL in a nutshell

SQL is a standardized programming language that is used to define and manipulate data in a relational database. The following sections provide a brief overview of the different methods in which SQL programs are implemented. They also highlight some of the characteristics that distinguish one type from the other.

2.1.1 Static SQL

Static SQL statements are embedded in the source code of a host application program. These host application programs are typically written in high-level languages, such as COBOL or RPG.

The host application source code must be processed by an SQL pre-compiler before compiling the host application program itself. The SQL pre-compiler checks the syntax of the embedded SQL statements and turns them into host language statements that interface with the database manager upon execution of the program. This process is often referred to as *binding*.

The SQL statements are therefore *prepared* before running the program and the associated access plan persists beyond the execution of the host application program.

2.1.2 Dynamic SQL

Dynamic SQL statements are prepared at the time an SQL application is executed. The SQL statements are passed to the database manager in the form of a character string. This string uses interfaces with the PREPARE and EXECUTE statements or an EXECUTE IMMEDIATE type of statement.

Access plans associated with dynamic SQL may not persist after a database connection or job is ended.

2.1.3 Extended dynamic SQL

In extended dynamic SQL, the SQL statements are dynamic. However, SQL packages are used to make the implementation of the dynamic SQL statements similar to that of static SQL. The QSQPRCED application program interface (API) is used to prepare the dynamic SQL statements and store the access plan in an SQL package. The SQL statements contained in the resulting SQL package persist until the SQL package is deleted or the SQL statement is explicitly dropped.

The iSeries Access Open Database Connectivity (ODBC) driver and Java™ Database Connectivity (JDBC™) driver both have extended dynamic SQL options available. They interface with the QSQPRCED API on behalf of the application program.

For more information about SQL in DB2 for i5/OS, refer to the *DB2 Universal Database for iSeries SQL Reference* in the iSeries Information Center at the following Web site:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/db2/rbafzmst.htm

2.2 Query processing before OS/400 V5R2

To fully understand the implementation of query management and processing in DB2 for i5/OS starting with OS/400 V5R2, it is important to see how the queries were implemented in previous releases of OS/400 for DB2 Universal Database for iSeries.

Figure 2-1 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. The interaction between the optimizer and the database engine occurs across the Machine Interface (MI). This results in a definite performance overhead. This is one of the areas that was targeted for improvement during the re-engineering process of DB2 Universal Database for iSeries for OS/400 V5R2.

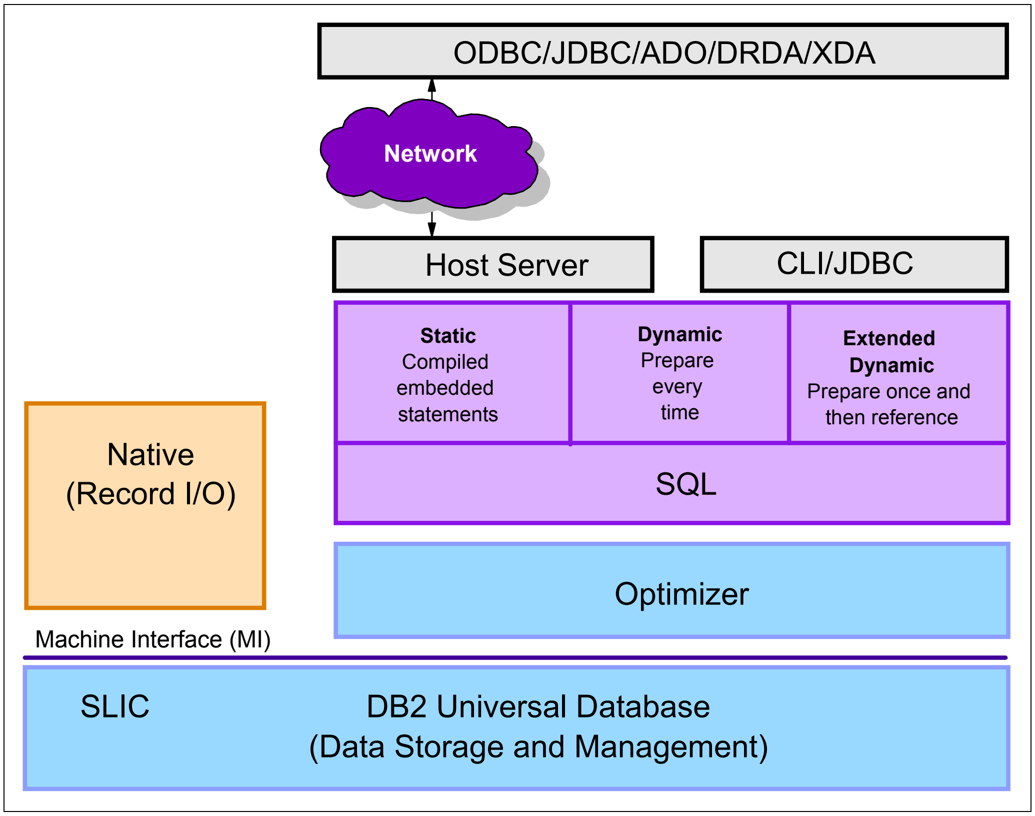


Figure 2-1 Database architecture before OS/400 V5R2

2.3 Query processing starting with OS/400 V5R2

There are several new and updated components of SQE starting from OS/400 V5R2. The SQE components include:

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

In the following section, we look briefly at the underlying architecture that supports SQE in OS/400 starting with V5R2. We follow this with a detailed discussion of the SQE components.

The architecture

Figure 2-2 shows an overview of the DB2 for i5/OS architecture starting with OS/400 V5R2 and where each SQE component fits. The functional separation of each SQE component is clearly evident. In line with our design objectives, this division of responsibility enables IBM to more easily deliver functional enhancements to the individual components of SQE, as and when required.

Notice that most of the SQE Optimizer components are implemented below the MI. This translates into enhanced performance efficiency.

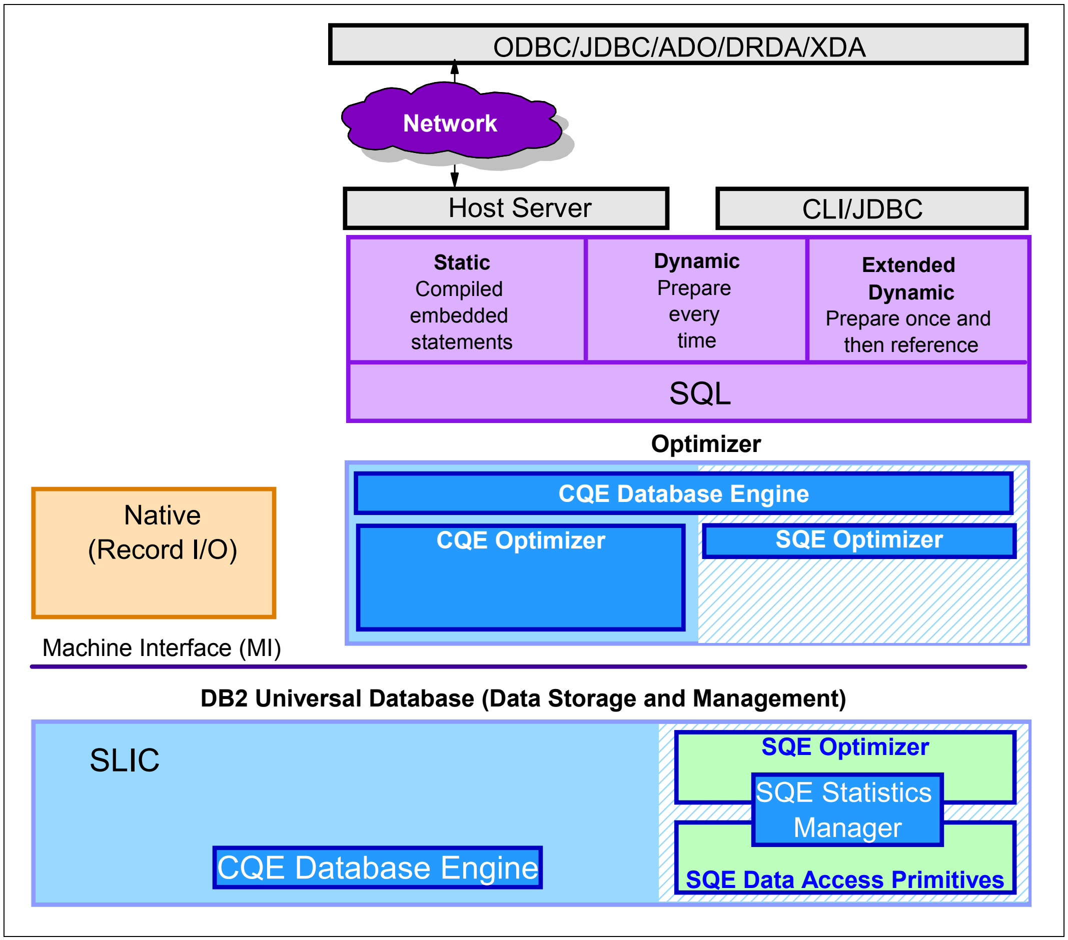


Figure 2-2 Database architecture starting with OS/400 V5R2

Object-oriented technology implementation

As mentioned previously, SQE is implemented using an object-oriented design and implementation approach. It uses a tree-based model, where each node is an independent and reusable query part. These components can interact and interface with each other in any given order or combination. In line with the overall design methodology, this allows for greater flexibility when creating new methods for implementing a query.

This design also allows SQE to optimize and execute these nodes independently from one another, thereby increasing performance and efficiency. In contrast, CQE uses a procedural model to implement a query. This model is less flexible, making it more difficult to implement new optimization methods and enhancements.

Figure 2-3 shows an example of the node-based implementation used in SQE. In this example, NODE1 represents a typical *index probe* access method. NODE2 and NODE3, each designed to perform one specific function, are independent from NODE1. They are not interdependent on each other either. Therefore, each node can be optimized and run independently from each other. In this example, NODE 2 only has one function, to retrieve the index and place it into main memory. Therefore, it is possible for any node in this tree to call on NODE2 to perform this function if it requires.

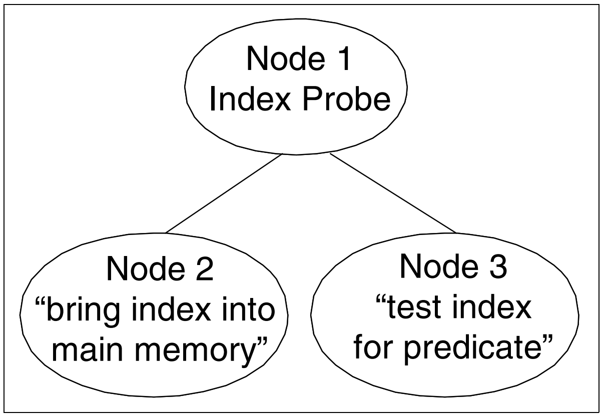


Figure 2-3 Index probe node implementation in SQE

The ability to traverse an index in reverse order is another example of how a new data access method can be added to SQE with relative ease. Again, the procedural nature of CQE prevents it from being easily enhanced to read an index in reverse order. A tree node was invented and implemented to accomplish this more easily in SQE. For details about the benefits of traversing an index in reverse order, see 6.6, "Reading through an index backwards" on page 151. The fact that most of the optimizer functions are implemented below the MI assisted greatly in implementing this functionality.

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

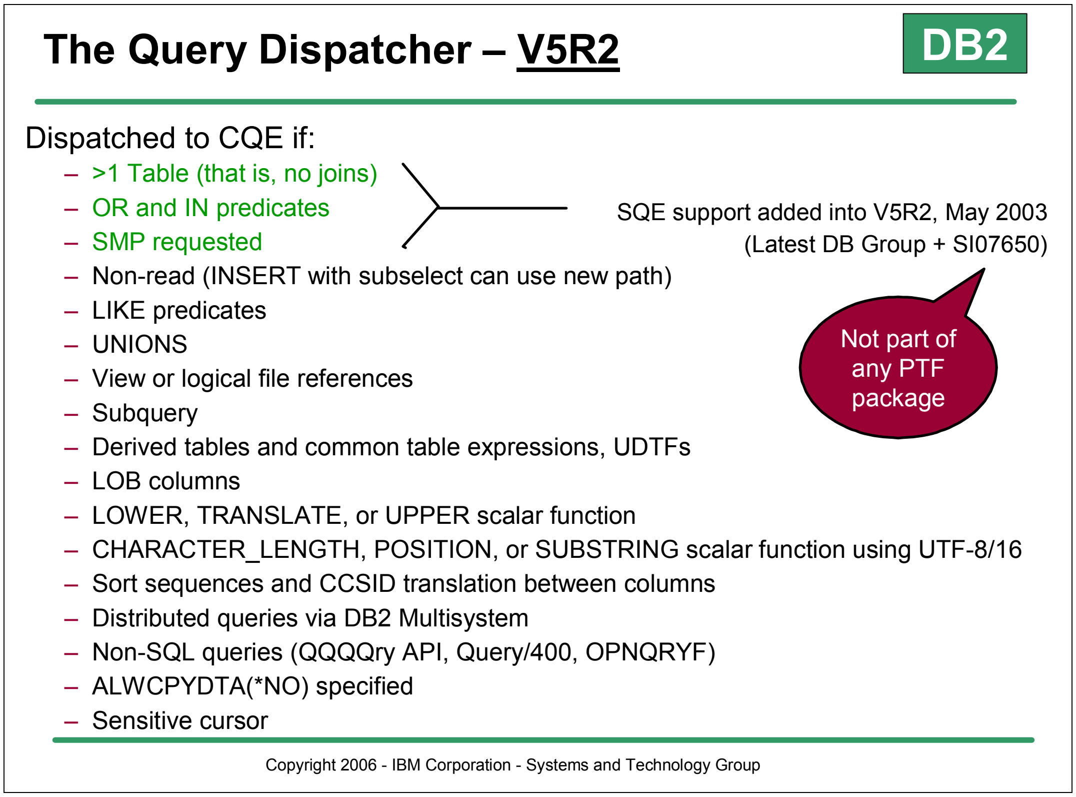
```
Select * from testtable where 1 = 0
```

Surprisingly, many applications use this type of query to force no rows to be returned. Due to 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. The scan checks each row of the table, in hopes of finding a row where 1 equals 0. In contrast, the tree node model of SQE easily allows a node to be added to check for non-sensical predicates before reading any rows in the specified table.

2.3.1 The 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 or non-SQL based. 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.

Depending on the version of OS/400 or i5/OS, the dispatcher routes to SQE certain types of SQL requests. Figure 2-4 shows that all the SQL requests are routed to SQE (the first three items highlighted in green), and the other ones are routed to CQE for OS/400 V5R2. As you can see, only a small percentage of the SQL requests are routed initially to SQE.



The Query Dispatcher – V5R2

DB2

Dispatched to CQE if:

- >1 Table (that is, no joins)
- OR and IN predicates
- SMP requested
- Non-read (INSERT with subselect can use new path)
- LIKE predicates
- UNIONS
- View or logical file references
- Subquery
- Derived tables and common table expressions, UDTFs
- LOB columns
- LOWER, TRANSLATE, or UPPER scalar function
- CHARACTER_LENGTH, POSITION, or SUBSTRING scalar function using UTF-8/16
- Sort sequences and CCSID translation between columns
- Distributed queries via DB2 Multisystem
- Non-SQL queries (QQQQry API, Query/400, OPNQRYF)
- ALWCPYDTA(*NO) specified
- Sensitive cursor

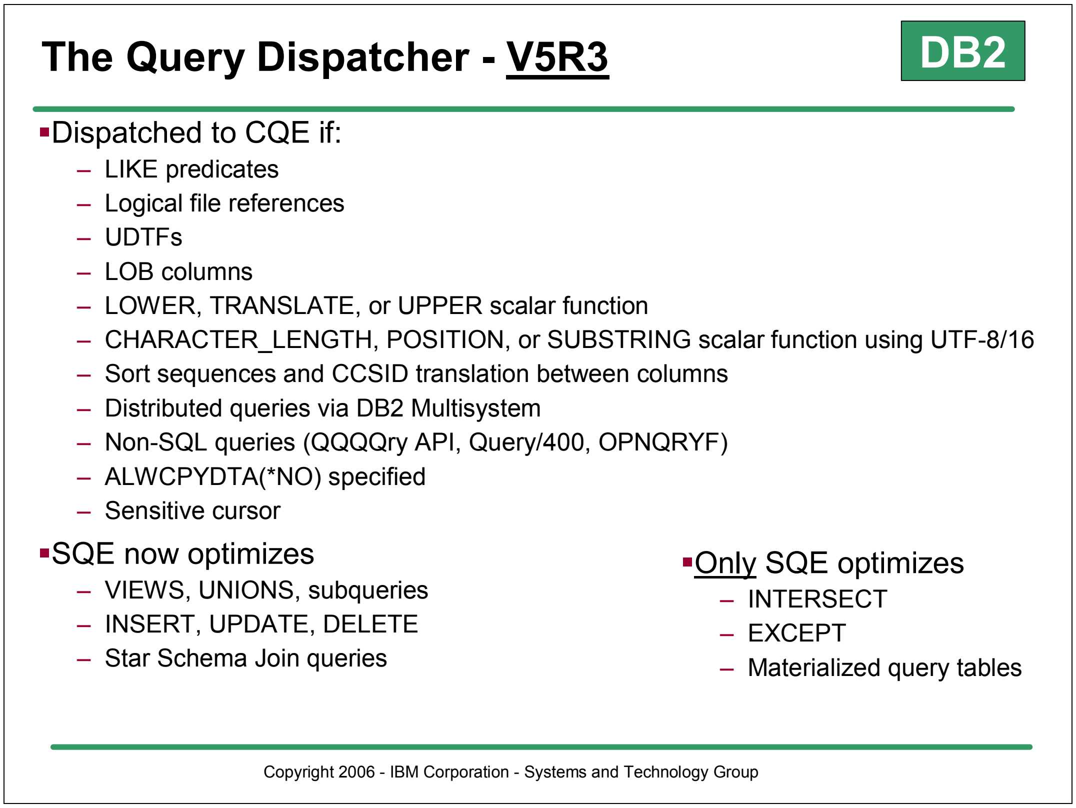
SQE support added into V5R2, May 2003
(Latest DB Group + SI07650)

Not part of any PTF package

Copyright 2006 - IBM Corporation - Systems and Technology Group

Figure 2-4 Query Dispatcher routing to SQE in V5R2

In OS/400 V5R3, the SQL requests that are routed to SQE are the first three (shown in green) from Figure 2-4, as well as the ones stated in Figure 2-5.



The Query Dispatcher - V5R3

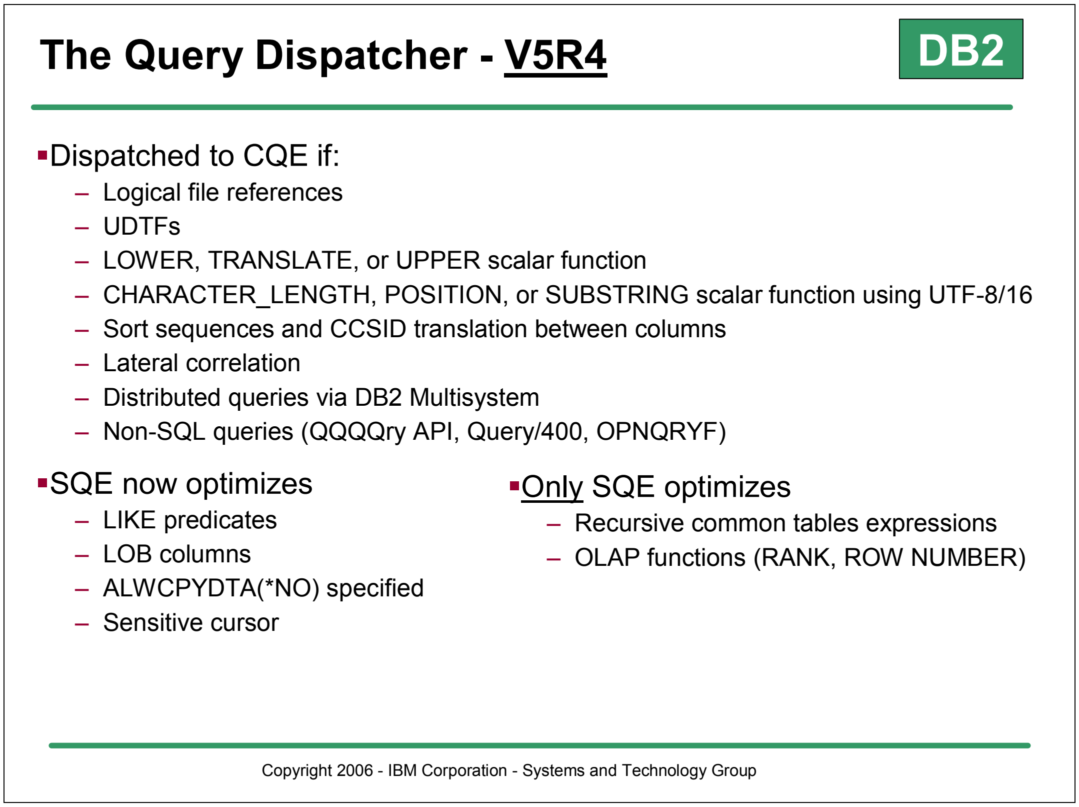
DB2

- Dispatched to CQE if:
  - LIKE predicates
  - Logical file references
  - UDTFs
  - LOB columns
  - LOWER, TRANSLATE, or UPPER scalar function
  - CHARACTER_LENGTH, POSITION, or SUBSTRING scalar function using UTF-8/16
  - Sort sequences and CCSID translation between columns
  - Distributed queries via DB2 Multisystem
  - Non-SQL queries (QQQQry API, Query/400, OPNQRYF)
  - ALWCPYDTA(*NO) specified
  - Sensitive cursor
- SQE now optimizes
  - VIEWS, UNIONS, subqueries
  - INSERT, UPDATE, DELETE
  - Star Schema Join queries
- Only SQE optimizes
  - INTERSECT
  - EXCEPT
  - Materialized query tables

Copyright 2006 - IBM Corporation - Systems and Technology Group

Figure 2-5 Query Dispatcher routing to SQE in V5R3

In i5/OS V5R4, additional types of SQL requests are routed to SQE as shown in Figure 2-6. As you can see with each release, more and more SQL types of requests are routed to SQE.



The Query Dispatcher - V5R4 DB2

- Dispatched to CQE if:
  - Logical file references
  - UDTFs
  - LOWER, TRANSLATE, or UPPER scalar function
  - CHARACTER_LENGTH, POSITION, or SUBSTRING scalar function using UTF-8/16
  - Sort sequences and CCSID translation between columns
  - Lateral correlation
  - Distributed queries via DB2 Multisystem
  - Non-SQL queries (QQQQry API, Query/400, OPNQRYF)
- SQE now optimizes
  - LIKE predicates
  - LOB columns
  - ALWCPYDTA(*NO) specified
  - Sensitive cursor
- Only SQE optimizes
  - Recursive common tables expressions
  - OLAP functions (RANK, ROW NUMBER)

Copyright 2006 - IBM Corporation - Systems and Technology Group

Figure 2-6 Query Dispatcher routing to SQE on V5R4

Table 2-1 summarizes which statements are being added to the SQE based on release.

Table 2-1 SQE integration timeline

	V5R2	V5R2 + PTF SI07650	V5R3	V5R4
Single Table	x	x	x	x
Grouping	x	x	x	x
Distinct	x	x	x	x
Ordering	x	x	x	x
Joins		x	x	x
OR & IN Predicates		x	x	x
SMP			x	x
START_JOIN INI Option			x	x
Views/NTEs/CTEs			x	x
Update/Delete			x	x
Subqueries			x	x
Unions			x	x

	V5R2	V5R2 + PTF SI07650	V5R3	V5R4
Like Predicates				x
LOBs (Referenced in queries)				x
Sensitive Cursors (ALWCPYDTA(*NO))				x
Lateral Correlations NTEs				
CCSID/Sort Sequence/Translation				
Select/Omit and Derived Key Index Support (ignore option via QAQQINI file)				
User Defined Table Functions				
Read Triggers				
Distributed Queries via DB2 Multisystem				
Native Database Logical File references				
Non-SQL Queries (QQQQry API, Query/400, OPNQRYF)				

Not all SQL statements are routed to SQE. For this reason, as of V5R4, the dispatcher routes an SQL statement to CQE if it finds that the statement references or contains the following items:

- Table function or lateral correlation
- LOWER, TRANSLATE or UPPER scalar function
- CHARACTER_LENGTH, POSITION, or SUBSTRING scalar function with a UTF-8 or UTF-16 argument
- CCSID character convention
- Sort sequence other than *HEX
- Distributed table (that is DB2 Multisystem)
- Reference to a data description specifications (DDS)-created logical file
- Reference to a table that has a logical file containing Select/Omit or Key derivations

> **Note:** The IGNORE_DERIVED_INDEX QAQQINI option can be used to have the dispatcher ignore the Select/Omit and derived logical files and route the query to SQE.

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 back 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
- Non-standard indexes or derived keys, for example 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 may 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 knows to route 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.

The following SQL statements are supported only by the SQE Engine:

- Recursive common table expressions
- INTERSECT & EXCEPT operations
- OLAP Expressions - RowNumber, Rank, and DenseRank
- ORDER OF
- Scalar full-selects

Therefore, if one of these SQL features is run in an environment that is not supported by the SQE, an error is signaled (SQL0255). For instance, imagine an application that is successfully executing an SQL statement that contains a recursive table expression. Later the application is changed to use a non-default sort sequence (that is, not *HEX) that an SQL statement containing the recursive table expression will return an error since the non-default sort sequence forces the dispatcher to use the CQE.

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

2.3.2 The 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.

The SQE Optimizer is a cost-based optimizer. This means that it attempts to implement a query using an access method that has the least amount of central processing unit (CPU) utilization and input/output (I/O) overhead costs, while still satisfying the required result set.

A fundamental characteristic distinguishes the SQE Optimizer from the CQE Optimizer. This is the fact that the SQE Optimizer has a separate component that is in charge of the knowledge of the metadata or the system's processing and performance capabilities. It simply asks questions that relate to the system and the tables used in the query. It uses the answers in its algorithms to calculate the cost. For example, to perform its cost calculations, the SQE Optimizer typically asks such questions as “How many rows are in the table?” or “How long will it take to scan the entire table?” In line with the policy of separation of responsibilities built into the new database engine, the Statistics Manager and SQE Data Access Primitives supply the answers to most of the Optimizer's questions. You can learn more about the Statistics Manager in 2.3.3, “Statistics Manager” on page 22. Also refer to 2.3.4, “Data access primitives” on page 24, which provides more information about SQE Data Access Primitives.

As shown in Figure 2-2 on page 14, most of the optimizer functionality is implemented beneath the MI. This close proximity to the data and database management system allows for greater flexibility and increased performance.

The SQE Optimizer uses both traditional optimization strategies and new techniques to ensure the best query implementation. From a user perspective, some of the more noticeable differences with the SQE Optimizer are:

- Temporary indexes are no longer considered.
- Table scans are more frequently considered, due to the implementation of the new, more efficient SQE Data Access Primitives.

The access plans remain true to the object-oriented design and implementation methodologies. They are now organized into tree-based structures to allow for maximum flexibility. The optimizer remains responsible for collecting and processing query feedback information messages. To learn more about query feedback messages, see 5.1, “Query optimization feedback” on page 90.

The design of DB2 for i5/OS includes a way in which SQE manages the time it spends optimizing dynamic SQL queries. The objective in both SQE and CQE is not to spend a small fraction of the total time needed to run the query, to determine the most efficient way to run it.

Prior to OS/400 V5R2, the CQE Optimizer used a clock-based time-out algorithm. This limited the time that was spent on optimizing the access plans of dynamic SQL queries where the access plan would not be saved. Using an all-inclusive list of available indexes, it resequenced the indexes, based on the number of matching index columns and the operators used in the WHERE clause of the SQL statement. This approach ensured that the most efficient indexes were optimized first, before the set time limit expired.

Starting with OS/400 V5R2, the amount of time that the SQE Optimizer spends optimizing an access plan is *not* limited based on a set clock-based time limit. 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. Next, these indexes are resequenced so that the most appropriate indexes are processed first and then reorganized further based on index-only access, index probe selectivity, total index selectivity, and the size of the index keys.

In processing the list of indexes, the SQE Optimizer always considers every index that is index-only access capable. However, while processing the remaining indexes, the SQE Optimizer stops the optimization process as soon as it encounters an index that is more expensive to use than the best index so far.

With the introduction of the plan cache component, the SQE Optimizer more consistently ensures that it implements the best possible plan. It is justifiable to spend more time optimizing a query to ensure that the best possible plan is saved in the plan cache from where it will be retrieved for future reuse. You can learn more about the plan cache component in 2.3.5, “The plan cache” on page 24.

Constraint and Referential Integrity Awareness

As of V5R3, the SQL Query Engine is constraint awareness. This means it is capable of using the constraint definitions from the database to rewrite queries in a more efficient way. We illustrate this with an example. Let us add a CHECK constraint to table TB1 as shown in Example 2-1.

Example 2-1 CHECK constraint on table TB1

```
ALTER TABLE TB1
CHECK(COL1 BETWEEN 1 AND 100)
```

A particular user submits the query illustrated in Example 2-2. Assume that the user typed 150 for the host variable value.

Example 2-2 Original user query

```
SELECT * FROM TB1 WHERE COL1 = :HOSTVAR
```

The SQL Query Engine is cabable of rewriting the query as shown in Example 2-3. After the query is rewritten, SQE does not have to read data since the result of the query is an empty set.

Example 2-3 Rewritten query

```
SELECT * FROM TB1 WHERE (:HOSTVAR BETWEEN 1 AND 100) AND COL1= :HOSTVAR
```

As of V5R3, the SQL Query Engine also has Referential Integrity (RI) and Constraint Awareness. This means that the optimizer is capable of using the RI conditions to rewrite the queries to eliminate unneccesary join combinations.

2.3.3 Statistics Manager

The CQE and SQE Optimizer use statistical information to determine the best access plan for implementing a query. Therefore, statistical information must be accurate, complete, and current to be of good use to the optimizer. Better and more accurate statistics enable the query optimizer to build a more accurate and realistic query implementation plan, which results in a more efficient query.

One of the stated objectives in the design of DB2 for i5/OS was to enhance a component, separate from the optimizer, that:

- Provides more sophisticated database statistics
- Provides more complete database statistics
- Interfaces with the optimizer in a flexible manner
- Exploits modern and statistics gathering techniques
- Automates collection and refresh of statistics

Prior to OS/400 V5R2, the optimizer relied solely on indexes as a source of statistical information for any given table. In the design of DB2 for i5/OS, the function of collecting and managing statistical data is performed by a new, separate component known as the *Statistics Manager*. Unlike many other database products that rely on manual intervention and processes to collect database statistics, this is a fully automated process in DB2 for i5/OS. However, you have the facility to manually manage database statistics if you require to do so for specific reasons.

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

The Statistics Manager does not actually run or optimize the query. Instead, it 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. Using many different sources of statistical information, the Statistics Manager always provides an answer to the optimizer. In cases where it cannot provide an answer based on actual existing statistics information, it is designed to provide a predefined default answer.

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

- **Cardinality of values**: This is the number of unique or distinct occurrences of a specific value in a single column or multiple columns of a table.
- **Selectivity**: Also known as a *histogram*, this information is an indication of the number of rows that 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.

The SQE Statistics Manager uses a hybrid approach to manage database statistics. The majority of this information can be obtained from existing binary-radix indexes or encoded-vector indexes (EVI). An advantage of using indexes is that the information is available to the Statistics Manager as soon as the index is created or maintained.

In cases where the required statistics cannot be gathered from existing indexes, statistical information, such as cardinality, histograms, and frequent values, is constructed over single columns of a table and stored internally as part of the table. 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. Refer to Chapter 4, “Statistics Manager” on page 63, which discusses the Statistics Manger in more detail.

Note: The total size of the database object increases by approximately 8 KB to 12 KB per column for which statistics are being maintained.

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 only indexes with statistics 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 for i5/OS continues to rely on its indexing technology to provide statistics information and a data access method.

For more information about indexing and indexing strategies, refer to the IBM white paper *Indexing and Statistics Strategies for DB2 UDB for iSeries* on the Web at:

http://www-03.ibm.com/servers/enable/site/bi/strategy/strategy.pdf

2.3.4 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, it provides the actual implementation plan of the query.

You may notice that the SQE Data Access Primitives use main memory and subsystem resources more aggressively than with CQE. This is due to the fact that the SQE data access algorithms are more in tune with OS/400 and i5/OS single-level storage. This more efficient algorithm is the primary reason why the table scan access method is used more often on SQE than with CQE.

For example, the mechanisms for storing temporary results were remodelled. Complex SQL queries often combine multiple tables. This required DB2 Universal Database to combine all of the tables into a temporary result table before the specified grouping or ordering could be performed. By exploiting the efficient structures of the data access primitives and its close proximity to the actual data, the design minimizes data movement and eliminates the need to create real database objects to store temporary results. Instead of creating real database objects, SQE employs expert cache to simultaneously retrieve data into memory pages and process the data directly from these memory pages. This significantly reduces processing overhead.

Although the SQE Data Access Primitives employ many of the same data access methods as used in CQE, it underwent a total redesign. It features several data access methods and algorithms for distinct processing and implements queries by fully using the symmetric multiprocessing (SMP) feature. On average, CPU consumption is much less than what it was previously. For a more detailed discussion about the data access primitives, refer to Chapter 3, “Data access methods” on page 27.

2.3.5 The plan cache

The *plan cache* is a repository that contains query implementation plans for queries that are optimized by the SQE Optimizer. Query access plans generated by CQE are not stored in the plan cache. The architecture of DB2 for i5/OS allows for only one plan cache per System i model 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. Furthermore, when an access plan is tuned, for example when creating an index, all queries can benefit from this updated access plan. This eliminates the need to re-optimize the query and results in greater efficiency faster processing time. In the case of CQE, each query had to update its own access plan to benefit from the newly created index.

Figure 2-7 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 run to determine if a valid access plan exists that satisfies the requirements of the query. If a valid access plan is found, it is used to implement the query. Otherwise, a new access plan is created and stored in the plan cache for future use. The plan cache is automatically updated when new query access plans are created or when new statistics or indexes become available. Access plans generated by CQE are not stored in the SQE Plan Cache.

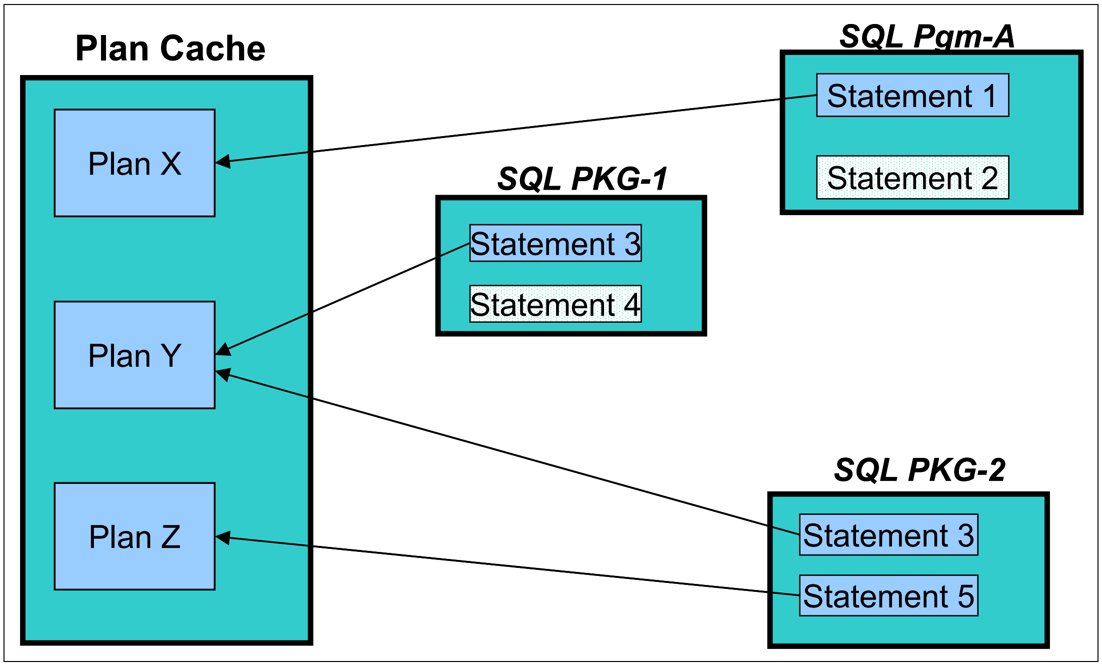


Figure 2-7 Plan cache functionality

To demonstrate this concept, assume that Statement 2 in Figure 2-7 is executed for the first time by the SQE. Statement 4 can only be implemented by CQE, which is why Statement 4 has no relationship to the plan cache. In this example, a new access plan is inserted into the plan cache to satisfy Statement 2. The access plan for Statement 4 is not be stored in the plan cache.

The SQE Plan Cache works in close conjunction with the system wide statement cache, as well as SQL programs, SQL packages, and service programs. Initially created with an overall size of 256 million bytes, occupying approximately 250 MB, a plan cache entry contains the SQL statement text, the optimized access plan, and some additional information of the runtime attributes. All systems are currently configured with the same size plan cache, regardless of the server size or hardware configuration.

While each individual SQE access plan may vary in size, in its compressed state, it remains significantly smaller than those generated and stored by CQE. When the plan cache exceeds the designated size, a background task is automatically scheduled to remove old access plans from the plan cache. Access plans are automatically deleted from the plan cache based upon the age of the access plan and how frequently it is being used. The total number of access plans stored in the plan cache depends largely upon the complexity of the SQL statements that are executed. In our testing environments, we typically observe around 6,000 unique access plans stored in the plan cache.

Multiple access plans can be maintained for a single SQL statement. Although the SQL statement is the primary hash key to the plan cache, a different environmental setting can cause different access plans to be stored in the plan cache, each one matching the specific environment. Examples of these environmental settings include:

- Different SMP degree settings for the same query
- Different library lists specified for the same SQL request
- Different settings for the job’s share of available memory in the current pool
- Different ALWCPYDTA settings

Currently, the plan cache can maintain a maximum of three different access plans for the same SQL statement. As new access plans are created for the same SQL statement, older and less frequently used access plans are discarded to make room for the new access plans. The number of access plans that can be stored for each SQL statement is adjustable and may change in the future.

Note: The SQE Plan Cache is cleared when a system initial program load (IPL) is performed.

Based on this information, you may notice the real potential for the SQE Plan Cache to serve as the foundation for a self-learning, self-adjusting query optimizer in future i5/OS releases. Such a query optimizer can compare the access plan stored in the plan cache against the actual execution time of the associated SQL request. Then it can automatically adjust its implementation strategies in an effort to improve performance and efficiency.

Note: V5R4 support of iSeries Navigator includes the capabilities to view and analyze the contents of the plan cache. It also has filtering capabilities to narrow down the analysis of the data from the plan cache.

2.4 Summary

You should now understand the architecture of DB2 for i5/OS, starting with OS/400 V5R2 through i5/OS V5R4, as well as the new components of SQE. With this knowledge, you should understand clearly how this newly enhanced technology can benefit you now and in future releases of i5/OS.

No two business environments are the same. Therefore, selecting and implementing the correct strategies to suit your particular environment is equally important if you want to achieve the benefits of the new database design and technology. In the next few chapters, we introduce and explain some of the new SQE components in greater detail to assist you in making better informed decisions that are appropriate for your particular environment and requirements.

3

Data access methods

As explained in Chapter 2, “The architecture of DB2 for i5/OS” on page 11, the process of optimizing a query (creating an access plan) is separate and different from the work of actually performing the query (executing the access plan).

In this chapter, we discuss the data access methods or “tools” that are available to the Structured Query Language (SQL) Query Engine (SQE) to perform the work of retrieving physical rows from the DB2 for i5/OS database. We explain how the SQE uses its arsenal of components (Primitives) to satisfy the variety of query requests that it receives. We also discuss how the attributes of a particular query’s requirements affect the access methods or algorithms used.

3.1 Available data access methods

The query engine has two kinds of raw material with which to satisfy a query request:

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

Only two types of *permanent* database objects can be used as source material for a query: tables (physical files) and indexes (binary-radix or encoded-vector indexes (EVIs). During the execution of an access plan, the query engine may need to use *temporary* objects or work spaces to hold interim results or references. The two major types of temporary objects are *hash tables* and *lists*. Lists are further subdivided into sorted, unsorted, and address lists. Address lists may take the form of either bitmaps or relative record number (RRN) lists.

The query engine's toolkit consists of three types of operations to manipulate this set of permanent and temporary objects: create, scan, and probe. Figure 3-1 and Figure 3-2 show the operations that can be performed on each object type. The symbols shown in the tables are similar to the icons used by Visual Explain (see 5.1.4, "Visual Explain" on page 118). Figure 3-1 shows the access methods for permanent objects.

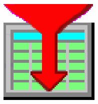
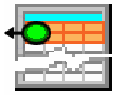
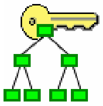
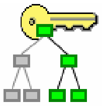
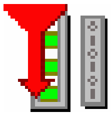
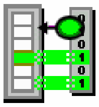
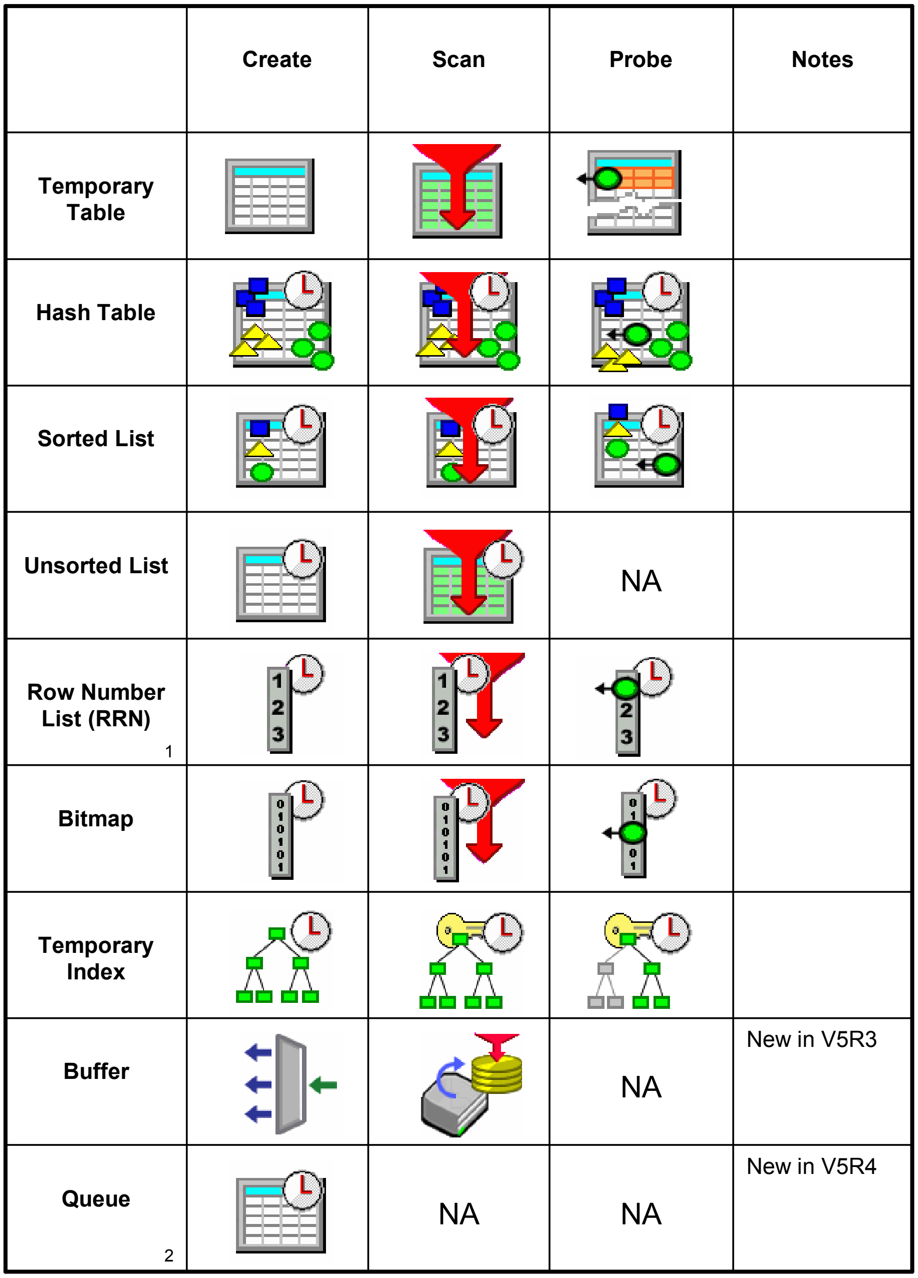
	Scan	Probe	Notes
Table			
Radix Index			
Encoded Vector Index (EVI)		 *	New in V5R3 * New in V5R4

Figure 3-1 Access methods for permanent objects

Figure 3-2 shows the access methods for temporary objects.

> **Important:** In Figure 3-2, the Create column is not an access method, but the query optimizer needs to create a temporary object in order to use the access methods.



	Create	Scan	Probe	Notes
Temporary Table				
Hash Table				
Sorted List				
Unsorted List			NA	
Row Number List (RRN) 1				
Bitmap				
Temporary Index				
Buffer			NA	New in V5R3
Queue 2		NA	NA	New in V5R4

Figure 3-2 Access methods for temporary objects

Notes for Figure 3-2:

- **RRN (1):** In the Classic Query Engine (CQE) environment, RRN is a temporary index created by the database above the Machine Interface (MI) level. In an SQE environment, it is an internal object that exists only in System Licensed Internal Code (SLIC).
- **Queue (2):** The Queue is a temporary object that allows the optimizer to feed the recursion of a recursive query by putting on the queue those data values that are needed for the recursion. This data typically includes those values used on the recursive join predicate and other recursive data that is being accumulated or manipulated during the recursive process.

3.2 SQE Primitives in action

SQE Primitives are responsible for executing the access plan as formulated by the optimizer phase of the query process. An access plan consists of one or more integrated steps (nodes) assembled to retrieve and massage data from DB2 tables to produce results desired by the information requestor. These steps may involve selecting, ordering, summarizing, and aggregating data elements from a single table or from related (joined) rows from multiple tables.

A helpful way to represent the relationships of the steps prescribed by a query access plan is to use a hierarchy diagram. In this diagram, the final results node is illustrated at the top of the chart, with the raw input source nodes illustrated toward the bottom of the chart. These diagrams tend to look like a tree, with the top nodes as the tree trunk and the lower nodes as the root system. A simple query that requires a scan of a single table (`SELECT * FROM table`) might appear like the example in Figure 3-3.

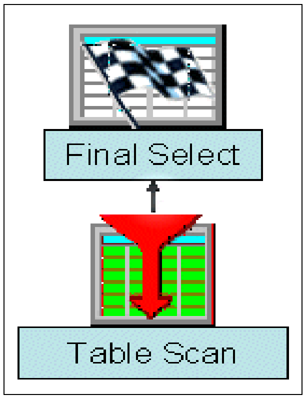


Figure 3-3 Graphical representation of a simple access plan

An example of a more complex query requiring a join and intermediate temporary objects (`SELECT * FROM table1, table2 WHERE table1.columna = table2.columnb`) might look like the example in Figure 3-4.

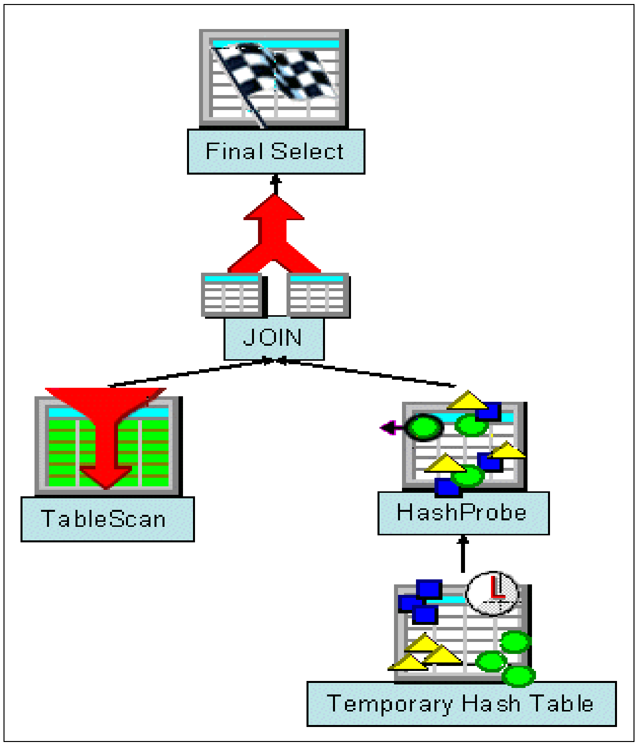


Figure 3-4 Graphical representation of a query access plan

Access plan flow and control

While the data provided by the permanent tables tends to flow upward through the nodes of the diagram, the initiation of the nodes is controlled from the top, down, and from left to right. For example, in Figure 3-4, the SQL requestor waits for the first row of results to be delivered from the *Final Select node*. This forces the Final Select node to request the first row from the result of the *Join node*. This, in turn, must produce the first row resulting from the join of the *table scan* with the corresponding row or rows from the *hash probe*. As soon as the Hash Probe is called upon to deliver rows, it triggers the creation and population of the hash table by the *Temporary Hash Table node*. Each node in the access plan is, in effect, both a supplier (source) of data to the node above it and a consumer (requestor) of data from the node below it. This simplified approach is a direct result of the object-oriented (OO) techniques employed in the enhanced query implementation referenced in Chapter 1, “Overview of DB2 for i5/OS” on page 1.

It is up to the query optimizer to construct a plan in which each node maximizes its use of disk inputs and outputs (I/Os), memory, and processor cycles. The job of the SQE Primitives is to ensure that each node performs its assigned work as quickly and efficiently as possible. This methodology also supports the ability to multithread each node or group of nodes within the access plan to use multiple processors where appropriate.

SQE takes advantage of multiple processors to the extent allowed in the PARALLEL_DEGREE environment variable in the QAQQINI table or via the Change Query Attribute (CHGQRYA) command. While the PARALLEL_DEGREE value specifies the maximum number of parallel threads for the overall query job, the actual number of threads employed by each node within the implementation plan may be limited by the access method and attributes of the temporary objects used. For example, if a temporary list is to be used for grouping, but only three values exist for the grouped-by column, then a node can use only three threads in parallel to create or scan the list.

3.3 Details about the data access methods

In the following section, we describe each data access method (sometimes referred to as *SQE Primitives*). We also provide examples of requirements that may cause the use of the method to satisfy a query. You can find symbols similar to the Visual Explain icons used to represent the methods. Where appropriate, sample debug messages and Database Monitor messages are included, which can indicate that a particular method is employed.

3.3.1 Permanent table operations

The tables and diagrams in the following sections explain the data access methods (scan and probe) for the permanent object types of tables and indexes.

Table scan

A table scan is the easiest and simplest operation that can be performed against a table. It sequentially processes all of the rows in the table to determine if they satisfy the selection criteria specified in the query. It does this in a way to maximize the I/O throughput for the table. Table 3-1 outlines the attributes of the table scan.

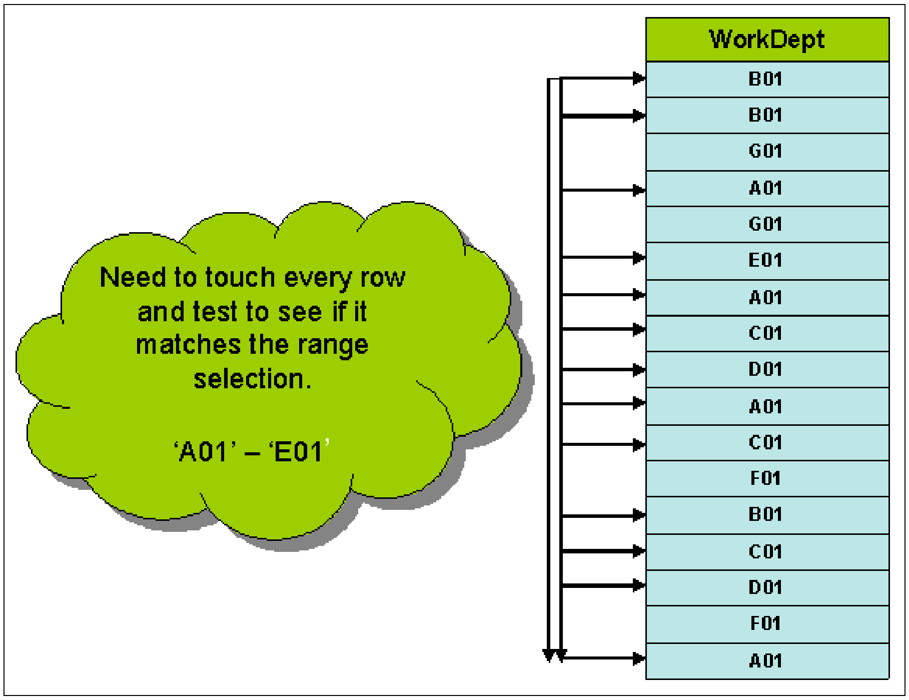
Table 3-1 Table scan attributes

Data access method	**Table scan**
Visual Explain icon	
Description	This scan reads all of the rows from the table and applies the selection criteria to each of the rows within the table. The rows in the table are processed in no guaranteed order, but typically they are processed sequentially.
Advantages	► It minimizes page I/O operations through asynchronous pre-fetching of the rows since the pages are scanned sequentially. ► It requests a larger I/O to fetch the data efficiently.
Considerations	All rows in the table are examined regardless of the selectivity of the query. Rows marked as deleted are still paged into memory even though none will be selected. You can reorganize the table to remove deleted rows.
Likely to be used	► When expecting a large number of rows returned from the table ► When the number of large I/Os needed to scan is fewer than the number of small I/Os that are required to probe the table

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

Messages indicating use	► Optimizer Debug:
	`CPI4329 – Arrival sequence was used for file EMPLOYEE`
	► PRTSQLINF:
	`SQL4010 – Table scan access for table 1.`

Figure 3-5 illustrates the concept of scanning an entire table to select only certain rows.


Need to touch every row and test to see if it matches the range selection.
'A01' – 'E01'

WorkDept
B01
B01
G01
A01
G01
E01
A01
C01
D01
A01
C01
F01
B01
C01
D01
F01
A01

Figure 3-5 Table scan concept

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. Table 3-2 outlines the attributes of the table probe.

Table 3-2 Table probe attributes

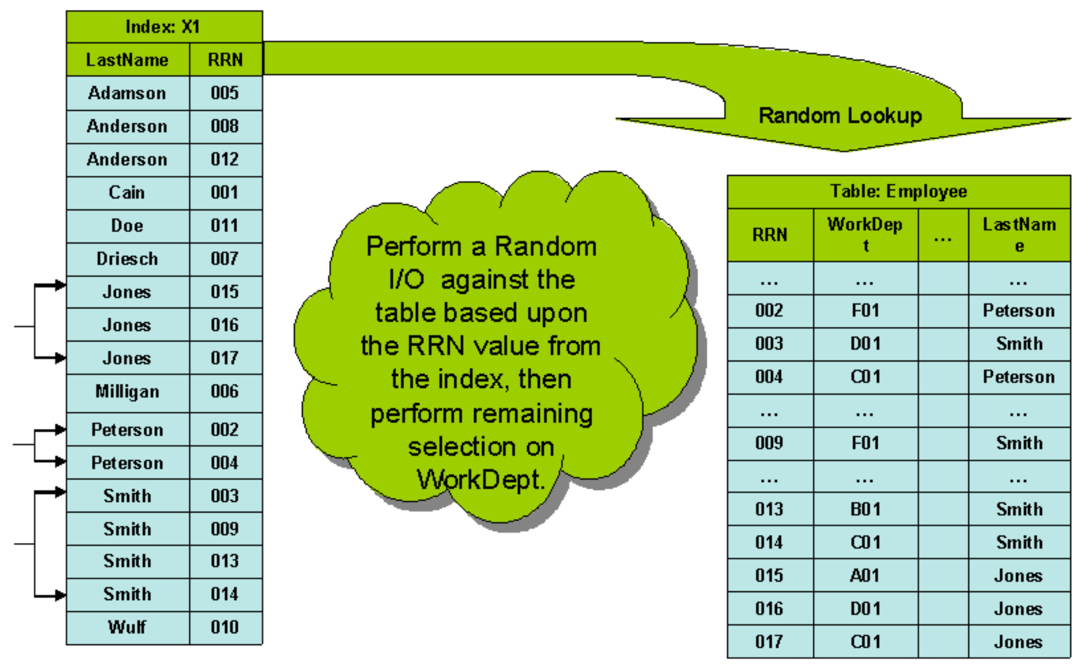
Data access method	Table probe
Visual Explain icon	
Description	This probe reads a single row from the table based upon a specific row number. A random I/O is performed against the table to extract the row.
Advantages	▶ It requests smaller I/Os to prevent paging rows into memory that are not needed. ▶ It can be used in conjunction with any access method that generates a row number for the table probe to process.
Considerations	Because of the synchronous random I/O, the probe can perform poorly when a large number of rows is selected.
Likely to be used	▶ When row numbers (either from indexes or temporary row number lists) are being used, but data from the underlying table rows are required for further processing of the query ▶ When processing any remaining selection or projection of the values

Data access method	**Table probe**

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

Messages indicating use	There is no specific message that indicates the use of a table probe. The messages in this example illustrate the use of a data access method that generates a row number that is used to perform the table probe operation.
	► Optimizer Debug:
	`CPI4328 – Access path of file X1 was used by query`
	► PRTSQLINF:
	`SQL4008 – Index X1 used for table 1.`
	`SQL4011 – Index scan-key row positioning (probe) used on table 1.`

Figure 3-6 shows the concept of probing individual rows within a table based on a record address.



Index: X1

LastName	RRN
Adamson	005
Anderson	008
Anderson	012
Cain	001
Doe	011
Driesch	007
Jones	015
Jones	016
Jones	017
Milligan	006
Peterson	002
Peterson	004
Smith	003
Smith	009
Smith	013
Smith	014
Wulf	010

Perform a Random I/O against the table based upon the RRN value from the index, then perform remaining selection on WorkDept.

Table: Employee

RRN	WorkDept	...	LastName
...	...		...
002	F01		Peterson
003	D01		Smith
004	C01		Peterson
...	...		...
009	F01		Smith
...	...		...
013	B01		Smith
014	C01		Smith
015	A01		Jones
016	D01		Jones
017	C01		Jones

Figure 3-6 Table probe concept

Radix index scan

A radix index scan operation is used to retrieve the rows from a table in a keyed sequence. Like a table scan, all of the rows in the index are sequentially processed, but the resulting row numbers are sequenced based upon the key columns. Table 3-3 outlines the attributes of the index scan.

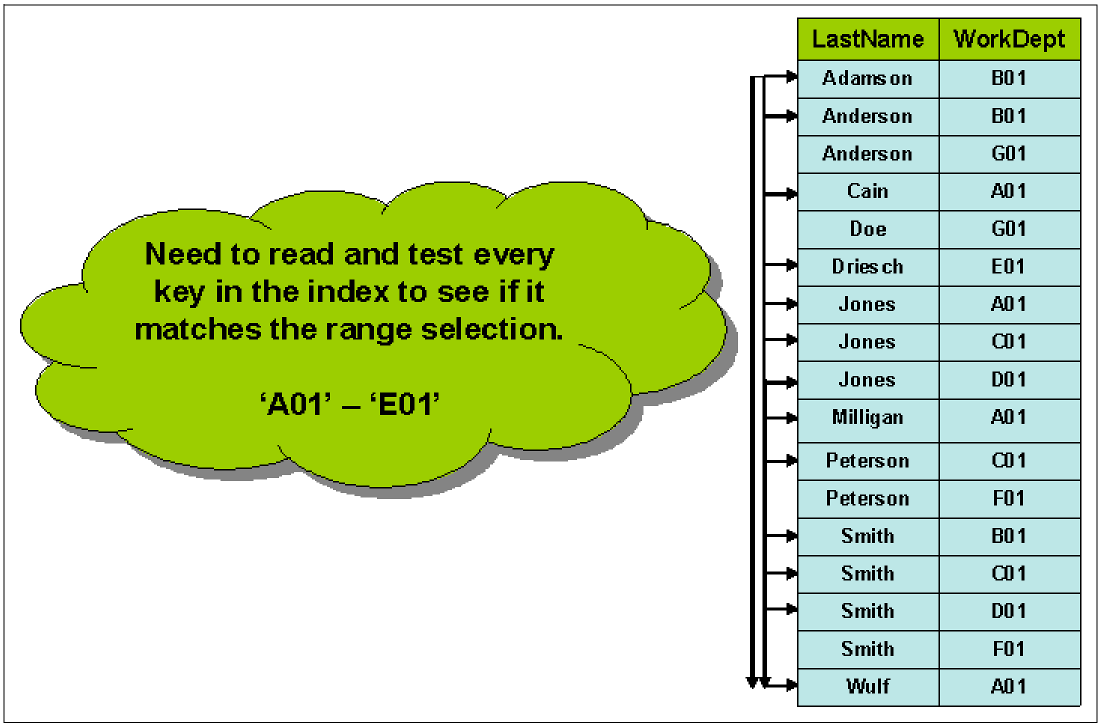
Table 3-3 Radix index scan attributes

Data access method	**Radix index scan**
Visual Explain icon	
Description	This method sequentially scans and processes all of the keys associated with the index. Any selection is applied to every key value of the index before a table row.
Advantages	► Only those index entries that match any selection continue to be processed. ► There is the potential to extract all of the data from the index keys' values, eliminating the need for a table probe. ► It returns the rows back in a sequence based upon the keys of the index.
Considerations	Generally requires a table probe to be performed to extract any remaining columns required to satisfy the query. Can perform poorly when a large number of rows are selected because of the random I/O associated with the table probe.
Likely to be used	► When asking for or expecting only a few rows to be returned from the index ► When sequencing the rows is required for the query (for example, ordering or grouping) ► When the selection columns cannot be matched against the leading key columns of the index

Example SQL statement	`CREATE INDEX X1 ON Employee (LastName, WorkDept)`
	
	`SELECT * FROM Employee`
	`WHERE WorkDept BETWEEN 'A01' AND 'E01'`
	`ORDER BY LastName`
	`OPTIMIZE FOR 30 ROWS`

Messages indicating use	► Optimizer Debug:
	`CPI4328 -- Access path of file X1 was used by query.`
	► PRTSQLINF:
	`SQL4008 -- Index X1 used for table 1.`

Figure 3-7 illustrates the concept of scanning an index to find selected key values.


Need to read and test every key in the index to see if it matches the range selection.
'A01' – 'E01'

LastName	WorkDept
Adamson	B01
Anderson	B01
Anderson	G01
Cain	A01
Doe	G01
Driesch	E01
Jones	A01
Jones	C01
Jones	D01
Milligan	A01
Peterson	C01
Peterson	F01
Smith	B01
Smith	C01
Smith	D01
Smith	F01
Wulf	A01

Figure 3-7 Index scan concept

Radix index probe

A radix index probe operation is used to retrieve the rows from a table in a keyed sequence. The main difference between the radix index probe and the radix index scan is that the rows being returned must first be identified by a probe operation to subset the rows being retrieved. Table 3-4 outlines the attributes of the index probe.

Table 3-4 Radix index probe attributes

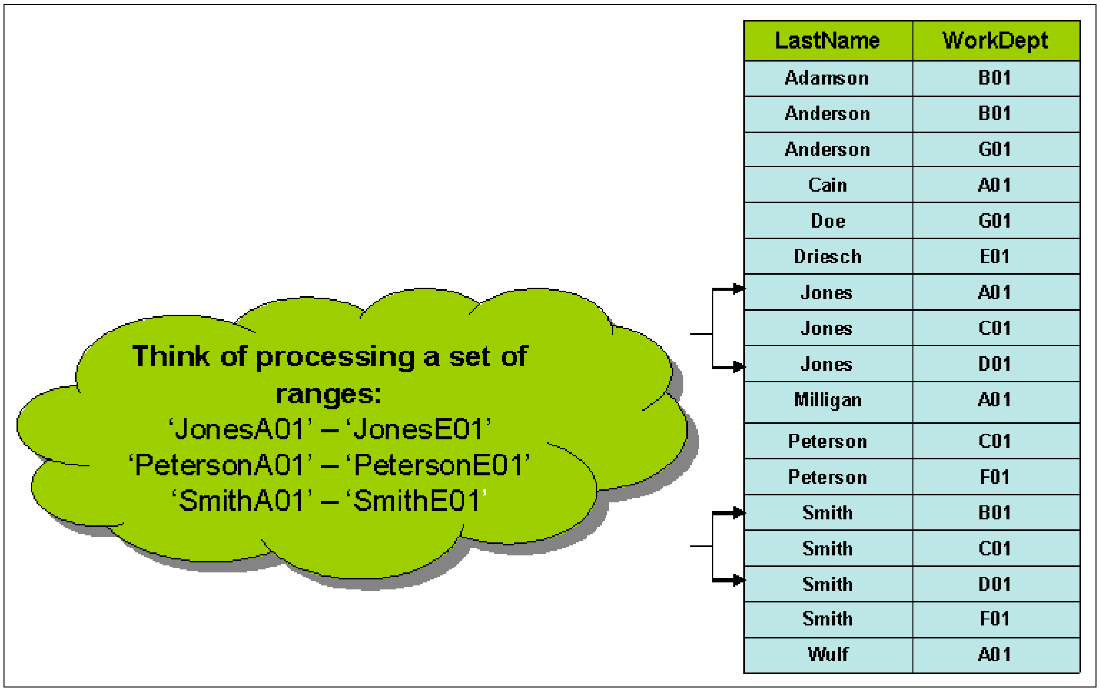
Data access method	**Radix index probe**
Visual Explain icon	
Description	The index is quickly probed based upon the selection criteria that was rewritten into a series of ranges. Only those keys that satisfy the selection are used to generate a table row number.
Advantages	► Only those index entries that match any selection continue to be processed. ► It provides quick access to the selected rows. ► There is the potential to extract all of the data from the index keys' values, eliminating the need for a table probe. ► It returns the rows back in a sequence based upon the keys of the index.
Considerations	It generally requires a table probe to be performed to extract any remaining columns required to satisfy the query. It can perform poorly when a large number of rows is selected because of the random I/O associated with the table probe.

Data access method	**Radix index probe**
Likely to be used	► When asking for or expecting only a few rows to be returned from the index ► When sequencing the rows is required by the query (for example, ordering or grouping) ► When the selection columns match the leading key columns of the index

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

Messages indicating use	► Optimizer Debug:
	`CPI4328 -- Access path of file X1 was used by query.`
	► PRTSQLINF:
	`SQL4008 -- Index X1 used for table 1.`
	`SQL4011 -- Index scan-key row positioning used on table 1.`

Figure 3-8 illustrates the concept of probing a permanent index to subset or sequence the rows that are eventually selected.



Think of processing a set of ranges:
'JonesA01' – 'JonesE01'
'PetersonA01' – 'PetersonE01'
'SmithA01' – 'SmithE01'

LastName	WorkDept
Adamson	B01
Anderson	B01
Anderson	G01
Cain	A01
Doe	G01
Driesch	E01
Jones	A01
Jones	C01
Jones	D01
Milligan	A01
Peterson	C01
Peterson	F01
Smith	B01
Smith	C01
Smith	D01
Smith	F01
Wulf	A01

Figure 3-8 Index probe concept

Encoded-vector index scan

An EVI symbol table scan operation is used to retrieve the entries from the symbol table portion of the index. Table 3-5 outlines the attributes of the EVI index scan.

Table 3-5 Encoded-vector index scan

Data access method	**Encoded-vector index symbol table scan**
Visual Explain icon	
Description	This method sequentially scans and processes all of the symbol table entries that are associated with the index. Any selection is applied to every entry in the symbol table. Selected entries are retrieved directly without any access to the vector or the associated table.
Advantages	► The presummarized results are readily available. ► It only processes the unique values in the symbol table, avoiding processing table records. ► It extracts all of the data from the index unique key values, eliminating the need for a table probe or vector scan.
Considerations	There is dramatic performance improvement for grouping queries where the resulting number of groups is relatively small compared to the number of records in the underlying table. It can perform poorly when there are a large number of groups involved such that the symbol table is very large, especially if a large portion of symbol table has been put into the overflow area.
Likely to be used	► When asking for GROUP BY, DISTINCT, COUNT, or COUNT DISTINCT from a single table and the referenced column or columns are in the key definition ► When the number of unique values in the column or columns of the key definition is small relative to the number of records in the underlying table ► When there is no selection (Where clause) within the query or the selection does not reduce the result set much

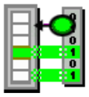
Example SQL statement	`CREATE ENCODED VECTOR INDEX EVI1 ON Sales (Region)`
	Example 1
	`SELECT Region, count(*)`
	`FROM Sales`
	`GROUP BY Region`
	`OPTIMIZE FOR ALL ROWS`
	Example 2
	`SELECT DISTINCT Region`
	`FROM Sales`
	`OPTIMIZE FOR ALL ROWS`
	Example 3
	`SELECT COUNT(DISTINCT Region)`
	`FROM Sales`

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

Encoded-vector index probe

The EVI is quickly probed based upon the selection criteria that was rewritten into a series of ranges. It produces either a temporary row number list or bitmap. Table 3-6 outlines the attributes of the EVI index probe.

Table 3-6 Encoded-vector index probe

Data access method	**Encoded-vector index probe**
Visual Explain icon	
Description	The EVI is quickly probed based upon the selection criteria that was rewritten into a series of ranges. It produces either a temporary row number list or bitmap.
Advantages	▸ Only those index entries that match any selection continue to be processed. ▸ It provides quick access to the selected rows. ▸ It returns the row numbers in ascending sequence so that the table probe can be more aggressive in pre-fetching the rows for its operation.
Considerations	EVIs are generally built over a single key. The more distinct the column is and the higher the overflow percentage is, the less advantageous the EVI becomes. EVIs always require a table probe to be performed on the result of the EVI probe operation.
Likely to be used	▸ When the selection columns match the leading key columns of the index ▸ When an EVI exists and savings in reduced I/O against the table justifies the extra cost of probing the EVI and fully populating the temporary row number list

Example SQL statement	`CREATE ENCODED VECTOR INDEX EVI1 ON`
	`     Employee (WorkDept)`
	`CREATE ENCODED VECTOR INDEX EVI2 ON`
	`     Employee (Salary)`
	`CREATE ENCODED VECTOR INDEX EVI3 ON`
	`     Employee (Job)`
	
	`SELECT *`
	`FROM Employee`
	`WHERE WorkDept = 'E01' AND Job = 'CLERK'`
	`AND Salary = 5000`
	`OPTIMIZE FOR 99999 ROWS`

Messages indicating use	▸ Optimizer Debug:
	`CPI4329 -- Arrival sequence was used for file`
	`           EMPLOYEE.`
	`CPI4338 -– 3 Access path(s) used for bitmap`
	`           processing of file EMPLOYEE.`
	▸ PRTSQLINF:
	`SQL4010 -- Table scan access for table 1.`
	`SQL4032 -- Index EVI1 used for bitmap processing`
	`           of table 1.`
	`SQL4032 -- Index EVI2 used for bitmap processing`
	`           of table 1.`
	`SQL4032 -- Index EVI3 used for bitmap processing`
	`           of table 1.`

3.3.2 Temporary result object methods

Temporary objects are created by the optimizer in order to process a query. In general, these temporary objects are internal objects and cannot be accessed by a user. The temporary objects and access methods in this section may be used during an SQE execution.

Hash table creation

The temporary hash table is a temporary object that allows the optimizer to collate the rows based upon a column or set of columns. The hash table can be either scanned or probed by the optimizer to satisfy different operations of the query. Table 3-7 outlines the attributes of hash table creation.

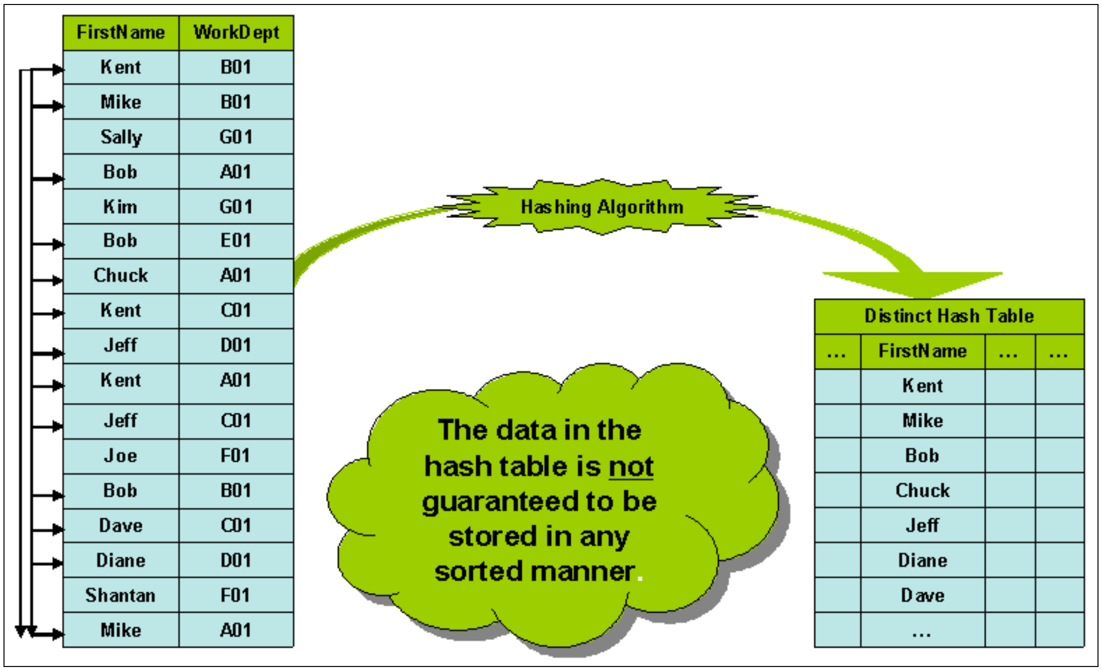
Table 3-7 Hash table creation

Data access method	**Hash table creation**
Visual Explain icon	
Description	Use a hashing algorithm to create a table that collates data with a common value together.
Advantages	► The hash table can be populated with all the rows or only the distinct rows based on the key or keys used for the hashing algorithm. ► The data is organized for efficient retrieval from the hash table, reducing I/O as the query progresses. ► Selection can be performed before generating the hash table to subset the number of rows processed.
Considerations	► A temporary copy of the data is required to be stored within the hash table. ► It can perform poorly when the entire hash table does not stay memory resident as it is being processed. ► High cardinality columns *can* cause frequent collisions on a hash value. Collisions are handled by a linked list. Long lists can degrade overall query throughput.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When the data is required to be collated based upon a column or columns, most typically for grouping, distinct, or join processing

Example SQL statement	`SELECT COUNT(*), FirstName FROM Employee`
	`WHERE WorkDept BETWEEN ‘A01’ AND ‘E01’`
	`GROUP BY FirstName`

Messages indicating use	► Optimizer Debug:
	`CPI4329 - Arrival sequence was used for file EMPLOYEE`
	► PRTSQLINF:
	`SQL4012 - Table scan access for table 1`
	`SQL4029 - Hashing algorithm was used to implement the grouping`
	Note: The SQL4029 message indicates that the hash table being created contains only the grouping column, for example the hash key, FirstName.

Figure 3-9 illustrates the concept of creating and populating a hash table with distinct values from an existing table (or tables).



FirstName	WorkDept
Kent	B01
Mike	B01
Sally	G01
Bob	A01
Kim	G01
Bob	E01
Chuck	A01
Kent	C01
Jeff	D01
Kent	A01
Jeff	C01
Joe	F01
Bob	B01
Dave	C01
Diane	D01
Shantan	F01
Mike	A01

Distinct Hash Table

...	FirstName	...	...
	Kent		
	Mike		
	Bob		
	Chuck		
	Jeff		
	Diane		
	Dave		
	...		

Hashing Algorithm
The data in the hash table is not guaranteed to be stored in any sorted manner.

Figure 3-9 Hash table creation concept

Hash table scan

During a hash table scan operation, the entire temporary hash table is scanned, and all of the entries contained within the hash table are processed. Table 3-8 outlines the attributes of the hash table scan.

Table 3-8 Hash table scan attributes

Data access method	Hash table scan
Visual Explain icon	
Description	It reads all of the entries in a temporary hash table. The hash table may perform distinct processing to eliminate duplicates or takes advantage of the temporary hash table to collate all of the rows with the same value together.
Advantages	► It reduces the random I/O to the table that is generally associated with longer running queries that may otherwise use an index to collate the data. ► Selection can be performed before generating the hash table to subset the number of rows in the temporary object.
Considerations	It is generally used for distinct or group by processing. It can perform poorly when the entire hash table does not stay resident in memory as it is being processed.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When the data is required to be collated based upon a column or columns for distinct or grouping

Data access method	**Hash table scan**

Example SQL statement	`SELECT COUNT(*), FirstNme FROM Employee`
	`WHERE WorkDept BETWEEN 'A01' AND 'E01'`
	`GROUP BY FirstNme`

Messages indicating use	There are multiple ways in which a hash scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine indicate that a hash scan was used.
	► Optimizer Debug:
	`CPI4329 -- Arrival sequence was used for file EMPLOYEE.`
	► PRTSQLINF:
	`SQL4010 -- Table scan access for table 1.`
	`SQL4029 -- Hashing algorithm used to process the grouping.`

Figure 3-10 illustrates the use of a temporary hash table to collate like data into distinct groups. This diagram and those that follow are formatted more like the flowcharts that Visual Explain produces. The symbols that are shown are similar to Visual Explain's symbols, but they are not identical in all cases.

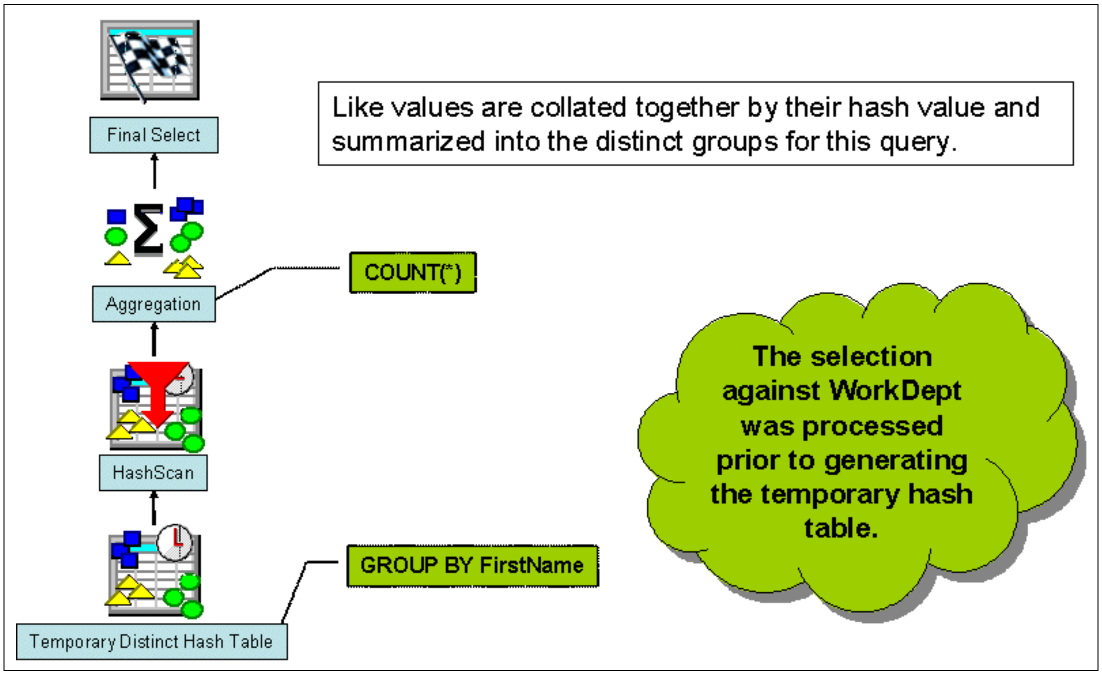

Final Select
Like values are collated together by their hash value and summarized into the distinct groups for this query.
Aggregation
COUNT(*)
HashScan
The selection against WorkDept was processed prior to generating the temporary hash table.
Temporary Distinct Hash Table
GROUP BY FirstName

Figure 3-10 Hash scan

Hash table probe

A hash table probe operation is used to retrieve rows from a temporary hash table based upon a probe lookup operation. Table 3-9 outlines the attributes of the hash table probe.

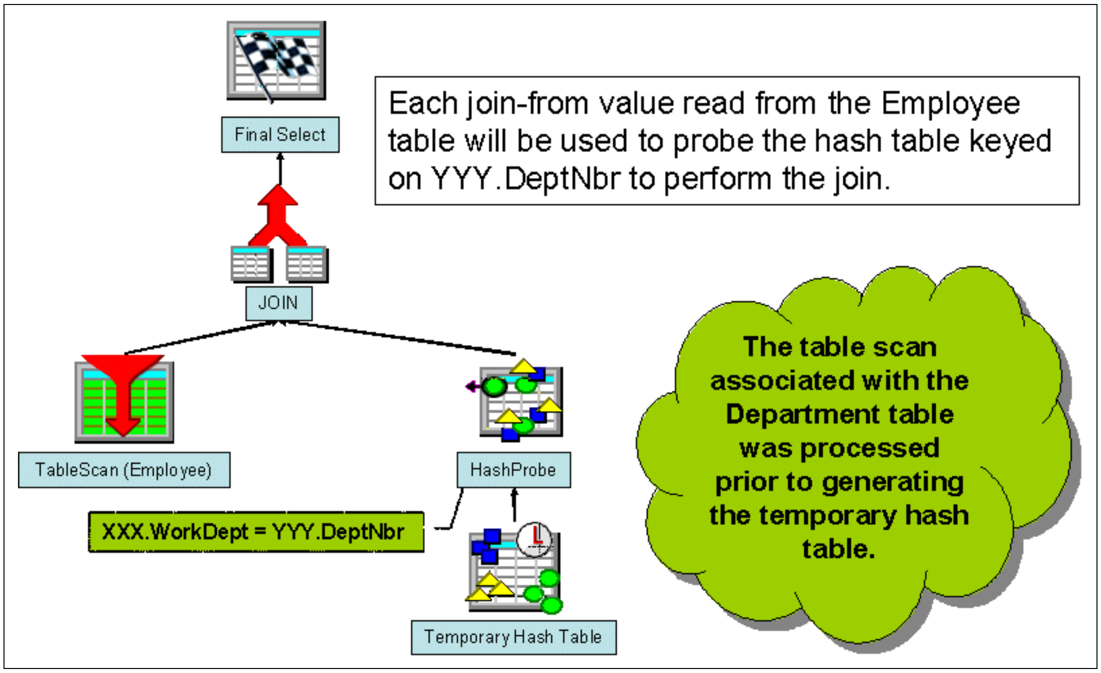
Table 3-9 Hash table probe attributes

Data access method	Hash table probe
Visual Explain icon	
Description	It quickly probes into a temporary hash table to subset the entries to only those that match the selection or join criteria.
Advantages	► It provides quick access to the selected rows that match the probe criteria. ► It reduces the random I/O to the table that is generally associated with longer running queries that use an index to collate the data. ► Selection can be performed before generating the hash table to subset the number of rows in the temporary object.
Considerations	It is generally used to process equal join criteria. It can perform poorly when the entire hash table does not stay resident in memory as it is being processed.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When data is required to be collated based upon a column or columns for join processing ► When the join criteria was specified using an equals (=) operator

Example SQL statement	`SELECT * FROM Employee XXX, Department YYY`
	`WHERE XXX.WorkDept = YYY.DeptNbr`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	There are multiple ways in which a hash probe can be indicated through messages. The messages in this example illustrate how the SQL Query Engine indicates that a hash probe was used.
	► Optimizer Debug:
	`CPI4327 -- File EMPLOYEE processed in join`
	`            position 1.`
	`CPI4327 -- File DEPARTMENT processed in join`
	`            position 2.`
	► PRTSQLINF:
	`SQL4007 -- Query implementation for join`
	`            position 1 table 1.`
	`SQL4010 -- Table scan access for table 1.`
	`SQL4007 -- Query implementation for join`
	`            position 2 table 2.`
	`SQL4010 -- Table scan access for table 2.`

Figure 3-11 illustrates the use of the probe access method with a hash table.



Each join-from value read from the Employee table will be used to probe the hash table keyed on YYY.DeptNbr to perform the join.

The table scan associated with the Department table was processed prior to generating the temporary hash table.

Figure 3-11 Hash table probe

Sorted list creation

The temporary sorted list is a temporary object that allows the optimizer to sequence rows based upon a column or set of columns. The sorted list can be either scanned or probed by the optimizer to satisfy different operations of the query. Table 3-10 outlines the attributes of the sorted list creation.

Table 3-10 Sorted list creation

Data access method	**Sorted list creation**
Visual Explain icon	
Description	A sort routine is used to create a sorted list to order or sequence data with a common value together.
Advantages	► The sorted list can be populated with all the rows or only the distinct rows based on the key or keys used for the sort algorithm. ► Selection can be performed prior to generating the sorted list to subset the number of rows processed.
Considerations	► Two passes are made through the data to populate and then sort the values. ► A temporary copy of the data is required to be stored within the sorted list.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When the data is required to be ordered based upon a column or columns that are most typically for ordering or distinct processing

Data access method	Sorted list creation

Example SQL statements	`CREATE INDEX X1 ON Employee (LastName, WorkDept)`
	`SELECT *FROM Employee`
	`WHERE WorkDept BETWEEN ‘A01’ AND ‘E01’`
	`ORDER BY FirstName`

Messages indicating use	► Optimizer Debug:
	`CPI4328 - Access path of file X1 was used by query`
	`No explicit reference that a sort is used. Check monitor data messages or VE diagram`
	► PRTSQLINF:
	`SQL4002 - Reusable ODP sort used`
	`SQL4008 - Index X1 used for table 1`

Figure 3-12 illustrates the creation of a temporary sorted list.

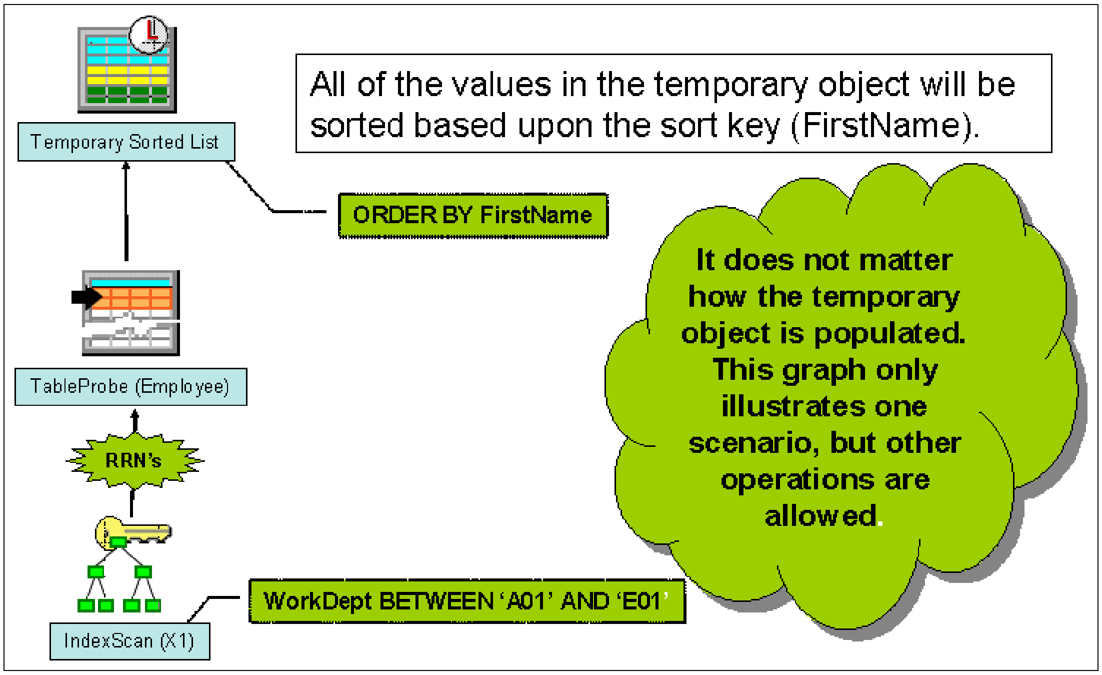

Temporary Sorted List
All of the values in the temporary object will be sorted based upon the sort key (FirstName).
ORDER BY FirstName
It does not matter how the temporary object is populated. This graph only illustrates one scenario, but other operations are allowed.
TableProbe (Employee)
RRN's
WorkDept BETWEEN ‘A01’ AND ‘E01’
IndexScan (X1)

Figure 3-12 Sorted list creation

Sorted list scan

During a sorted list scan operation, the entire temporary sorted list is scanned, and all of the entries contained within the sorted list are processed. Table 3-11 outlines the attributes of the sorted list scan.

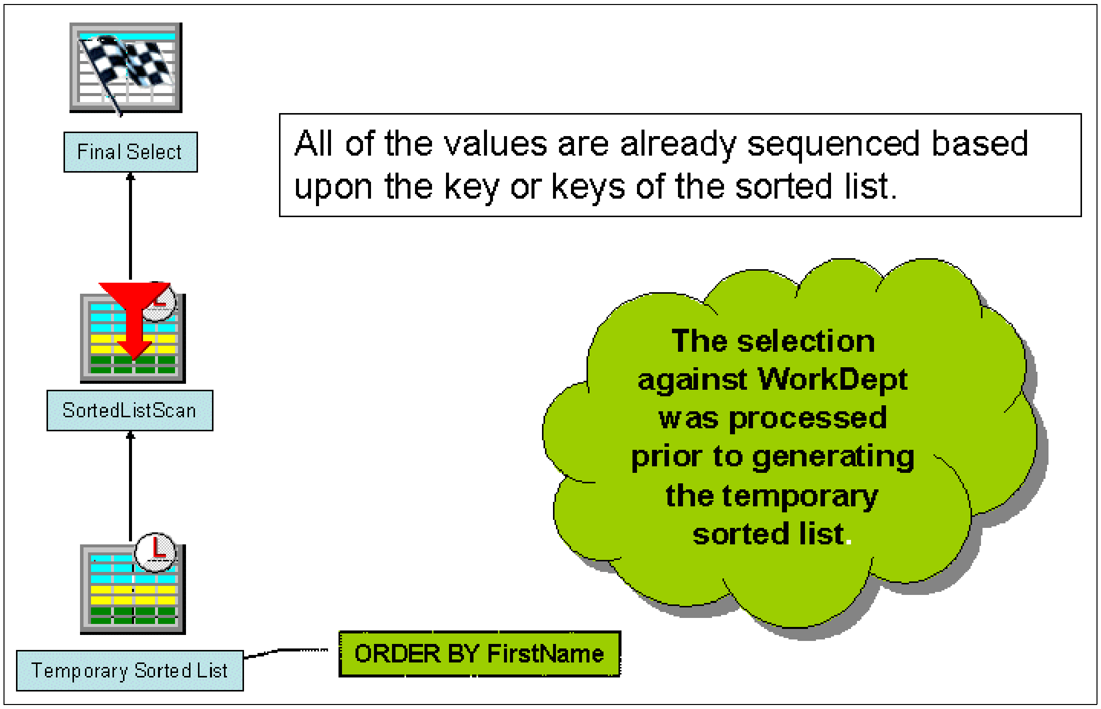
Table 3-11 Sorted list scan

Data access method	**Sorted list scan**
Visual Explain icon	
Description	This scan reads all of the entries in a temporary sorted list. The sorted list may perform distinct processing to eliminate duplicate values or take advantage of the temporary sorted list to sequence all of the rows.
Advantages	► It reduces the random I/O to the table that is generally associated with longer running queries that otherwise uses an index to sequence the data. ► Selection can be performed prior to generating the sorted list to subset the number of rows in the temporary object.
Considerations	It is generally used to process ordering or distinct processing. It can perform poorly when the entire sorted list does not stay resident in memory as it is being populated and processed.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When the data is required to be ordered based upon a column or columns for ordering or distinct processing

Example SQL statements	`CREATE INDEX X1 ON Employee (LastName, WorkDept)`
	
	`SELECT * FROM Employee`
	`WHERE WorkDept BETWEEN 'A01' AND 'E01'`
	`ORDER BY FirstNme`
	`OPTIMZE FOR ALL ROWS`

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

Figure 3-13 shows a diagram of how a sorted list scan may look.



All of the values are already sequenced based upon the key or keys of the sorted list.

The selection against WorkDept was processed prior to generating the temporary sorted list.

Figure 3-13 Sorted list scan

Sorted list probe

A sorted list probe operation is used to retrieve rows from a temporary sorted list based upon a probe lookup operation. Table 3-12 outlines the attributes of the sorted list probe.

Table 3-12 Sorted list probe attributes

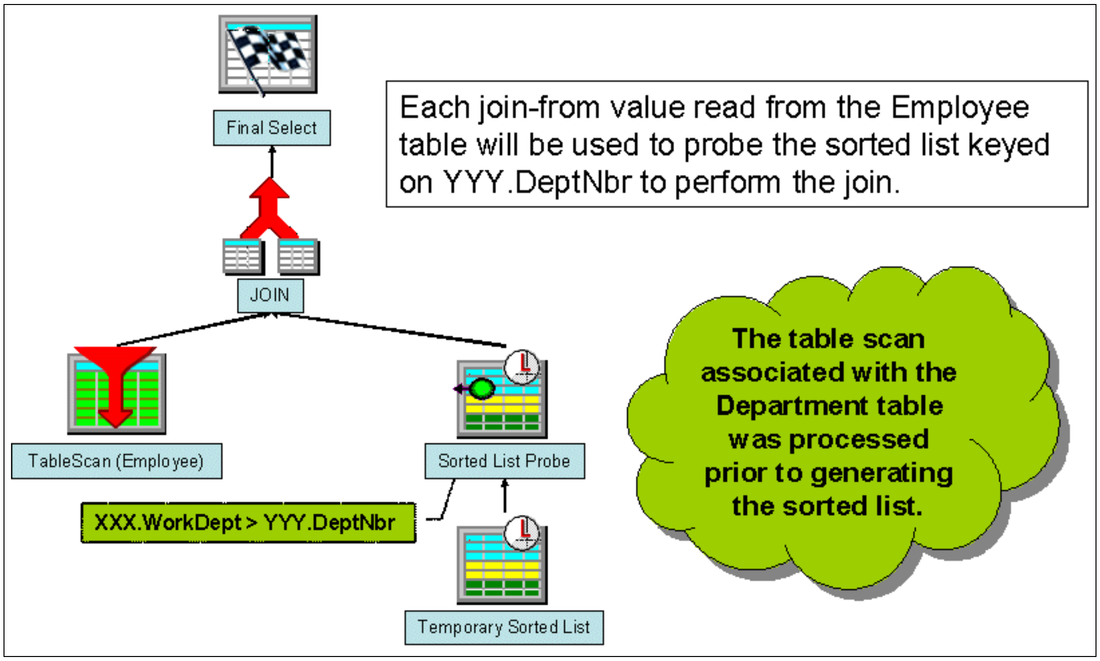
Data access method	**Sorted list probe**
Visual Explain icon	
Description	The temporary sorted list is quickly probed based upon the join criteria.
Advantages	► It provides quick access to the selected rows that match probe criteria. ► It reduces the random I/O to the table that is generally associated with longer running queries that otherwise use an index to collate the data. ► Selection can be performed before generating the sorted list to subset the number of rows that are in the temporary object.
Considerations	It is generally used to process non-equal join criteria. It can perform poorly when the entire sorted list does not stay resident in memory as it is being populated and processed.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When the data is required to be collated based upon a column or columns for join processing ► When the join criteria was specified using a non-equals operator

Data access method	Sorted list probe

Example SQL statements	`SELECT * FROM Employee XXX, Department YYY`
	`WHERE XXX.WorkDept > YYY.DeptNbr`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	There are multiple ways in which a sorted list probe can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine will indicate a sorted list probe was used.
	► Optimizer Debug:
	`CPI4327 -- File EMPLOYEE processed in join position 1.`
	`CPI4327 -- File DEPARTMENT processed in join position 2.`
	► PRTSQLINF:
	`SQL4007 -- Query implementation for join position 1 table 1.`
	`SQL4010 -- Table scan access for table 1.`
	`SQL4007 -- Query implementation for join position 2 table 2.`
	`SQL4010 -- Table scan access for table 2.`

Figure 3-14 illustrates the use of the sorted list probe access method.



Each join-from value read from the Employee table will be used to probe the sorted list keyed on YYY.DeptNbr to perform the join.

The table scan associated with the Department table was processed prior to generating the sorted list.

Figure 3-14 Sorted list probe

Unsorted list creation

The temporary unsorted list is a temporary object that allows the optimizer to store intermediate results of a query. The list is an unsorted data structure that is used to simplify the operation of the query. Since the list does not have any keys, the rows within the list can only be retrieved by a sequential scan operation. Table 3-13 outlines the attributes of the unsorted list creation.

Table 3-13 Unsorted list creation attributes

Data access method	**Unsorted list creation**
Visual Explain icon	
Description	An unsorted list or temporary table is created to store data for an intermediate operation of the query. No key is associated with this object.
Advantages	Selection can be performed prior to generating the unsorted list to subset the number of rows processed for each iteration.
Considerations	► No key is associated with this object, so the entire list must be scanned to process any rows. ► A temporary copy of the data is required to be stored within the unsorted list.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When the query contains a complex operation that requires the engine to break up the request into multiple steps to complete the query, most typically associated with complex views, grouping, join queries, or symmetric multiprocessing (SMP)

Example SQL statements	`SELECT XXX.* FROM Employee XXX, Department YYY`
	`WHERE XXX.WorkDept > YYY.DeptNbr`
	`GROUP BY YYY.DeptName, XXX.WorkDept`

Messages indicating use	► Optimizer Debug:
	`CPI4325 - Temporary result file built for query`
	► PRTSQLINF:
	`SQL4001 - Temporary result created.`
	`SQL4007 - Query implementation for join position 1 table 1`
	`SQL4010 - Table scan access for table 1`
	`SQL4007 - Query implementation for join position 2 table 2`

Figure 3-15 represents the creation of a temporary unsorted list.

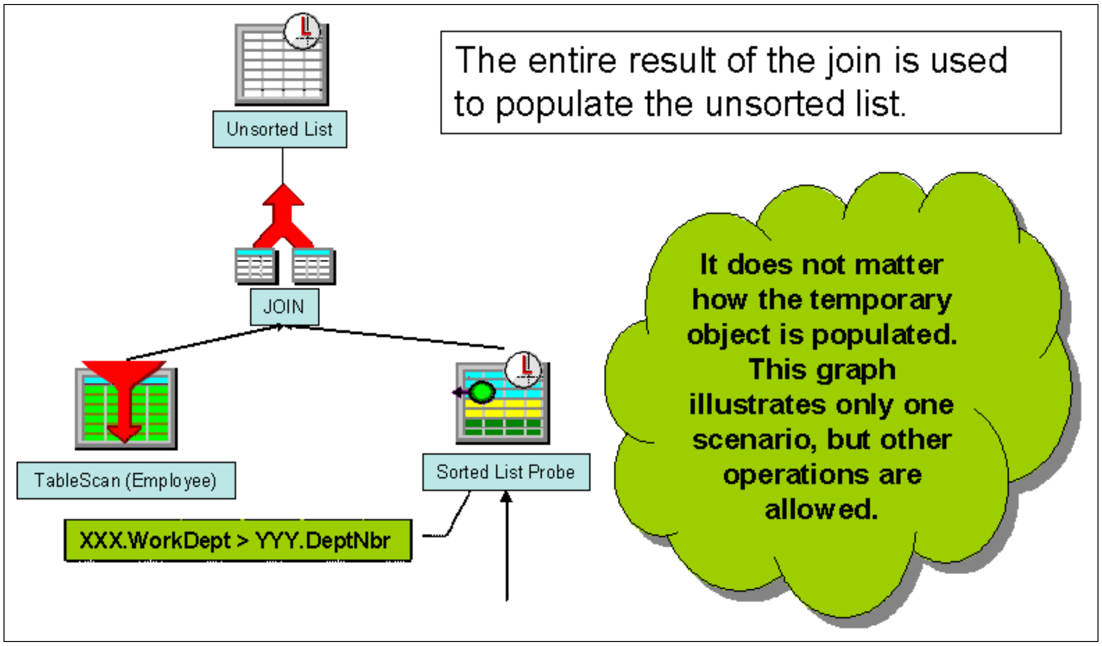

The entire result of the join is used to populate the unsorted list.
Unsorted List
JOIN
TableScan (Employee)
Sorted List Probe
XXX.WorkDept > YYY.DeptNbr
It does not matter how the temporary object is populated. This graph illustrates only one scenario, but other operations are allowed.

Figure 3-15 Unsorted list creation

Unsorted list scan

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

Table 3-14 outlines the attributes of the unsorted list scan.

Table 3-14 Unsorted list scan attributes

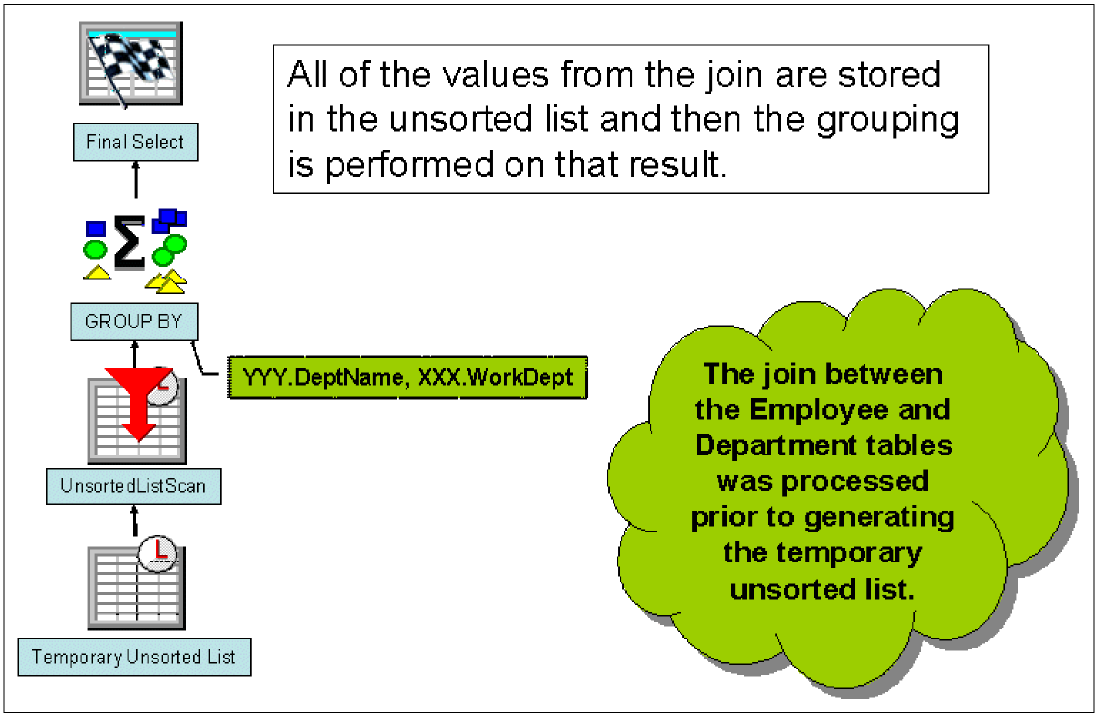
Data access method	**Unsorted list scan**
Visual Explain icon	
Description	This method sequentially scans and processes all of the rows in the temporary list.
Advantages	► The temporary list and list scan can be used by the optimizer to minimize repetition of an operation or to simplify the optimizer's logic flow. ► Selection can be performed before generating the list to subset the number of rows in the temporary object.
Considerations	It is generally used to prevent portions of the query from being processed multiple times when no key columns are required to satisfy the request.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When SMP will be used for the query

Data access method	**Unsorted list scan**

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

Messages indicating use	There are multiple ways in which a list scan can be indicated through the messages. The messages in this example illustrate how the SQL Query Engine indicates that a list scan was used.
	▶ Optimizer Debug:
	`CPI4325 -- Temporary result file built for query.`
	`CPI4327 -- File EMPLOYEE processed in join position 1.`
	`CPI4327 -- File DEPARTMENT processed in join position 2.`
	▶ PRTSQLINF:
	`SQL4007 -- Query implementation for join position 1 table 1.`
	`SQL4010 -- Table scan access for table 1.`
	`SQL4007 -- Query implementation for join position 2 table 2.`
	`SQL4001 -- Temporary result created`
	`SQL4010 -- Table scan access for table 2`

Figure 3-16 illustrates the process of an unsorted list scan.



All of the values from the join are stored in the unsorted list and then the grouping is performed on that result.

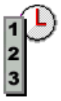
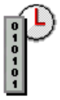
The join between the Employee and Department tables was processed prior to generating the temporary unsorted list.

Figure 3-16 Unsorted list scan

Address list creation (RRN or bitmap)

The temporary row number list (address list) is a temporary object that allows the optimizer to sequence rows based upon their row address (their row number). The row number list can be either scanned or probed by the optimizer to satisfy different operations of the query. In RRN and bitmaps, both methods (scan and probe) act in the same way. Table 3-15 outlines the attributes of the address list creation.

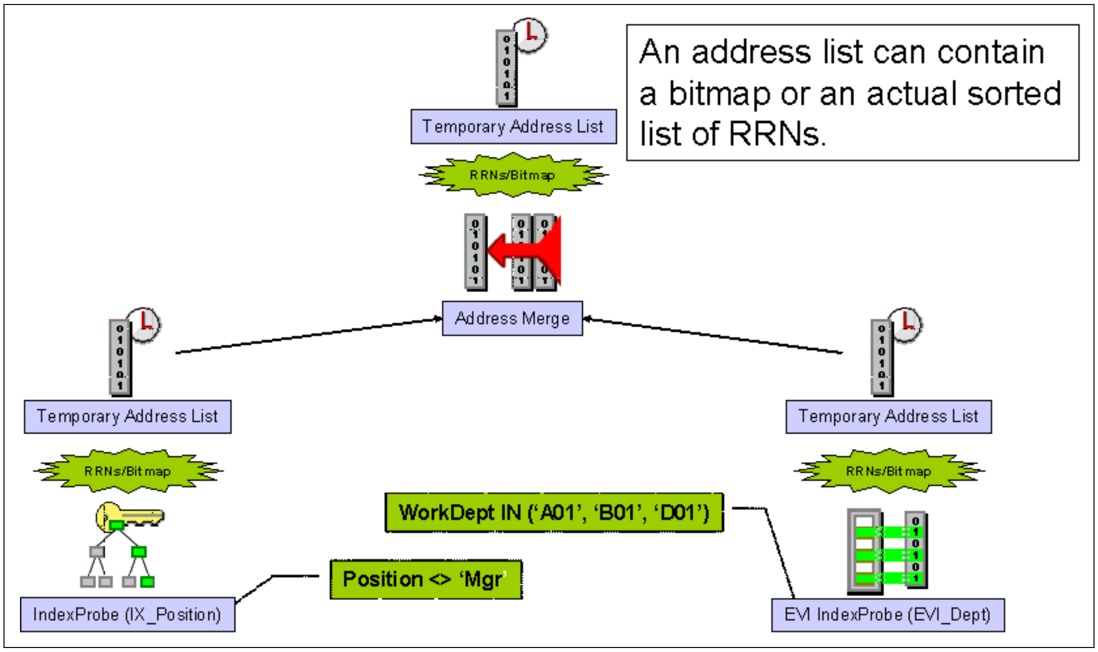
Table 3-15 Temporary row number list (address list)

Data access method	Address list creation
Visual Explain icon (RRN)	
Visual Explain icon (Bitmap)	
Description	This method generate a list of RRN values or a bitmap that represents the rows from a table that match the selection.
Advantages	► Selection can be applied against a variety of indexes using ANDing and ORing logic to best mix and match the indexes that exist over the table. ► No I/O is generated against the table while the address list is generated. ► An RRN list is ordered based on the RRN values, or the bits in a bitmap represent relative row locations within the table.
Considerations	The address list must be fully populated before any rows can be retrieved from the temporary object.
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When the query contains selection that matches an existing index over the table

Example SQL statement	`CREATE INDEX IX_Position ON Employee (Position)`
	`CREATE ENCODED VECTOR INDEX EVI_Dept ON Employee (WorkDept)`
	
	`SELECT * FROM Employee`
	`WHERE Position <> ‘Mgr’`
	`AND WorkDept IN (‘A01’, ‘B01’, ‘D01’)`
	`AND Salary BETWEEN 50000 AND 100000`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	► Optimizer Debug:
	`CPI432C - All access paths were considered for file EMPLOYEE`
	`CPI4338 - 2 Access path(s) used for bitmap processing of file EMPLOYEE`
	► PRTSQLINF:
	`SQL4010 - Table scan access for table 1`
	`SQL4032 - Index IX_Position used for bitmap processing of table 1`
	`SQL4032 - Index IX_EVI_Dept used for bitmap processing of table 1`

Figure 3-17 shows an example of creating a temporary address list (either RRNs or bitmaps).



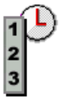
An address list can contain a bitmap or an actual sorted list of RRNs.

Figure 3-17 Address list creation

Address list scan (RRN or bitmap)

During a row number list scan operation (address list scan), the entire temporary row number list is scanned, and all of the row addresses contained within the row number list are processed. A row number list scan is generally considered when the optimizer is considering a plan that involves an EVI or if the cost of the random I/O associated with an index probe or scan operation can be reduced by first preprocessing and sorting the row numbers associated with the table probe operation. Table 3-16 outlines the attributes of the address list scan.

Table 3-16 Address list scan

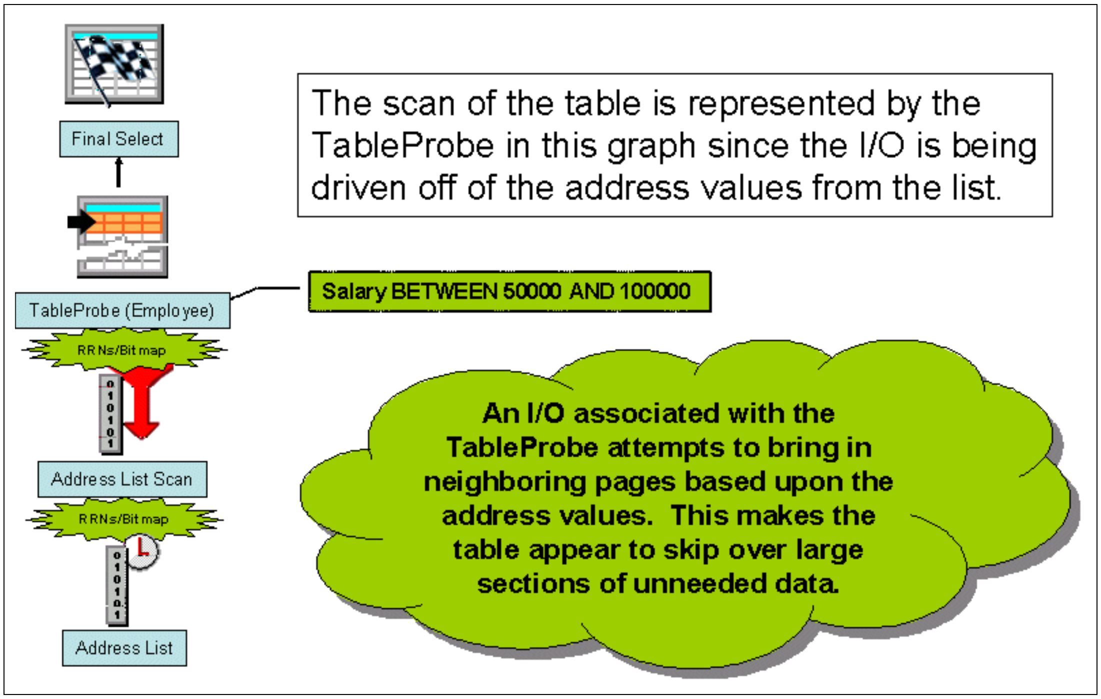
Data access method	**Address list scan**
Visual Explain icon (RRN)	
Visual Explain icon (Bitmap)	
Description	This method scans the list of address values from the temporary object. Since the address values are sorted, the table is sequentially processed.
Advantages	► It minimizes page I/O operations by paging larger sections of the table when the address values are clustered, for example groups of consecutive or nearly consecutive addresses. ► The majority of the selection can be applied prior to any table I/O operations.
Considerations	The address list is a static object and is not updated for the duration of the query.

Data access method	Address list scan
Likely to be used	► When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ► When the savings generated from minimizing I/O operations outweighs the cost to generate and process the address list

Example SQL statement	`SELECT * FROM Employee`
	`WHERE Position <> ‘Mgr’`
	`AND WorkDept IN (‘A01’, ‘B01’, ‘D01’)`
	`AND Salary BETWEEN 50000 AND 100000`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	► Optimizer Debug:
	`CPI432C - All access paths were considered for file EMPLOYEE`
	`CPI4338 - 2 Access path(s) used for bitmap processing of file EMPLOYEE`
	► PRTSQLINF:
	`SQL4010 - Table scan access for table 1`
	`SQL4032 - Index IX_Position used for bitmap processing of table 1`
	`SQL4032 - Index IX_EVI_Dept used for bitmap processing of table 1`
	
	Note: The scan of the address list is not explicitly identified in any of the messages generated by this query.

Figure 3-18 illustrates the use of an address list scan access method.



The scan of the table is represented by the TableProbe in this graph since the I/O is being driven off of the address values from the list.

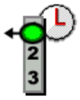
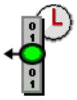
An I/O associated with the TableProbe attempts to bring in neighboring pages based upon the address values. This makes the table appear to skip over large sections of unneeded data.

Figure 3-18 Address list scan

Address list probe

A row number list probe (address list probe) operation is used to test row numbers that are generated by a separate operation against the selected rows of a temporary row number list. The row numbers can be generated by any operation that constructs a row number for a table. That row number is then used to probe into a temporary row number list to determine if that row number matches the selection used to generate the temporary row number list. Table 3-17 outlines the attributes of the address list probe.

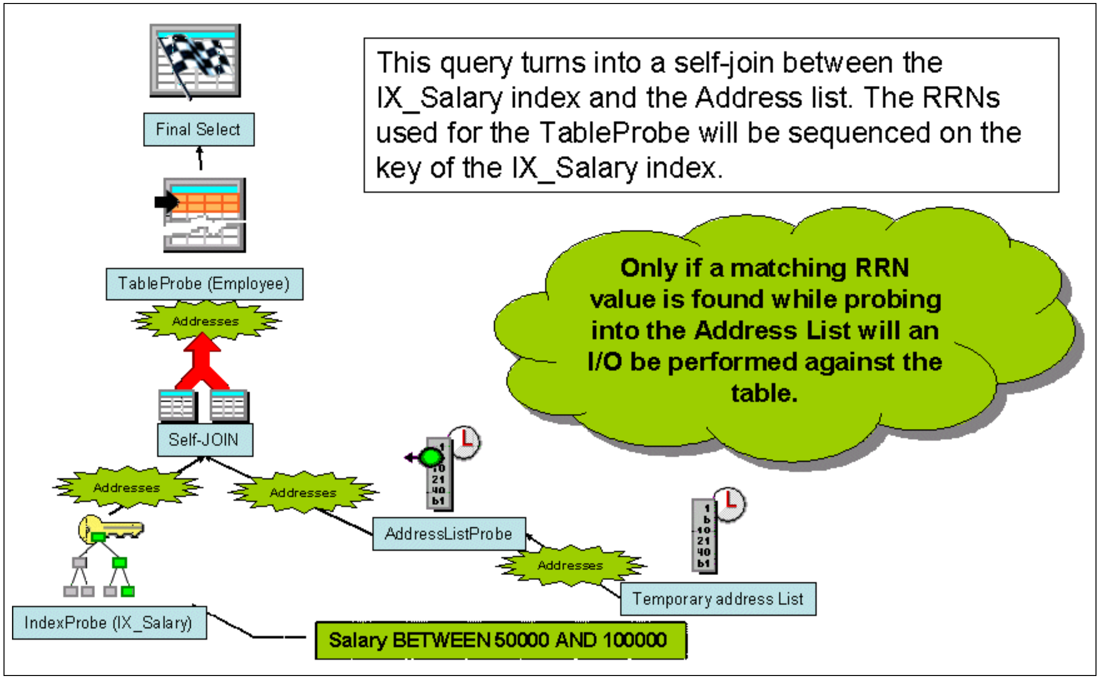
Table 3-17 Address list probe attributes

Data access method	**Address list probe**
Visual Explain icon (RRN)	
Visual Explain icon (Bitmap)	
Description	Using an RRN value generated from a primary index, this method probes into the temporary list of address values to determine if a matching address is found.
Advantages	It allows an index to be used to sequence the data and still use additional indexes to perform any available selection.
Considerations	The address list is a static object and is not updated for the duration of the query.
Likely to be used	▸ When the use of temporary results is allowed by the query environmental parameter (ALWCPYDTA) ▸ When the savings generated from minimizing I/O operations outweighs the cost to generate and process the address list ▸ When the data is required to be ordered based upon a column or columns that match up to an index, but does not cover all of the available selection

Example SQL statement	`CREATE INDEX IX_Salary ON Employee (Salary)`
	
	`SELECT * FROM Employee`
	`WHERE Position <> ‘Mgr’`
	`AND WorkDept IN (‘A01’, ‘B01’, ‘D01’)`
	`AND Salary BETWEEN 50000 AND 100000`
	`ORDER BY Salary`

Messages indicating use	▸ Optimizer Debug:
	`CPI4328 - Access path of file IX_Salary was used by query`
	`CPI4338 - 2 Access path(s) used for bitmap processing of file EMPLOYEE`
	▸ PRTSQLINF:
	SQL4008 - Index IX_Salary used for table 1
	SQL4011 - Index scan-key row positioning used on table 1
	SQL4032 - Index IX_Position used for bitmap processing of table 1
	SQL4032 - Index IX_EVI_Dept used for bitmap processing of table 1
	Note: The index probe against IX_Salary is used as the primary access method for this query to satisfy the ordering requirement.

Figure 3-19 illustrates the use of an address list probe access method.



This query turns into a self-join between the IX_Salary index and the Address list. The RRNs used for the TableProbe will be sequenced on the key of the IX_Salary index.

Only if a matching RRN value is found while probing into the Address List will an I/O be performed against the table.

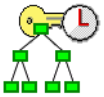
Figure 3-19 Address list probe

Temporary index scan

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

Additionally, the temporary index is optimized for use by the optimizer to satisfy a specific query request. This includes setting the logical page size and applying any selection to the creation to speed up the use of the temporary index after it has been created. Table 3-18 outlines the attributes of the temporary index scan.

Table 3-18 Temporary index scan

Data access method	**Temporary index scan**
Visual Explain icon	
Description	This method sequentially scans and processes all of the keys associated with the temporary index.
Advantages	► It has the potential to extract all of the data from the index keys' values, thus eliminating the need for a table probe. ► It returns the rows back in a sequence based upon the keys of the index.
Considerations	It generally requires a table probe to be performed to extract any remaining columns that are required to satisfy the query. It can perform poorly when a large number of rows is selected because of the random I/O associated with the table probe.

Data access method	Temporary index scan
Likely to be used	► When sequencing the rows is required for the query (for example, ordering or grouping) ► When the selection columns cannot be matched against the leading key columns of the index ► When the overhead cost associated with the creation of the temporary index can be justified against other alternative methods to implement this query

Example SQL Statement	`SELECT * FROM Employee`
	`WHERE WorkDept BETWEEN 'A01' AND 'E01'`
	`ORDER BY LastName`
	`OPTIMIZE FOR ALL ROWS`

Messages indicating use	► Optimizer Debug:
	`CPI4321 -- Access path built for file EMPLOYEE.`
	► PRTSQLINF:
	`SQL4009 -- Index created for table 1`

Temporary index probe

A temporary index probe operation is identical to the index probe operation that is performed upon the permanent radix index. Its main function is to provide a form of quick access against the index keys of the temporary index; however it can still used to retrieve the rows from a table in a keyed sequence. Table 3-19 outlines the attributes of the temporary index probe.

Table 3-19 Temporary index probe

Data access method	Temporary index probe
Description	The index is quickly probed based upon the selection criteria that was rewritten into a series of ranges. Only those keys that satisfy the selection will be used to generate a table row number.
Advantages	► Only those index entries that match any selection continue to be processed. It provides quick access to the selected rows. ► It has the potential to extract all of the data from the index keys' values, thus eliminating the need for a table probe. ► It returns the rows back in a sequence based upon the keys of the index.
Considerations	It generally requires a table probe to be performed to extract any remaining columns that are required to satisfy the query. It can perform poorly when a large number of rows is selected because of the random I/O associated with the table probe.
Likely to be used	► When the ability to probe the rows required for the query (for example, joins) exists ► When the selection columns cannot be matched against the leading key columns of the index ► When the overhead cost associated with the creation of the temporary index can be justified against other alternative methods to implement this query

Example SQL Statement	`SELECT * FROM Employee XXX, Department YYY`
	`WHERE XXX.WorkDept = YYY.DeptNo`
	`OPTIMIZE FOR ALL ROWS`

Data access method	Temporary index probe

Messages indicating use	► Optimizer Debug:
	`CPI4321 -- Access path built for file DEPARTMENT.`
	`CPI4327 -- File EMPLOYEE processed in join position 1.`
	`CPI4326 -- File DEPARTMENT processed in join position 2.`
	► PRTSQLINF:
	`SQL4007 -- Query implementation for join position 1 table 1.`
	`SQL4010 -- Table scan access for table 1.`
	`SQL4007 -- Query implementation for join position 2 table 2.`
	`SQL4009 -- Index created for table 2.`

Buffer scan

The temporary buffer is a temporary object that is used to help facilitate operations such as parallelism. It is an unsorted data structure that is used to store intermediate rows of a query. The main difference between a temporary buffer and a temporary list is that the buffer does not need to be fully populated in order to allow its results to be processed. Table 3-20 outlines the attributes of the buffer scan.

Table 3-20 Buffer scan

Data access method	Buffer scan
Visual Explain icon	
Description	This method sequentially scans and processes all of the rows in the temporary buffer. It enables SMP parallelism to be performed over a non-parallel portion of the query.
Advantages	► The temporary buffer can be used to enable parallelism over a portion of a query that is non-parallel. ► The temporary buffer does not need to be fully populated in order to start fetching rows.
Considerations	It is generally used to prevent portions of the query from being processed multiple times when no key columns are required to satisfy the request.
Likely to be used	► When the query is attempting to take advantage of DB2 Universal Database Symmetric Multiprocessing ► When a portion of the query cannot be performed in parallel (for example, index scan or index probe)

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

Data access method	**Buffer scan**

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

3.3.3 Access methods in V5R4

A *queue* is new temporary object added in V5R4. Unlike other temporary objects created by the optimizer, the queue is not populated in all at once by the underlying query node tree. Instead it is really a real-time temporary holding area for values that feed the recursion. In this regard, a queue is not considered temporary because it does not prevent the query from running if ALWCPYDTA(*NO) was specified. This is because the data can still flow up and out of the query at the same time that the recursive values are inserted into the queue to be used to retrieve additional join rows.

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

The queue has two operations allowed:

- Enqueue, which puts data on the queue
- Dequeue, which takes data off the queue

Enqueue

During an enqueue operation, an entry is placed on the queue that contains key values that are used by the recursive join predicates or data manipulated as a part of the recursion process. The optimizer always supplies an enqueue operation to collect the required recursive data on the query node directly above Union All. Table 3-21 outlines the attributes of the enqueue operation.

Table 3-21 Enqueue attributes

Data access method	**Enqueue**
Visual Explain icon	
Description	This operation places an entry on the queue needed to cause further recursion.
Advantages	► Required as a source for the recursion. Only enqueues required values for the recursion process. Each entry has short life span, until it is dequeued. ► Each entry on the queue can seed multiple iterative fullselects that are recursive from the same Recursive Common Table Expression (RCTE)/view.
Likely to be used	For a required access method for recursive queries

<table>
<tr><th>Data access method</th><th>Enqueue</th></tr>
<tr><td>Example SQL Statement</td><td><pre>
WITH RPL (PART, SUBPART, QUANTITY) AS
     (SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
        FROM PARTLIST ROOT
        WHERE ROOT.PART = '01'
      UNION ALL
        SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
        FROM RPL PARENT, PARTLIST CHILD
        WHERE PARENT.SUBPART = CHILD.PART
)
SELECT DISTINCT PART, SUBPART, QUANTITY
 FROM RPL
</pre></td></tr>
<tr><td>Messages indicating use</td><td>There are no explicit messages that indicate the use of an enqueue.</td></tr>
</table>

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

Use the SEARCH option in the definition of the recursive query to return the results of the recursion in the specified parent-child hierarchical ordering. The search choices are Depth or Breadth first. Depth first means that all the descendents of each immediate child are returned before the next child is returned. Breadth first means that each child is returned before their children are returned. SEARCH requires the specification of the relationship keys, which columns make up the parent child relationship and the search type of Depth or Breadth. It also requires an ORDER BY clause in the main query on the provided sequence column in order to fully implement the specified ordering.

Dequeue

During a dequeue operation, an entry is taken off the queue, and those values specified by recursive reference are fed back into the recursive join process. The optimizer always supplies a corresponding enqueue, dequeue pair of operations for each reference of a recursive common table expression or recursive view in the specifying query. Recursion ends when there are no more entries to pull off the queue. Table 3-22 outlines the attributes of the dequeue operation.

Table 3-22 Dequeue attributes

Data access method	Dequeue
Visual Explain icon	
Description	This operation removes an entry off the queue. It provides minimally one side of the recursive join predicate that feeds the recursive join and other data values that are manipulated through the recursive process. The dequeue is always on the left side of inner join with a constraint, where the right side of the join contains the target child rows.
Advantages	► It provides quick access to recursive values. ► It allows for post selection of local predicate on recursive data values.

Data access method	**Dequeue**
Likely to be used	► When there is a required access method for recursive queries ► When single dequeued values can feed the recursion of multiple iterative full selects that reference the same Recursive Common Table Expression (RCTE)/view

Example SQL Statement	`WITH RPL (PART, SUBPART, QUANTITY) AS`
	`     (  SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY`
	`        FROM PARTLIST ROOT`
	`        WHERE ROOT.PART = '01'`
	`      UNION ALL`
	`        SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY`
	`        FROM RPL PARENT, PARTLIST CHILD`
	`        WHERE  PARENT.SUBPART = CHILD.PART`
	`     )`
	`SELECT DISTINCT PART, SUBPART, QUANTITY`
	` FROM RPL`

Messages indicating use	There are no explicit messages that indicate the use of a dequeue.

4

Statistics Manager

Starting with OS/400 V5R2, the responsibility for statistical information was removed from the optimizer and given to a new component called the *Statistics Manager*. In this chapter, we discuss the interaction between the optimizer and Statistics Manager. We introduce the new concept of single-column statistics and how they are created and maintained automatically by the system.

In this chapter, we also explain how to manage statistics using iSeries Navigator. Finally we show why and how to influence collection of column statistics yourself.

4.1 Statistics and statistics collections

Before OS/400 V5R2, the retrieval of statistics was a function of the optimizer. When the optimizer needed to know information about a table, it looked at the table description to retrieve the row count and table size. If an index was available, the optimizer could extract further information about the data in the underlying table from that source. Figure 4-1 illustrates how Classic Query Engine (CQE) relies on indexes for statistics prior to V5R2.

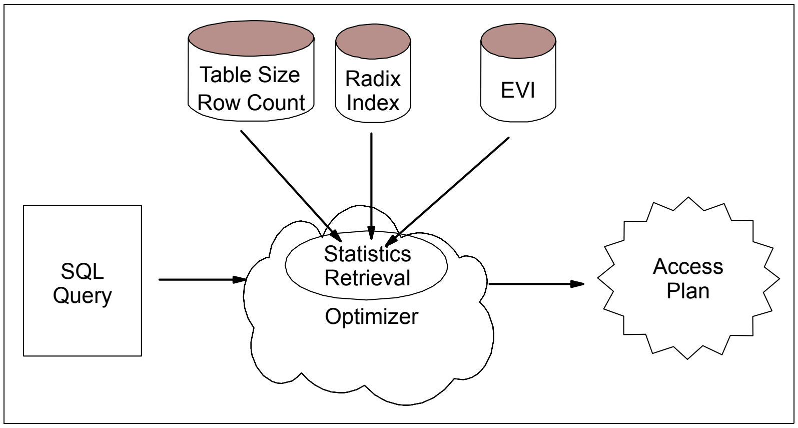


Figure 4-1 Statistics in CQE

The reliance on indexes for meaningful statistics often caused a dependency on creating new indexes to solve performance problems. Creating new indexes to supply statistics impacted disk requirements for the indexes and increased the central processing unit (CPU) load required to maintain the new indexes.

With the introduction of Structured Query Language (SQL) Query Engine (SQE) to OS/400 V5R2, the collection of statistics was removed from the optimizer. It is now handled by a separate component called the *Statistics Manager* (see Figure 4-2). The Statistics Manager has two major functions:

- Create and maintain column statistics
- Answer questions the optimizer asks when finding the best way to implement a given query

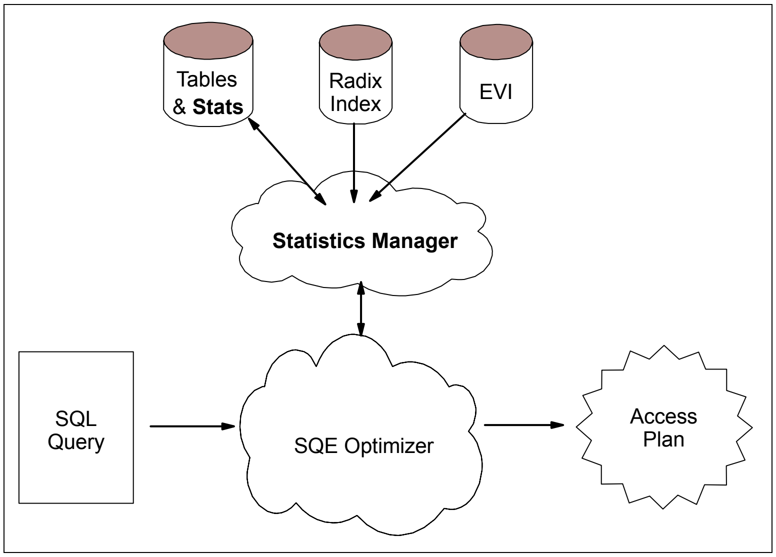


Figure 4-2 SQE Optimizer and Statistics Manager

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 may 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 is available to provide a statistical answer, then the Statistics Manager bases the answer on default values (filter factors).

Important: 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.

4.1.1 Optimizer statistics request types

The optimizer questions the Statistics Manage about the following areas:

- Table and index definitions

  Examples of such questions are: “How many rows are in a certain table?” and “Which indexes are relevant to this query?”

- Filter factors

  These factors are used to estimate the total number of rows that are expected to be returned given specific selection criteria.

- Cardinality

  This area is used to estimate the distinct number of values for a column.

- Number of entries accessed

  The entries are used to estimate the effectiveness of using a particular index as the access method.

- Input/output (I/O) costing

  Costing provides an estimate of how many I/O operations will be performed and how much parallelism can be performed.

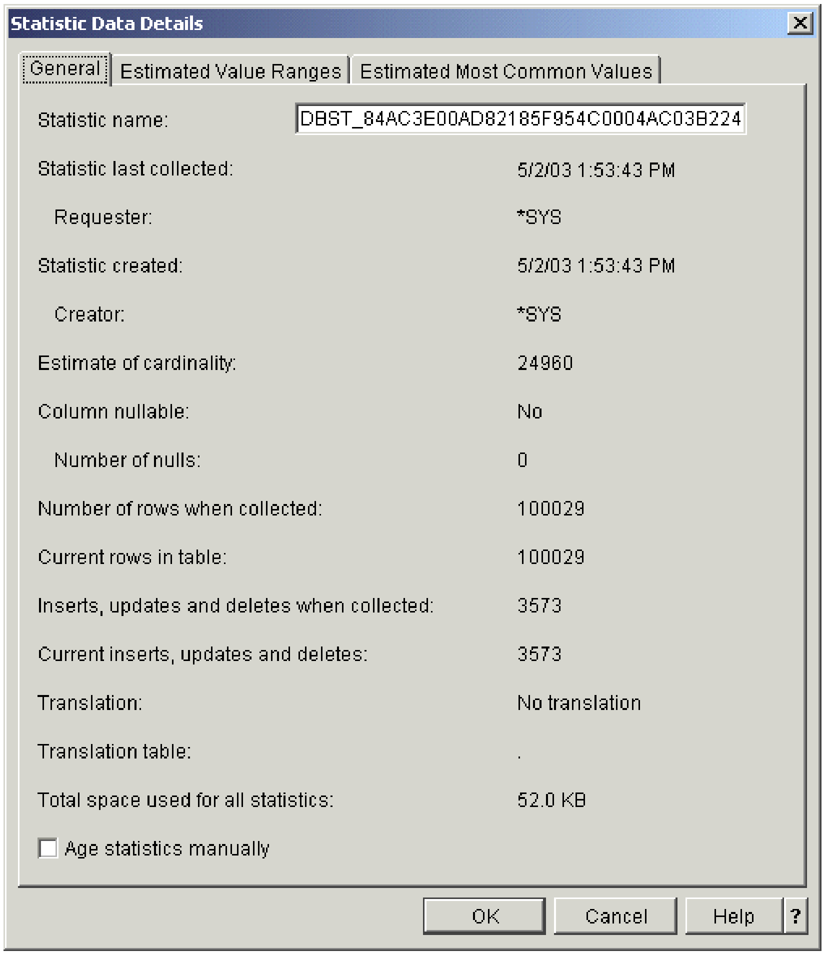
The Statistics Manager must always provide an answer to the questions from the optimizer. It uses the best method that is available to provide the answers. For example, it may use a single-column statistic or perform a key range estimate over an index. Along with the answer, Statistics Manager returns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

4.1.2 Single-column statistics

Single-column statistics provide detailed information about one column within a table. The information is collected by performing a full table scan. Then the collected information is stored with the table.

Each column statistic provides the following information, which is provided in the Statistic Data Details window as shown in the following examples:

- General (Figure 4-3)
  - *Estimate of cardinality*: Provides an estimate of the number of distinct values for the column
  - *Number of nulls*: Provides the number of records with the null value



Statistic name: DBST_84AC3E00AD82185F954C0004AC03B224

Statistic last collected: 5/2/03 1:53:43 PM

Requester: *SYS

Statistic created: 5/2/03 1:53:43 PM

Creator: *SYS

Estimate of cardinality: 24960

Column nullable: No

Number of nulls: 0

Number of rows when collected: 100029

Current rows in table: 100029

Inserts, updates and deletes when collected: 3573

Current inserts, updates and deletes: 3573

Translation: No translation

Translation table: .

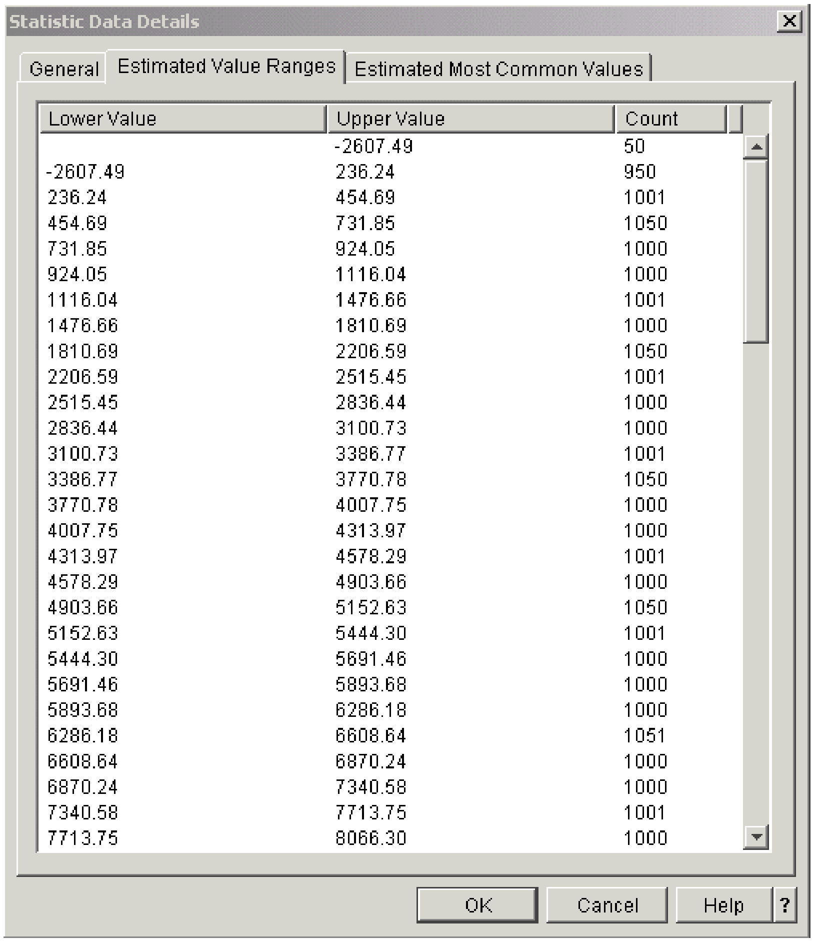
Total space used for all statistics: 52.0 KB

☐ Age statistics manually

Figure 4-3 Cardinality estimate and number of nulls

- Estimated value ranges (Figure 4-4)

  This range shows how the data in a column is spread over the different values. The range is also referred to as a *histogram*. The Statistics Manager tries to separate the rows into equally sized *buckets*. Each bucket has an associated low and high value to determine the range and the estimated number of records in that range. The lower value is exclusive, and the upper value is inclusive.

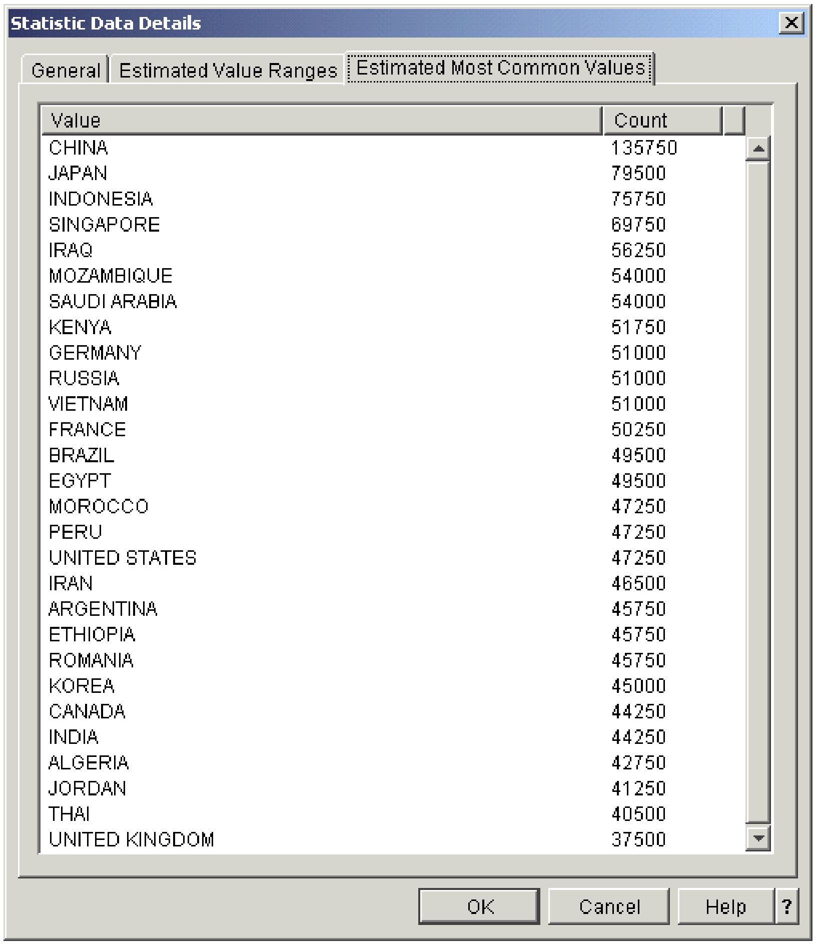


Lower Value	Upper Value	Count
	-2607.49	50
-2607.49	236.24	950
236.24	454.69	1001
454.69	731.85	1050
731.85	924.05	1000
924.05	1116.04	1000
1116.04	1476.66	1001
1476.66	1810.69	1000
1810.69	2206.59	1050
2206.59	2515.45	1001
2515.45	2836.44	1000
2836.44	3100.73	1000
3100.73	3386.77	1001
3386.77	3770.78	1050
3770.78	4007.75	1000
4007.75	4313.97	1000
4313.97	4578.29	1001
4578.29	4903.66	1000
4903.66	5152.63	1050
5152.63	5444.30	1001
5444.30	5691.46	1000
5691.46	5893.68	1000
5893.68	6286.18	1000
6286.18	6608.64	1051
6608.64	6870.24	1000
6870.24	7340.58	1000
7340.58	7713.75	1001
7713.75	8066.30	1000

Figure 4-4 Estimated Value Ranges page

- Estimated frequent values list (Figure 4-5)

  This list provides an estimate of the most frequent values and an estimated count for each value. This information is shown on the Estimated Most Common Values page.



Value	Count
CHINA	135750
JAPAN	79500
INDONESIA	75750
SINGAPORE	69750
IRAQ	56250
MOZAMBIQUE	54000
SAUDI ARABIA	54000
KENYA	51750
GERMANY	51000
RUSSIA	51000
VIETNAM	51000
FRANCE	50250
BRAZIL	49500
EGYPT	49500
MOROCCO	47250
PERU	47250
UNITED STATES	47250
IRAN	46500
ARGENTINA	45750
ETHIOPIA	45750
ROMANIA	45750
KOREA	45000
CANADA	44250
INDIA	44250
ALGERIA	42750
JORDAN	41250
THAI	40500
UNITED KINGDOM	37500

Figure 4-5 Estimated frequent value list

4.1.3 Automatic statistics collection

Most databases use statistics to assist with query optimization. They often rely on a database administrator (DBA) to create the statistics and to refresh the statistics when they become stale or no longer provide a meaningful representation of the data. With the goal of being an easy-to-manage database, Statistics Manager is set, by default, to collect and refresh column statistics automatically. Column statistics are automatically collected after an SQL statement is executed based on statistics requests from the optimizer.

When Statistics Manager prepares its responses to the optimizer, it keeps track of which responses are generated by using default filter factors, because column statistics or indexes were not available. It uses this information while the access plan is written to the plan cache to automatically generate a statistics collection request for such columns. As system resources become available, the requested column statistics are collected in the background. The intent is that the missing column statistics are available on future executions of the query. This allows Statistics Manager to provide more accurate information to the optimizer. More statistics make it easier for the optimizer to generate a good performing access plan.

In the plan cache, information is stored about which column statistics were used and which ones were missing while generating the access plan. If an SQL request is canceled before or during execution (for example, because of the query time limit), these requests for column statistics are created, but only as long as execution reaches the point where the generated access plan is written to the plan cache.

To minimize the number of passes through the table during statistics collection, the Statistics Manager attempts to group multiple requests for the same table together. For example, table T1 may not have any useful indexes or column statistics available. Two queries are executed, with one query having selection criteria on column C1 and the other query having selection criteria on column C2. In this case, Statistics Manager identifies both of these columns as good candidates for column statistics. When Statistics Manager reviews the requests, it looks for multiple requests for the same table and groups them together into one request. This allows both column statistics to be created with only one pass through table T1.

In our tests, we found that collecting column statistics over one column of a table with 130 million rows and 35 GB in size takes six minutes. Collecting two single-column statistics on a table with 260 million rows and 66 GB in size takes eight to nine minutes. These times ranges will vary in other environments since these timings are from a non-busy system.

Note: In some cases, the performance of an SQL statement run by SQE the first time on V5R2 is slower than on previous releases. This is because statistics are not automatically collected until a query or SQL request is run. The same query most likely runs faster on subsequent executions since the optimizer has the benefit of the column statistics that were automatically collected.

To avoid this possible first-time degradation in performance after upgrading to V5R2, then proactively collect statistics manually on columns that are frequently searched and that do not have indexes (or keyed logical files) created over them. The searched column or columns must be the leading columns in the index for it to provide proper statistics. If statistics only from indexes are available, then it is possible that SQE generates an access plan different from CQE using the same index statistics.

As stated earlier, column statistics are normally automatically created when Statistics Manager must answer questions from the optimizer using default filter factors. However, there is an *exception*. When an index is available that can be used to generate the answer, then column statistics are not automatically generated. There may be cases where optimization time benefits from column statistics in this scenario because using column statistics to answer questions from the optimizer is more efficient than using the index data. If you have cases where query optimization seems to be extended, check whether you have indexes over the relevant columns in your query. If this is the case, try to manually generate column statistics for these columns.

4.1.4 Automatic statistics refresh

Column statistics are not maintained when the underlying table data changes. This means that there must be some mechanism for Statistics Manager to determine whether the column statistics are still meaningful and when they need to be refreshed. This validation is done each time the following actions occur:

- An Open Data Path (ODP) creation occurs for a query where column statistics were used to create the access plan.
- A new plan is added to the plan cache, either because a completely new query was optimized or because a plan was re-optimized.

At that point, Statistics Manager checks whether either of the following situations applies:

- The number of rows in the table has increased or decreased by more than 15%.
- The number of rows changed in the table is more than 15% of the total table row count.

Statistics Manager then uses the stale column statistics to answer the question from the optimizer. However, it also marks the column statistics as stale in the plan cache and generates a request to refresh the stale column statistics.

Performing an ALTER TABLE and changing the attributes of a column that has column statistics available, such as changing the length of a character field from 25 to 10, also automatically starts a refresh of the column statistics.

4.1.5 Indexes versus column statistics

A major difference between indexes and column statistics is that *indexes* are permanently updated when changes to the underlying table occur. *Column statistics* are not directly updated in this case. Consider an environment where a specific index is needed only for getting statistical information. Then compare the use of the index to the use of column statistics. In this scenario, there are two situations if the data changes constantly. One is that the optimizer might rely on stale column statistics frequently. The other is that maintaining the index after each change to the table might take up more system resources than refreshing the stale column statistics after a group of changes to the table has occurred.

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

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

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

Another important difference is that column statistics can be used *only* for query optimization. They *cannot* be used for the actual implementation of a query, where indexes can be used for both.

Indexes and statistics working together

Let us illustrate how the optimizer uses statistics and indexes together in the execution of a query. Assume that a user executes a query (Q1), with a grouping clause (GROUP BY) on Customer_No. Also assume that there are no indexes and no statistics on Customer_No. We illustrate this example in Figure 4-6.

On the first execution of the query, since there are no indexes and no column statistics by Customer_No, the query (Q1) is optimized with the default values. It also submits a request to create a column statistics by Customer_No.

On the second execution of the query, since there is a good chance that the column statistics have already been created, the query (Q1) is optimized using the column statistic. There is also a good chance that the optimizer has given feedback advising the creation of an index on the table by Customer_No.

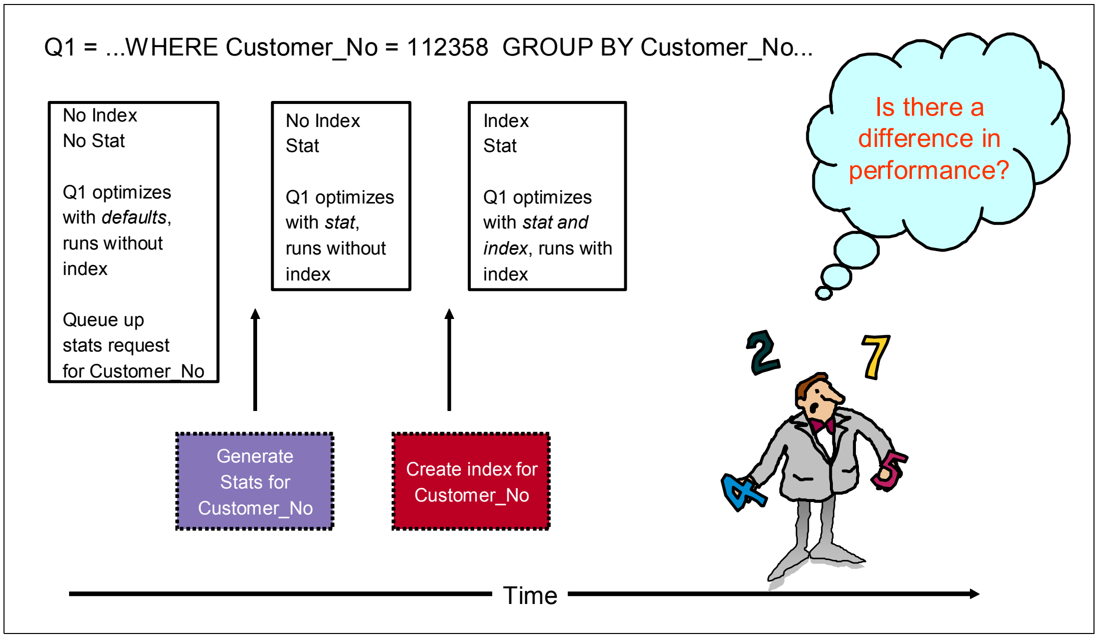

Q1 = ...WHERE Customer_No = 112358 GROUP BY Customer_No...
No Index
No Stat
Q1 optimizes with *defaults*, runs without index
Queue up stats request for Customer_No
No Index
Stat
Q1 optimizes with *stat*, runs without index
Index
Stat
Q1 optimizes with *stat and index*, runs with index
Is there a difference in performance?
Generate Stats for Customer_No
Create index for Customer_No
2
7
4
5
Time

Figure 4-6 Example of statistics and indexes working together

Let us assume that the database analyst decides to create an index by Customer_No. On the third run of the query (Q1), the query is optimized with the statistic and the index, and the optimizer may decide to use the index in the implementation of the query.

The example presented in this section illustrates how both concepts complement each other.

4.1.6 Monitoring background statistics collections

System value QDBFSTCCOL controls who is allowed to create statistics in the background. To display the value of QDBFSTCCOL, on the command line, type:

```
DSPSYSVAL SYSVAL(QDBFSTCCOL)
```

The following list provides the possible values:

- ***ALL**: Allows all statistics to be collected in the background. This is the default setting.
- ***NONE**: Restricts everyone from creating statistics in the background. This does *not* prevent immediate user-requested statistics from being collected.
- ***USER**: Allows only user-requested statistics to be collected in the background.
- ***SYSTEM**: Allows only system-requested statistics to be collected in the background.

> **Note:** User-initiated pending background requests are visible, for example, through the iSeries Navigator graphical user interface (GUI) (Viewing Statistics Collection Requests) and can be removed from that queue via the same interface.

When you switch the system value to something other than *ALL or *SYSTEM, it does not mean that Statistics Manager does not place information about the column statistics it wanted into the plan cache. It simply means that these column statistics are not gathered. When switching the system value back to *ALL, for example, this initiates some background processing that analyzes the entire plan cache and looks for any column statistics requests that are there. This background task also checks which column statistics are used by any plan in the plan cache. It checks whether these column statistics have become stale. Requests for new column statistics and for refresh of column statistics are then executed.

For a user to manually request column statistics collection, the user must have *OBJOPR and *OBJALTER authority for the underlying table.

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

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

- Priority 90 is for short user requests.
- Priority 93 is for long user requests.
- Priority 96 is for short system requests.
- Priority 99 is for long system requests.

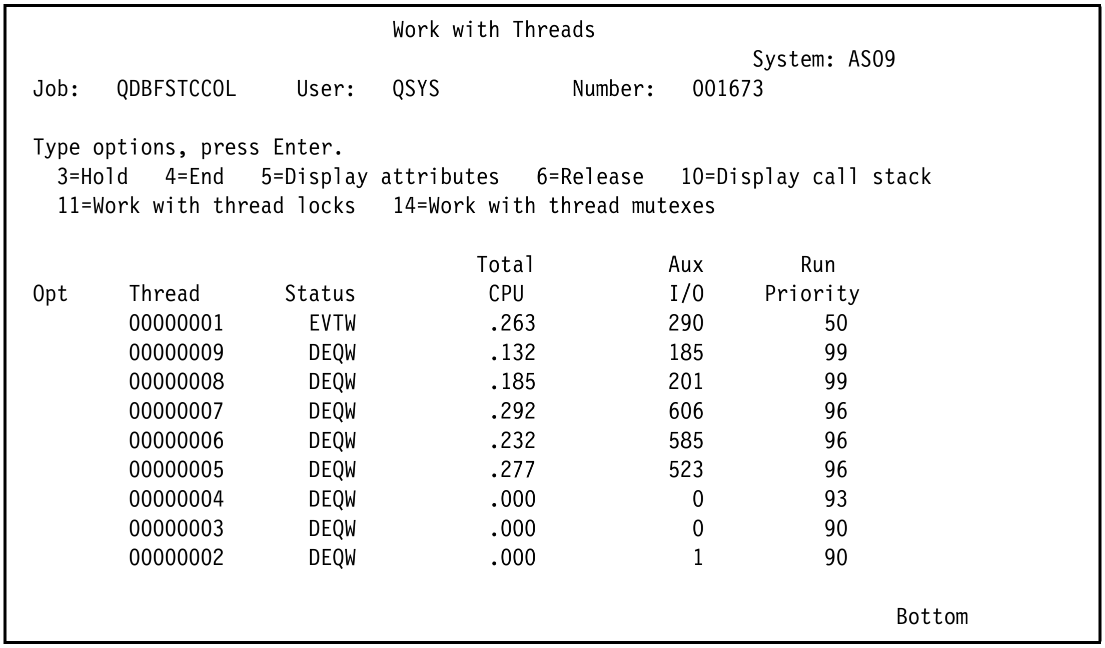
Background versus immediate statistics collection: *Background* statistics collections attempt to use as much parallelism as possible. This parallelism is independent of the symmetric multiprocessing (SMP) feature installed on the System i platform. However, parallel processing is allowed only for *immediate* statistics collection if SMP is installed on the system and the job requesting the column statistics is set to allow parallelism.

To view the number of threads that are set up, follow these steps:

1. Type the following command:

```
WRKJOB JOB(QDBFSTCCOL)
```

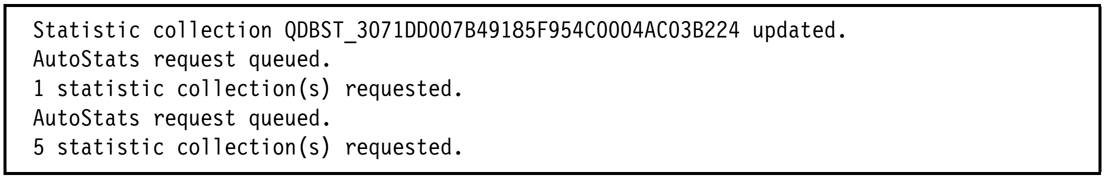
2. Select menu option `20` to work with threads. Figure 4-7 shows the Work with Threads display that opens.



```
                                Work with Threads
                                                              System: AS09
 Job:   QDBFSTCCOL     User:   QSYS          Number:   001673

 Type options, press Enter.
   3=Hold   4=End   5=Display attributes   6=Release   10=Display call stack
   11=Work with thread locks   14=Work with thread mutexes

                                       Total          Aux        Run
 Opt     Thread       Status           CPU            I/O     Priority
         00000001       EVTW           .263           290          50
         00000009       DEQW           .132           185          99
         00000008       DEQW           .185           201          99
         00000007       DEQW           .292           606          96
         00000006       DEQW           .232           585          96
         00000005       DEQW           .277           523          96
         00000004       DEQW           .000             0          93
         00000003       DEQW           .000             0          90
         00000002       DEQW           .000             1          90

                                                                     Bottom
```

Figure 4-7 Threads for the QDBFSTCCOL job

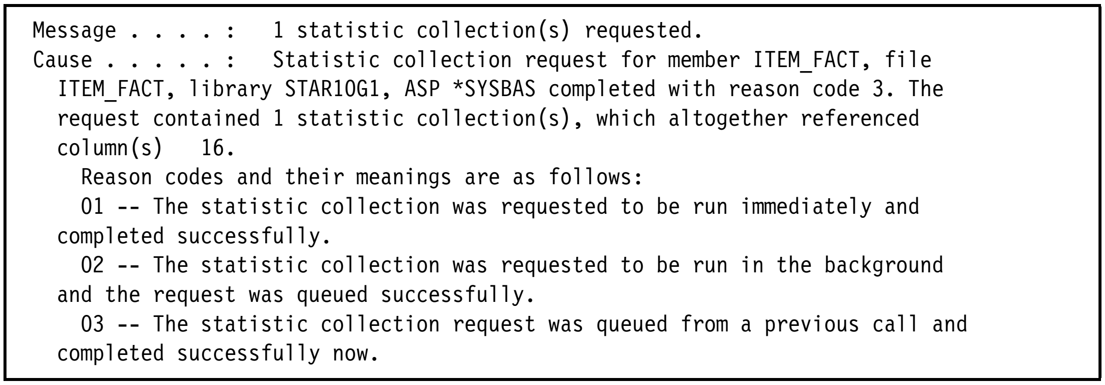
The job log of the QDBFSTCCOL job contains information about which columns statistics were requested, queued, or updated. Figure 4-8 shows some examples of the information that can be found in the job log.



```
Statistic collection QDBST_3071DD007B49185F954C0004AC03B224 updated.
AutoStats request queued.
1 statistic collection(s) requested.
AutoStats request queued.
5 statistic collection(s) requested.
```

Figure 4-8 Information found in the QDBFSTCCOL job log

The detailed messages provide further information about the reason for the column statistics request and the table for which the column statistics were collected. Figure 4-9 shows an example of this.



```
Message . . . . :   1 statistic collection(s) requested.
Cause . . . . . :   Statistic collection request for member ITEM_FACT, file
  ITEM_FACT, library STAR10G1, ASP *SYSBAS completed with reason code 3. The
  request contained 1 statistic collection(s), which altogether referenced
  column(s)   16.
    Reason codes and their meanings are as follows:
    01 -- The statistic collection was requested to be run immediately and
  completed successfully.
    02 -- The statistic collection was requested to be run in the background
  and the request was queued successfully.
    03 -- The statistic collection request was queued from a previous call and
  completed successfully now.
```

Figure 4-9 Detailed message from the QDBFSTCCOL job log

4.1.7 Manual collection and refresh of column statistics

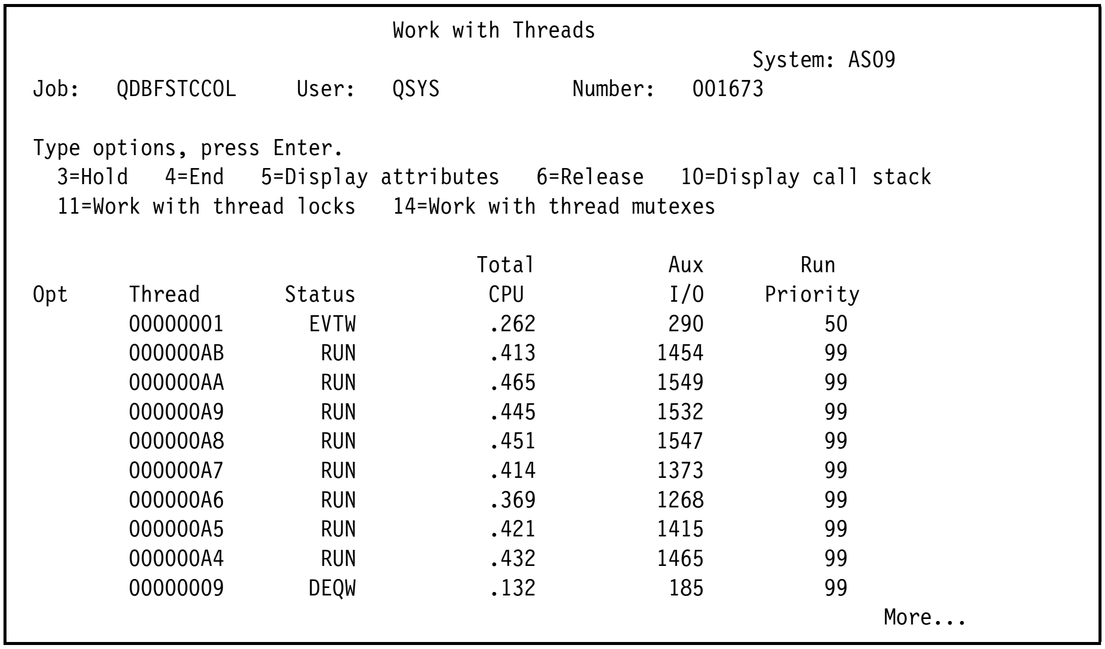
As stated earlier, the process of collecting and refreshing column statistics is, by default, a completely automated process that does not require user intervention. However, it is possible to manually collect column statistics and refresh column statistics.

You can do this by using iSeries Navigator, as explained in 4.2, “Using iSeries Navigator to manage statistics” on page 75, or by using the Statistics Manager application program interfaces (APIs) as discussed in Appendix A, “Statistics Manager API code examples” on page 155. Refer to 4.3, “Proactive statistics collection” on page 82, for when you want to manually collect column statistics.

4.1.8 Statistics collection performance considerations

The system background statistics collection is designed to minimize the impact on other user jobs. This is mainly accomplished by running these collection jobs at a low priority. However, system resource utilization, such as CPU usage, can increase. That is because the system statistics collection uses the remaining available resources to accomplish its tasks.

Figure 4-10 shows how a currently active collection of column statistics looks in Work with Active Jobs (WRKACTJOB). Look at the number of active threads with priority 99 that are working on generating system requested column statistics. Do not become confused by the first thread that shows a priority of 50. The real work for collecting column statistics is done in all the other threads that show priority 99.



```
                              Work with Threads
                                                              System:   AS09
Job:   QDBFSTCCOL     User:   QSYS          Number:   001673

Type options, press Enter.
  3=Hold   4=End   5=Display attributes   6=Release   10=Display call stack
  11=Work with thread locks   14=Work with thread mutexes

                                    Total          Aux        Run
Opt     Thread       Status           CPU          I/O     Priority
        00000001       EVTW          .262          290          50
        000000AB        RUN          .413         1454          99
        000000AA        RUN          .465         1549          99
        000000A9        RUN          .445         1532          99
        000000A8        RUN          .451         1547          99
        000000A7        RUN          .414         1373          99
        000000A6        RUN          .369         1268          99
        000000A5        RUN          .421         1415          99
        000000A4        RUN          .432         1465          99
        00000009       DEQW          .132          185          99
                                                                    More...
```

Figure 4-10 Active collection of column statistics in WRKACTJOB

To reduce the impact of automatic statistics collection on any other workload on the system even more, two additional concepts are used:

- The introduction of low priority I/O
- Work management changes for the QDBFSTCCOL job

Low priority I/O is implemented in the code of the driver for the hard disk drive. This code makes sure that only a small percentage of the I/O resources can be used by statistics collection at any given point in time.

Regarding the CPU consumption of the QDBFSTCCOL, you are familiar with the concept that a job starts using the CPU. The job stays there until it either goes into wait status (because of waiting for an I/O activity to finish, for example) or reaches its time slice end. The QDBFSTCCOL job monitors the system for other jobs with higher priority that are waiting for CPU resources. If this happens, it goes into wait status although it is not waiting for I/O and has not reached its time slice end.

Note: If you schedule a low priority job that is permanently active on your system and that is supposed to use all spare CPU cycles for processing, then this prevents an automatic statistics collection from becoming active because of the previously mentioned concept.

4.1.9 Propagation of column statistics on copy operations

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

Statistics are copied when using the Create Duplicate Object (CRTDUPOBJ) command with DATA(*YES). Using this command to create a copy of a file may be an alternative to manually creating column statistics after using a CPYF command.

Since column statistics are part of the table, they are saved at the time that a save operation to the table occurs. The same applies when we restore a table. The column statistics that were saved with the table are restored with the restore operation.

4.2 Using iSeries Navigator to manage statistics

iSeries Navigator provides a GUI to the statistics information through the use of statistic APIs. The following functionality was added to iSeries Navigator in iSeries Access for Windows® V5R2:

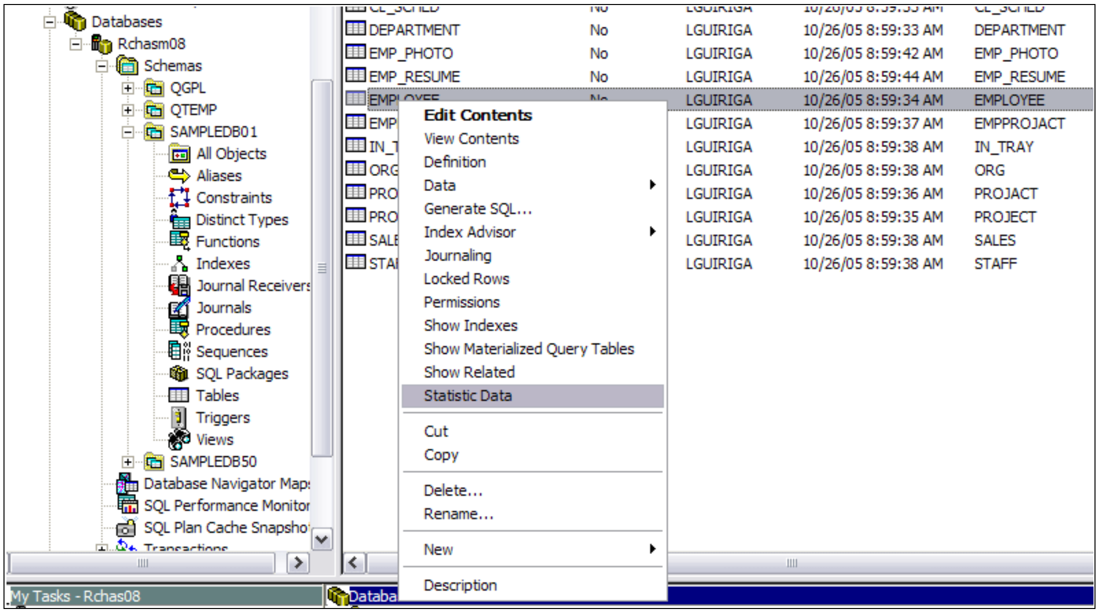
- View statistics
- Create statistics
- Delete statistics
- Refresh statistics
- Block statistics from being created for a table
- View statistics collection requests

Previous versions of iSeries Access for Windows do not contain any statistics collection functions.

4.2.1 Viewing statistics

To view statistics for a table:

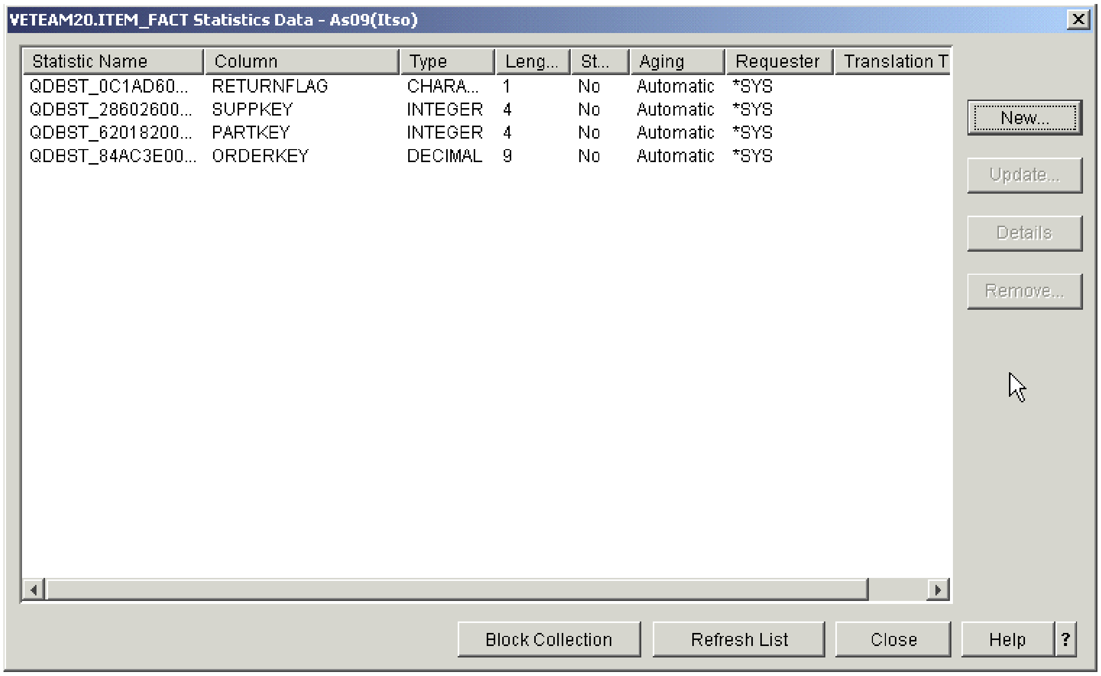
1. Open iSeries Navigator.
2. Select the library in which the table is located. A list of all the tables and SQL aliases in the library appears in the right panel of the window as shown in the example in Figure 4-11. Right-click the desired table name and select **Statistic Data**.



DEPARTMENT	No	LGUIRIGA	10/26/05 8:59:33 AM	DEPARTMENT
EMP_PHOTO	No	LGUIRIGA	10/26/05 8:59:42 AM	EMP_PHOTO
EMP_RESUME	No	LGUIRIGA	10/26/05 8:59:44 AM	EMP_RESUME
EMPLOYEE	No	LGUIRIGA	10/26/05 8:59:34 AM	EMPLOYEE
EMP		LGUIRIGA	10/26/05 8:59:37 AM	EMPPROJACT
IN_T		LGUIRIGA	10/26/05 8:59:38 AM	IN_TRAY
ORG		LGUIRIGA	10/26/05 8:59:38 AM	ORG
PRO		LGUIRIGA	10/26/05 8:59:36 AM	PROJACT
PRO		LGUIRIGA	10/26/05 8:59:35 AM	PROJECT
SALE		LGUIRIGA	10/26/05 8:59:38 AM	SALES
STAI		LGUIRIGA	10/26/05 8:59:38 AM	STAFF

Figure 4-11 Viewing statistics

3. The statistics viewer window (Figure 4-12) opens. This window shows the basic information for each column for which statistics are created. It also provides buttons for the maintenance of the column statistics. Select the column statistics in which you are interested and click the **Details** button.


VETEAM20.ITEM_FACT Statistics Data - As09(Itso)

Statistic Name	Column	Type	Leng...	St...	Aging	Requester	Translation T
QDBST_0C1AD60...	RETURNFLAG	CHARA...	1	No	Automatic	*SYS	
QDBST_28602600...	SUPPKEY	INTEGER	4	No	Automatic	*SYS	
QDBST_62018200...	PARTKEY	INTEGER	4	No	Automatic	*SYS	
QDBST_84AC3E00...	ORDERKEY	DECIMAL	9	No	Automatic	*SYS	

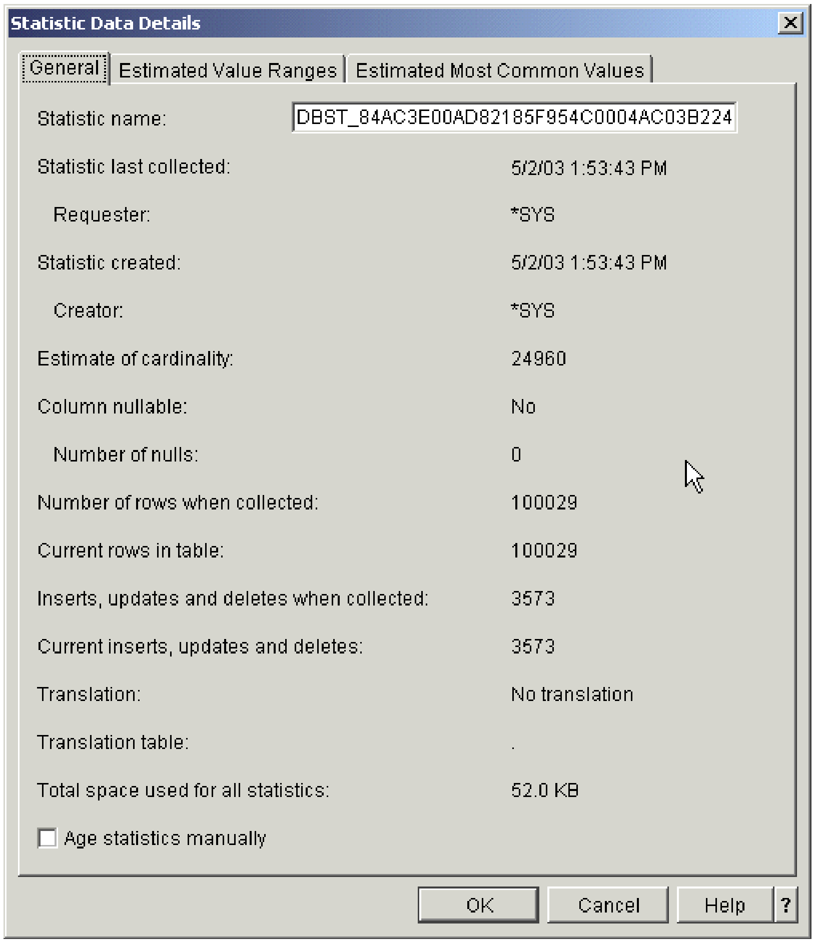
Figure 4-12 Statistics viewer window

Then you see the Statistic Data Details window (Figure 4-13) with detailed statistical information. The General tab provides the following information:

- **Statistics name**: The name of the statistics can be changed.
- **Timestamps**: These are for the last request and initial creation, as well as the user that requested each of them.
- **Statistics data**: This data shows the estimate for cardinality and the number of nulls.
- **Record counts and changed record counts**: This is the primary factor when determining a statistics staleness.

  If there is a significant change in the number of rows in the table or a significant increase in the number of inserts, updates, and deletes, then the stale indicator is set to “Yes” in the statistics viewer window (Figure 4-12).
- **Translation table** (if used)
- **Total space used for all statistics** (collected for this table)
- **Refresh option**: The Age statistics manually check box at the bottom of the window allows for changing between manual and system refresh options.

The second and third tabs provide a view of the estimated value ranges and the frequent values list. See 4.1.2, “Single-column statistics” on page 65, for more information and an example of each statistic type.



Statistic name: DBST_84AC3E00AD82185F954C0004AC03B224

Statistic last collected:	5/2/03 1:53:43 PM
Requester:	*SYS
Statistic created:	5/2/03 1:53:43 PM
Creator:	*SYS
Estimate of cardinality:	24960
Column nullable:	No
Number of nulls:	0
Number of rows when collected:	100029
Current rows in table:	100029
Inserts, updates and deletes when collected:	3573
Current inserts, updates and deletes:	3573
Translation:	No translation
Translation table:	.
Total space used for all statistics:	52.0 KB

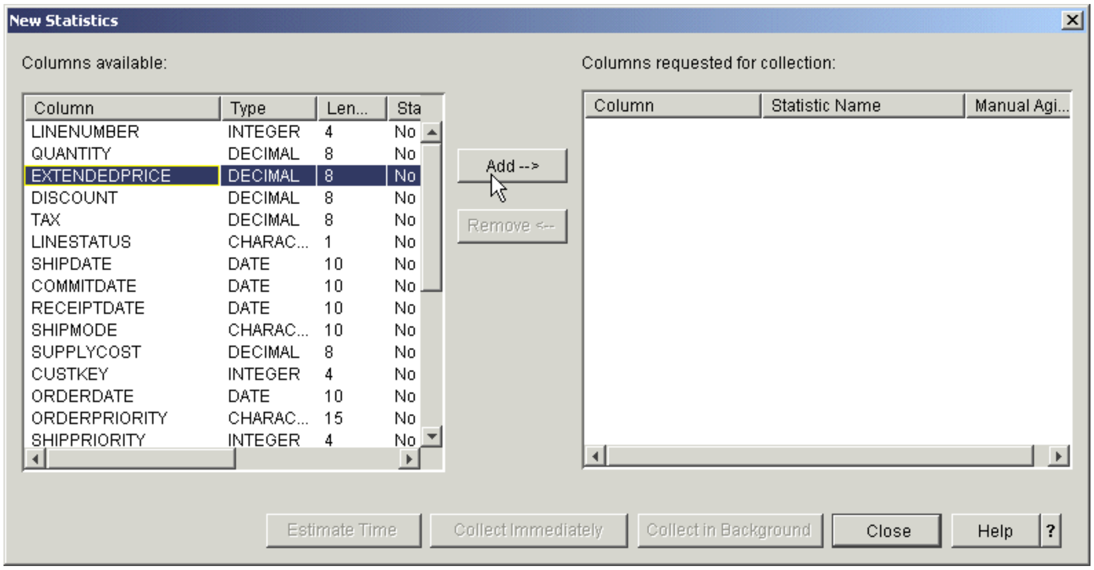
□ Age statistics manually

Figure 4-13 Statistic Data Details: General page

4.2.2 Creating new statistics

To create new column statistics:

1. Go to the statistics viewer window (see 4.2.1, “Viewing statistics” on page 75). Click the **New** button on the statistics viewer window (Figure 4-12).
2. The New Statistics window (Figure 4-14) that opens lists the columns that are available for statistics. The scrollable panel on the left contains all of the columns that do not currently have column statistics available.
   a. To select the column for a possible statistics collection, select the column name and click the **Add** button.

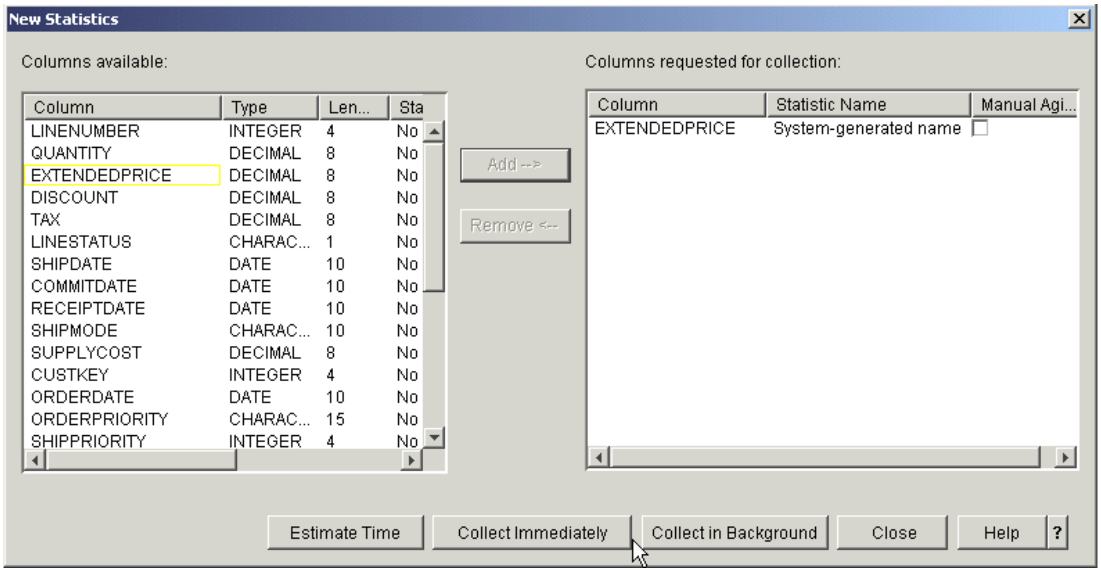


Column	Type	Len...	Sta
LINENUMBER	INTEGER	4	No
QUANTITY	DECIMAL	8	No
EXTENDEDPRICE	DECIMAL	8	No
DISCOUNT	DECIMAL	8	No
TAX	DECIMAL	8	No
LINESTATUS	CHARAC...	1	No
SHIPDATE	DATE	10	No
COMMITDATE	DATE	10	No
RECEIPTDATE	DATE	10	No
SHIPMODE	CHARAC...	10	No
SUPPLYCOST	DECIMAL	8	No
CUSTKEY	INTEGER	4	No
ORDERDATE	DATE	10	No
ORDERPRIORITY	CHARAC...	15	No
SHIPPRIORITY	INTEGER	4	No

Column	Statistic Name	Manual Agi...

Figure 4-14 New Statistics window

After you select a column, the right pane shows the column that you selected, the name of the column statistics, and the refresh option as shown in the example in Figure 4-15. You can specify a unique own name for the column statistics for maintenance purposes. You can also change the name of the column statistics after adding it by clicking the system generated name that appears in the window. Selecting the Age statistics manually check box sets the column statistics to manual refresh only. If you do not select the Age statistics manually check box, the column statistics are automatically refreshed by the system.



Column	Type	Len...	Sta
LINENUMBER	INTEGER	4	No
QUANTITY	DECIMAL	8	No
EXTENDEDPRICE	DECIMAL	8	No
DISCOUNT	DECIMAL	8	No
TAX	DECIMAL	8	No
LINESTATUS	CHARAC...	1	No
SHIPDATE	DATE	10	No
COMMITDATE	DATE	10	No
RECEIPTDATE	DATE	10	No
SHIPMODE	CHARAC...	10	No
SUPPLYCOST	DECIMAL	8	No
CUSTKEY	INTEGER	4	No
ORDERDATE	DATE	10	No
ORDERPRIORITY	CHARAC...	15	No
SHIPPRIORITY	INTEGER	4	No

Column	Statistic Name	Manual Agi...
EXTENDEDPRICE	System-generated name	☐

Figure 4-15 New Statistics window with a column selected

b. After you select all of the columns for which you want statistics collected, click one of the buttons at the bottom of the window (Figure 4-15):

  - **Estimate Time**: Provides an estimate of how long all of the selected statistics collections will take
  - **Collect Immediately**: Starts the collection of the statistics and locks the window until the statistics collection finishes
  - **Collect Background**: Places the requests on one of the user background request queues

    The statistics collection starts when system resources are available and the system value QDBFSTCCOL allows it.

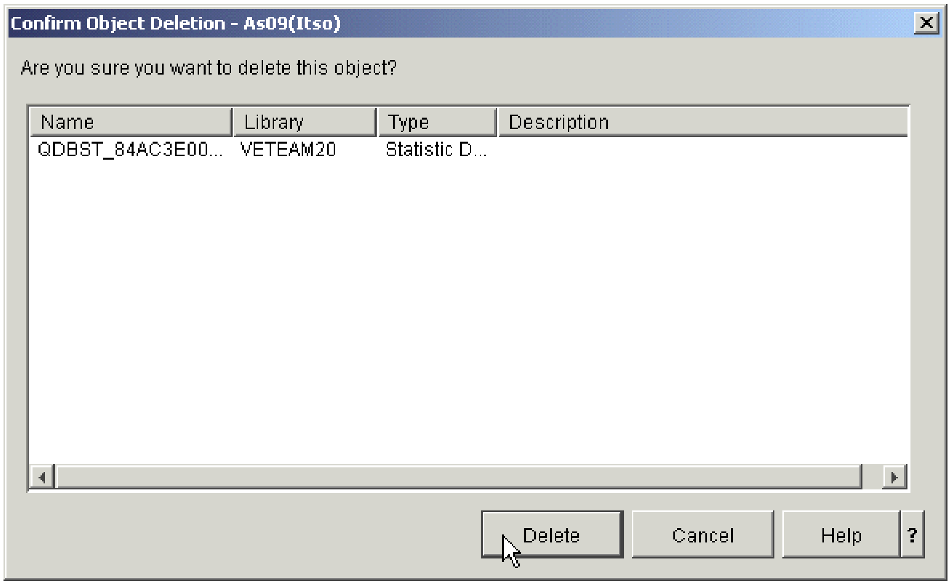
  - **Close button**: Closes the window and disregards any columns that might have been selected

The statistics collection starts when system resources are available and the system value allows it.

4.2.3 Deleting a statistic

To delete a statistic:

1. Go to the statistics viewer window (see 4.2.1, “Viewing statistics” on page 75). In the statistics viewer window (Figure 4-12 on page 76), select one or more column statistics and then click the **Remove** button.
2. In the Confirm Object Deletion window (Figure 4-16), you can either click the **Delete** button to delete the statistics or click the **Cancel** button to return to the statistics viewer window without deleting the statistics.



Are you sure you want to delete this object?

Name	Library	Type	Description
QDBST_84AC3E00...	VETEAM20	Statistic D...	

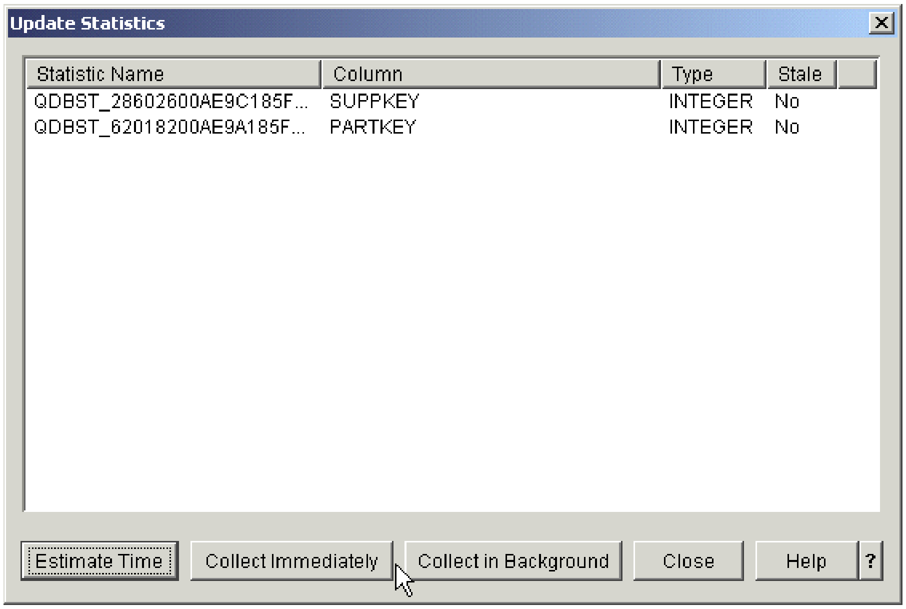
Figure 4-16 Deleting a statistics collection

4.2.4 Refreshing a statistic

To refresh a statistic:

1. Go to the statistics viewer window (see 4.2.1, “Viewing statistics” on page 75). In the statistics viewer window (Figure 4-12 on page 76), select one or more statistics and then click the **Update** button.
2. In the Update Statistics window (Figure 4-17), you can choose to click any of the following buttons at the bottom of the Update Statistics window:
   – **Estimate Time**: Provides an estimate of how long the statistics refresh will take
   – **Collect Immediately**: Starts the collection of the statistics and locks the window until the statistics collection finishes
   – **Collect Background**: Places the requests on one of the user background request queues

     The statistics collection starts when system resources are available and the system value allows it.
   – **Close**: Closes the window and disregards any columns that might have been selected



Statistic Name	Column	Type	Stale
QDBST_28602600AE9C185F...	SUPPKEY	INTEGER	No
QDBST_62018200AE9A185F...	PARTKEY	INTEGER	No

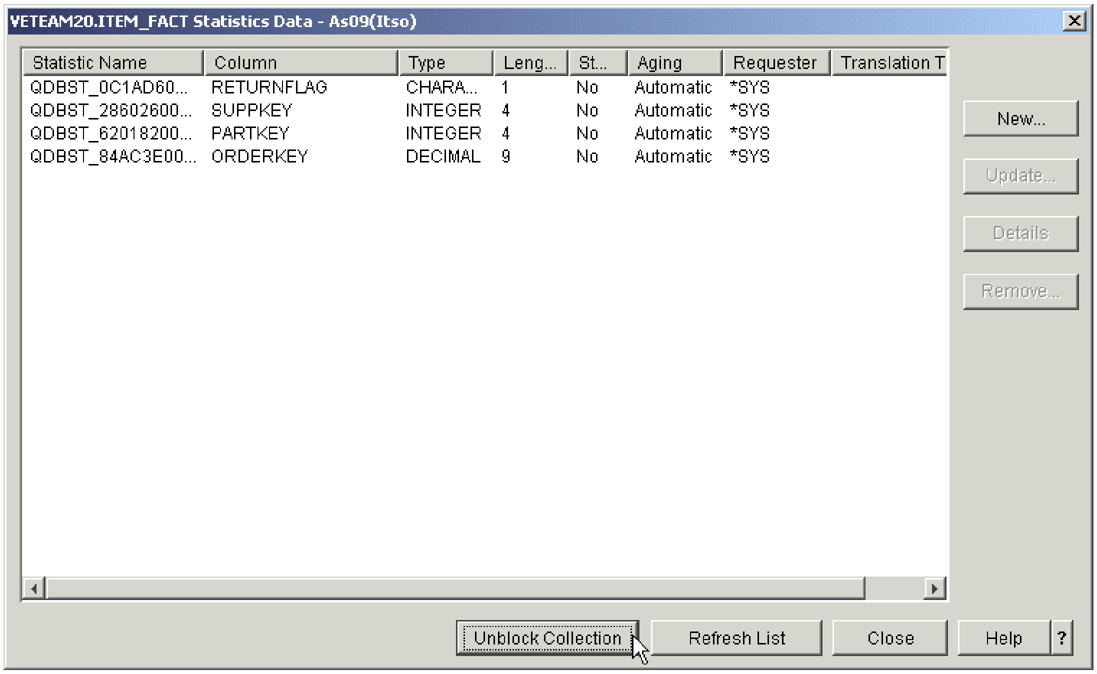
Figure 4-17 Refreshing a statistics collection

4.2.5 Blocking a statistics collection

Blocking a collection disables the system from automatically collecting and refreshing column statistics on the table. Manual statistics collections are still allowed for the table.

To block statistics collections:

1. Go to the statistics viewer window (see 4.2.1, “Viewing statistics” on page 75). In the statistics viewer window (Figure 4-12 on page 76), click the **Block Collection** button.
2. The button then changes to Unblock collection as shown in Figure 4-18. You can click the **Unblock Collection** button to enable automatic statistics collections again for the table.

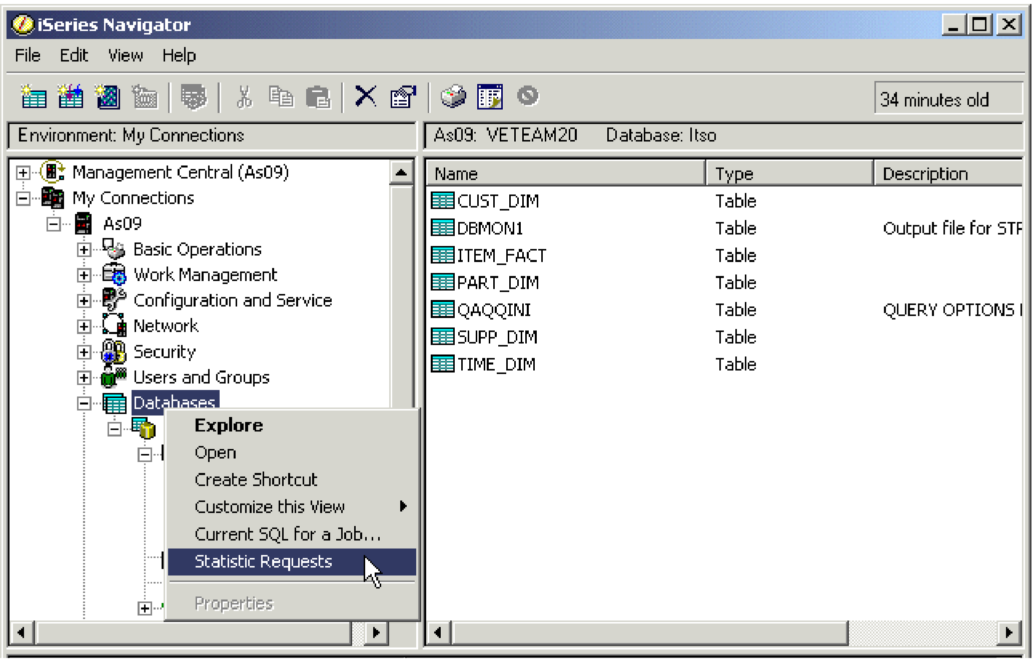


Statistic Name	Column	Type	Leng...	St...	Aging	Requester	Translation T
QDBST_0C1AD60...	RETURNFLAG	CHARA...	1	No	Automatic	*SYS	
QDBST_28602600...	SUPPKEY	INTEGER	4	No	Automatic	*SYS	
QDBST_62018200...	PARTKEY	INTEGER	4	No	Automatic	*SYS	
QDBST_84AC3E00...	ORDERKEY	DECIMAL	9	No	Automatic	*SYS	

Figure 4-18 Blocking automatic statistics collections

4.2.6 Viewing statistics collection requests

iSeries Navigator allows users to view pending user-requested statistics collections that are active. To view these requests, right-click **Database** in the left pane of the iSeries Navigator window and select **Statistic Requests** as shown in Figure 4-19.



Name	Type	Description
CUST_DIM	Table	
DBMON1	Table	Output file for STF
ITEM_FACT	Table	
PART_DIM	Table	
QAQQINI	Table	QUERY OPTIONS I
SUPP_DIM	Table	
TIME_DIM	Table	

Figure 4-19 Opening statistics requests view

Then the Statistics Requests window (Figure 4-20) opens. This window shows all user requested statistics collections that are pending or active. It also shows all system requested statistics collections that are active or have failed.

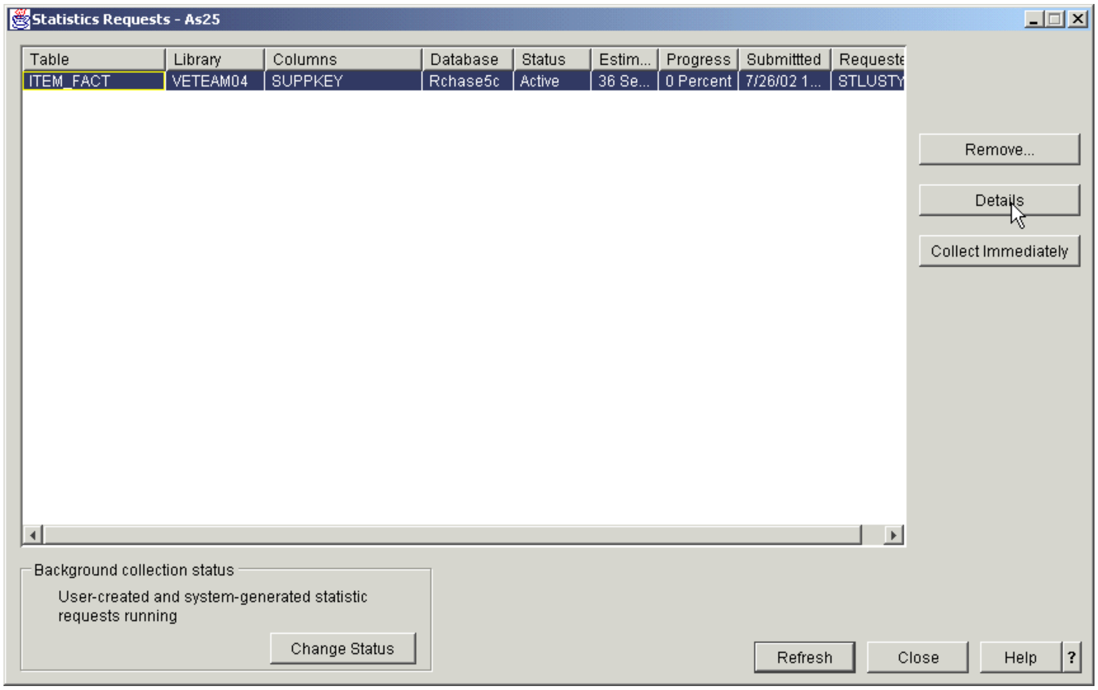


Table	Library	Columns	Database	Status	Estim...	Progress	Submittted	Requeste
ITEM_FACT	VETEAM04	SUPPKEY	Rchase5c	Active	36 Se...	0 Percent	7/26/02 1...	STLUSTY

Background collection status

User-created and system-generated statistic requests running

Figure 4-20 Statistics collections request viewer

4.3 Proactive statistics collection

As stated earlier, statistics collection and refreshing is, by default, a completely automated process on the System i platform that normally does not require any user intervention. However, there are some situations where you might want to control which column statistics are gathered or refreshed and when this maintenance is to occur. In the following sections, we describe these situations and show ways to influence the automatic collection of column statistics.

4.3.1 Determining which column statistics exist

When thinking about proactively managing the collection of column statistics, you may first be interested in knowing which column statistics were automatically collected so far on your system. The answer can be determined by running the SQL statement shown in Example 4-1.

Example 4-1 SQL to view a list of auto-collected statistics

```
SELECT filnam, mbrnam, colnam
      FROM TABLE(dbstcmd.lstdbfstc(
                             asp_id,
                             libname10,
                             table_name,
                             member_name))AS A
      WHERE crtusr='*SYS' ORDER BY 1,2,3;
```

This example assumes that you created the List Statistics Collection user-defined table function (UDTF) shown in 4.4, “Statistics Manager APIs” on page 87, and stored the UDTF as LSTDBFSTC in library DBSTCMD.

When using the UDTF, pad all input parameters with blanks at the end so that they contain 10 characters each. Also be sure to use all uppercase characters for the names. The possible input values are:

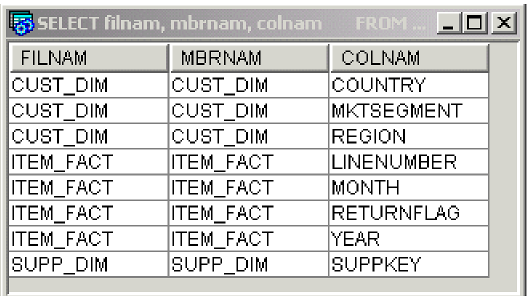
- **For asp_id**: The actual ASP name, special value *SYSBAS or * (meaning current)
- **For library10**: The actual library name, special values *LIBL, *CURLIB or *USRLIBL
- **For table_name**: The actual table name or special value *ALL if an actual library name was provided
- **For member_name**: The actual member name, special value *FIRST, *LAST or *ALL

  For SQL created tables, the value *FIRST should be used.

For example, consider running the following statement with iSeries Navigator:

```
SELECT filnam, mbrnam, colnam
      FROM TABLE(dbstcmd.lstdbfstc('*         ',
                                   'STAR10G   ',
                                   '*ALL      ',
                                   '*ALL      ')) AS A
       WHERE crtusr ='*SYS'  ORDER BY 1,2,3;
```

This statement produces the results shown in Figure 4-21.



FILNAM	MBRNAM	COLNAM
CUST_DIM	CUST_DIM	COUNTRY
CUST_DIM	CUST_DIM	MKTSEGMENT
CUST_DIM	CUST_DIM	REGION
ITEM_FACT	ITEM_FACT	LINENUMBER
ITEM_FACT	ITEM_FACT	MONTH
ITEM_FACT	ITEM_FACT	RETURNFLAG
ITEM_FACT	ITEM_FACT	YEAR
SUPP_DIM	SUPP_DIM	SUPPKEY

Figure 4-21   List of columns with statistics collected

The example code works on a per-library basis. If you want to learn about column statistics for all libraries, you must run this statement multiple times.

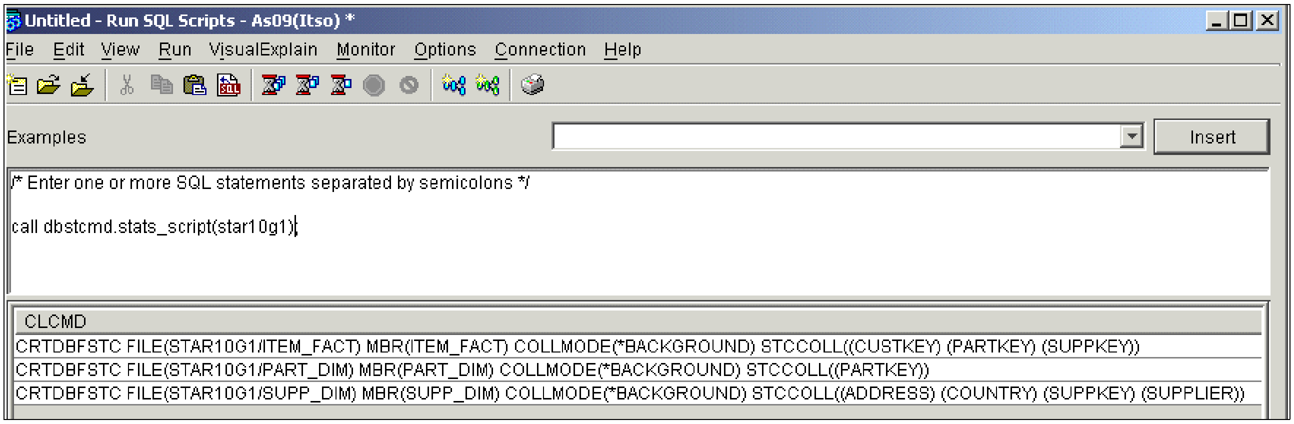
A common follow-on to determine which statistics are automatically collected is to see how you can automate the recreation of those column statistics. The procedure in Example 4-2 creates a database file that contains the commands to recreate these column statistics. Again, this is based on the CL commands in Appendix A, “Statistics Manager API code examples” on page 155.

Example 4-2   Creating a command list file to recreate column statistics

```
/* How to call the SQL-procedure for library SQTEST; must be run inside SQL.
/*          call dbstcmd.stats_script('SQTEST’)

/* SQL script to create procedure ----------------------------

CREATE PROCEDURE dbstcmd.stats_script(IN libname VARCHAR(10))
LANGUAGE SQL
RESULT SETS 1
set option dbgview=*SOURCE
```

```
sp: BEGIN

/* Library name has to be fixed-length 10 character string */
DECLARE libname10 CHAR(10);

/* Search System ASP - SYSBAS */
DECLARE asp_id CHAR(10) DEFAULT '*         ';

/* Search all tables & files in the specified library   */
DECLARE table_name CHAR(10) DEFAULT '*ALL      ';

/* Search all members in the specified library   */
DECLARE member_name CHAR(10) DEFAULT '*ALL      ';

DECLARE current_table CHAR(10) DEFAULT NULL;
DECLARE current_mbr CHAR(10);
DECLARE columnlist  CHAR(8000);

DECLARE temp_table_exists CONDITION FOR SQLSTATE '42710';

/* Result set cursor */
DECLARE statcmds CURSOR WITH RETURN FOR SELECT * FROM dbstcmd.cmdtable;

DECLARE CONTINUE HANDLER FOR temp_table_exists
  SET current_table=NULL;

CREATE TABLE dbstcmd.cmdtable
  (clcmd VARCHAR(10000));

SET libname10 = UPPER(libname);

FOR loopvar AS
  collist CURSOR FOR
  SELECT filnam, mbrnam, colnam
      FROM TABLE(dbstcmd.lstdbfstc(
                                   asp_id,
                                   libname10,
                                   table_name,
                                   member_name))AS A
      WHERE crtusr='*SYS' ORDER BY 1,2,3
  DO
   IF current_table IS NULL THEN
       SET current_table = filnam;
       SET current_mbr = mbrnam;
       SET columnlist = '('||strip(colnam)||')';
    ELSEIF (current_table=filnam AND current_mbr=mbrnam)
      THEN
         SET columnlist = strip(columnlist) || ' ('||strip(colnam)||')';
      ELSE
        INSERT INTO dbstcmd.cmdtable VALUES(
          'CRTDBFSTC FILE(' || strip(libname10) || '/' ||
            strip(current_table) || ') MBR(' || strip(current_mbr) ||
            ') COLLMODE(*BACKGROUND) STCCOLL(' || strip(columnlist) || ')'
          );
         SET current_table = filnam;
         SET current_mbr = mbrnam;
         SET columnlist = '('||strip(colnam)||')';
      END IF;
  END FOR;

  /* Generate cmd string for last table the FOR loop was working on */
  IF current_table IS NOT NULL THEN
    INSERT INTO dbstcmd.cmdtable VALUES(
         'CRTDBFSTC FILE(' || strip(libname10) || '/' ||
           strip(current_table) || ') MBR(' || strip(current_mbr) ||
           ') COLLMODE(*BACKGROUND) STCCOLL(' || strip(columnlist) || ')'
          );


  END IF;

  /* Pass back result to invoker */
  OPEN statcmds;

END;
```

Figure 4-22 shows the example output when running the procedure.


Untitled - Run SQL Scripts - As09(Itso) *
File Edit View Run VisualExplain Monitor Options Connection Help
Examples
Insert

```
/* Enter one or more SQL statements separated by semicolons */

call dbstcmd.stats_script(star10g1);
```

```
CLCMD
CRTDBFSTC FILE(STAR10G1/ITEM_FACT) MBR(ITEM_FACT) COLLMODE(*BACKGROUND) STCCOLL((CUSTKEY) (PARTKEY) (SUPPKEY))
CRTDBFSTC FILE(STAR10G1/PART_DIM) MBR(PART_DIM) COLLMODE(*BACKGROUND) STCCOLL((PARTKEY))
CRTDBFSTC FILE(STAR10G1/SUPP_DIM) MBR(SUPP_DIM) COLLMODE(*BACKGROUND) STCCOLL((ADDRESS) (COUNTRY) (SUPPKEY) (SUPPLIER))
```

Figure 4-22 Output from the stats_script procedure

4.3.2 Reasons for proactive collection of column statistics

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

High availability solutions

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

You can accomplish this by using the SQL procedure shown in 4.3.1, “Determining which column statistics exist” on page 82. You must create the CL commands that are described in Appendix A, “Statistics Manager API code examples” on page 155, on your production system and on your backup system. Then you can run the procedure for each table for which you want to analyze or propagate the column statistics. The command list file generated by this example can then be replicated to the backup system and run there to recreate column statistics.

Notice that the sample code only shows the column statistics that were generated by the system. If you want to include user-requested column statistics, you must change the where clause selection of the sample program accordingly.

Independent software vendor preparation

An independent software vendor (ISV) may want to deliver a solution to a customer that already includes column statistics that are frequently used in the application instead of waiting for the automatic statistics collection to create them. A way to accomplish this is to run the application on the development system for some time and examine which column statistics were created automatically. You can then generate a script file to be shipped as part of the application that should be executed on the customer system after the initial data load took place. Generating this script file can again be accomplished with the code shown in Example 4-2 on page 83.

Business Intelligence environments

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

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

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

However, if you are working with a procedure that needs to delete and recreate the tables, two different options are available. You can either analyze the tables for column statistics that were created using the code shown in Example 4-2 on page 83 *before* you delete them. After you recreate and load the tables, you can use this information to recreate column statistics. Alternatively you can simply create column statistics over all the columns in all of the tables using the statistics APIs described in Appendix A, “Statistics Manager API code examples” on page 155. This may be a good option if the queries that run against your tables are difficult to predict. This may also be good if the time window for your data load is large enough to allow for the extra time needed to create all the column statistics.

Massive data updates

Updating rows in a column statistics-enabled table that significantly change the cardinality, add new ranges of values, or change the distribution of data values can affect the performance for queries when they are *first* run against the new data. This may happen because, on the first run of such a query, the optimizer uses stale column statistics to make decisions on the access plan. At that point, it starts a request to refresh the column statistics.

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

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

Backup and recovery

When thinking about backup and recovery strategies, keep in mind that creation of column statistics is not journaled. Column statistics that exist at the time that a save operation occurs are saved as part of the table and restored with the table. Any column statistics that are created after the save took place are lost and cannot be recreated by using techniques such as applying journal entries. This means that, if you have a rather long interval between save operations and rely heavily on journaling for restoring your environment to a current state, you must consider keeping track of column statistics that are generated after the latest save operation. You can accomplish this again by using the program code shown in Example 4-2 on page 83.

We did some testing regarding the impact that the save activity has on the collection of column statistics. These tests showed that both tasks run independently from each other. This means that a save does not lock a table in a way so that statistics collection is inhibited or vice versa. Statistics collections took longer, however, when the save activity was present. Our tests showed that collecting statistics for one column on a 16 GB table took about four times longer if the table was concurrently saved than without the save taking place.

4.4 Statistics Manager APIs

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

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

For more information about Statistics Manager APIs, refer to the following Information Centers:

- V5R2

  `http://publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm`

- V5R3

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

- V5R4

  `http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp`

For examples of using the Statistics Manager APIs, see Appendix A, “Statistics Manager API code examples” on page 155.

5

Monitoring and tuning SQE

System performance and performance tuning are high priority items on any system administrator’s agenda. The objective is to maximize system resource utilization, while achieving maximum performance throughput.

The previous chapters in this redbook introduce the functionality delivered with Structured Query Language (SQL) Query Engine (SQE) on DB2 for i5/OS starting in OS/400 V5R2. New and added functionality may come at a price if appropriate steps are not taken to tune your particular environment and SQL queries to take full advantage of these features and functions.

In this chapter, we discuss performance considerations that relate to SQL queries and SQE. We begin by examining the tools that are available to monitor the performance of your queries. Then we introduce a few methods that you can employ to tune your SQL queries to achieve increased performance.

For more detailed information about this topic, refer to the existing IBM Redbook *SQL Performance Diagnosis on IBM DB2 Universal Database for iSeries*, SG24-6654.

5.1 Query optimization feedback

Monitoring your queries is the first important step to ensure that your queries are tuned for optimal performance. Consider this monitoring as a type of diagnostic tool for early determination and identification of potential problem areas. By using the information obtained through the monitoring processes, you can take the appropriate corrective actions.

In the following sections, we introduce and discuss the tools and methods that can assist you in monitoring the performance of your SQL queries. Each tool uses the feedback mechanism of the query optimizer (CQE or SQE).

We start by describing the different feedback mechanisms that query optimizer has in OS/400 and i5/OS. For V5R2 and V5R3 of OS/400, the feedback mechanisms are illustrated in Figure 5-1. An SQL request goes through the optimization phase, and the feedback is sent to the different tools such as:

- PRTSQLINF: Provides a way to view information pertaining to SQL packages, service programs, and embedded SQL statements from programs and packages
- Query debug messages
- Visual Explain
- Database monitoring tools, such as the SQL Performance Monitors, also known as the *database monitors*
- Visual Explain
- SQE Plan Cache

Note: In Figure 5-1, the SQE Plan Cache box is represented by a dotted line because, in V5R2 and V5R3, the plan cache is not externalized to the user. There is no way to see the contents of the plan cache.

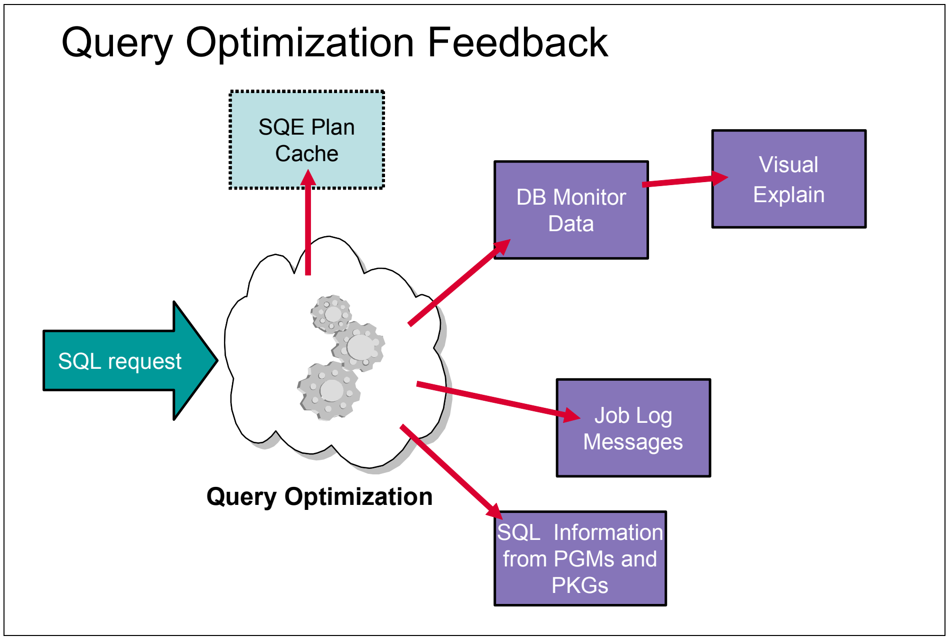


Figure 5-1 Query Optimization feedback mechanisms in V5R2 and V5R3

In V5R4 of i5/OS, additional tools were added to monitor and tune SQL queries. These tools are shown along the top of Figure 5-2 and include:

- System wide Index Advisor or Indexes Advised
- SQE Plan Cache with the capability of visualizing the plan cache with filtering capabilities

  Note that the SQE Plan Cache box is no longer dotted. This is because V5R4 has a visualizer that externalizes the content of the plan cache.
- SQE Plan Cache Snapshots

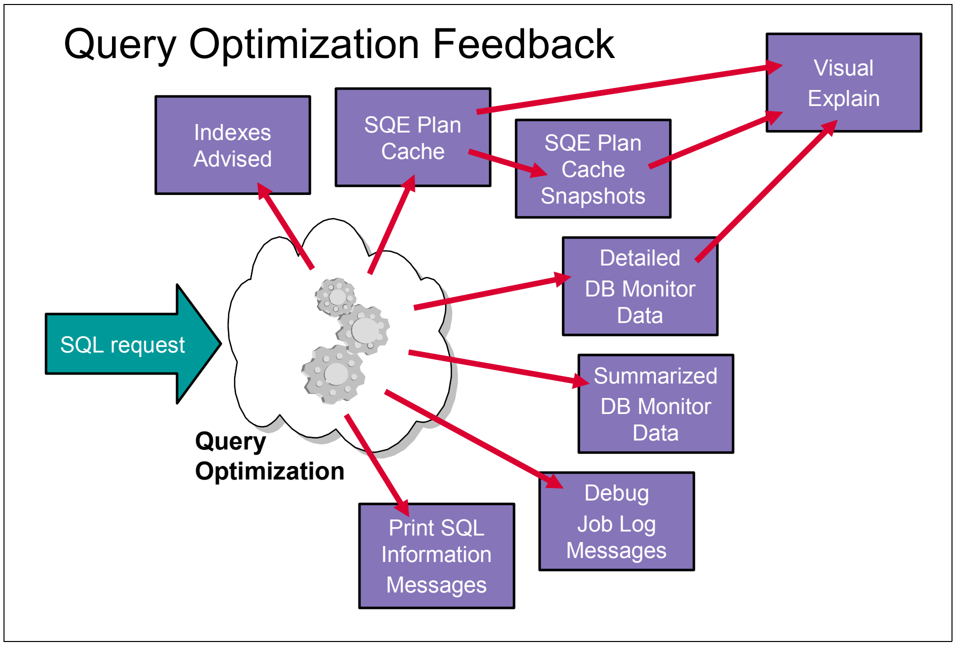


Figure 5-2 Query Optimization feedback mechanisms in V5R4

The best way to learn how to be effective with these different performance analysis tools is to attend the DB2 for i5/OS SQL Performance Workshop. You can find information about the workshop on the Web at:

http://www.ibm.com/servers/eserver/iseries/service/igs/db2performance.html

For some of these tools, you can also download more information from the Educational Resources for Business Partners Web site at the following address:

http://www.ibm.com/servers/enable/site/education/ibo/view.html?oc#db2

In the following sections, we describe each of the tools.

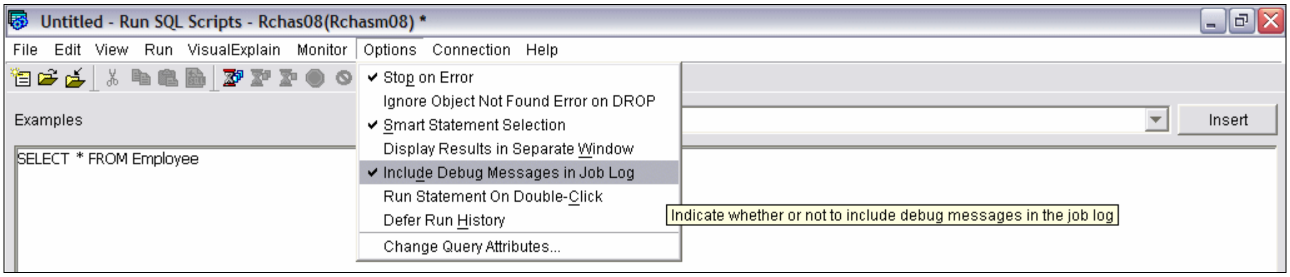
5.1.1 Debug messages

The SQL Query Engine supports the generation of debug messages. However, the debug messages were not enhanced for any of the access methods used by SQE. The SQE optimizer chooses and uses the closest CQE debug message to the access method that it is using. Thus, the debug messages for an SQL request processed by SQE may not be completely accurate. For this reason, Visual Explain and the Database Monitor should be your preferred tools for analyzing SQE Performance.

There are several ways in which you can direct the system to generate debug messages while running your SQL statements, including:

- 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 a particular job to its job log. Using iSeries Navigator, you click **Options** → **Include Debug Messages in Job Log** as shown in Figure 5-3.


Untitled - Run SQL Scripts - Rchas08(Rchasm08) *
File
Edit
View
Run
VisualExplain
Monitor
Options
Connection
Help
✔ Stop on Error
Ignore Object Not Found Error on DROP
✔ Smart Statement Selection
Display Results in Separate Window
✔ Include Debug Messages in Job Log
Run Statement On Double-Click
Defer Run History
Change Query Attributes...
Examples
Insert
Indicate whether or not to include debug messages in the job log

SELECT * FROM Employee

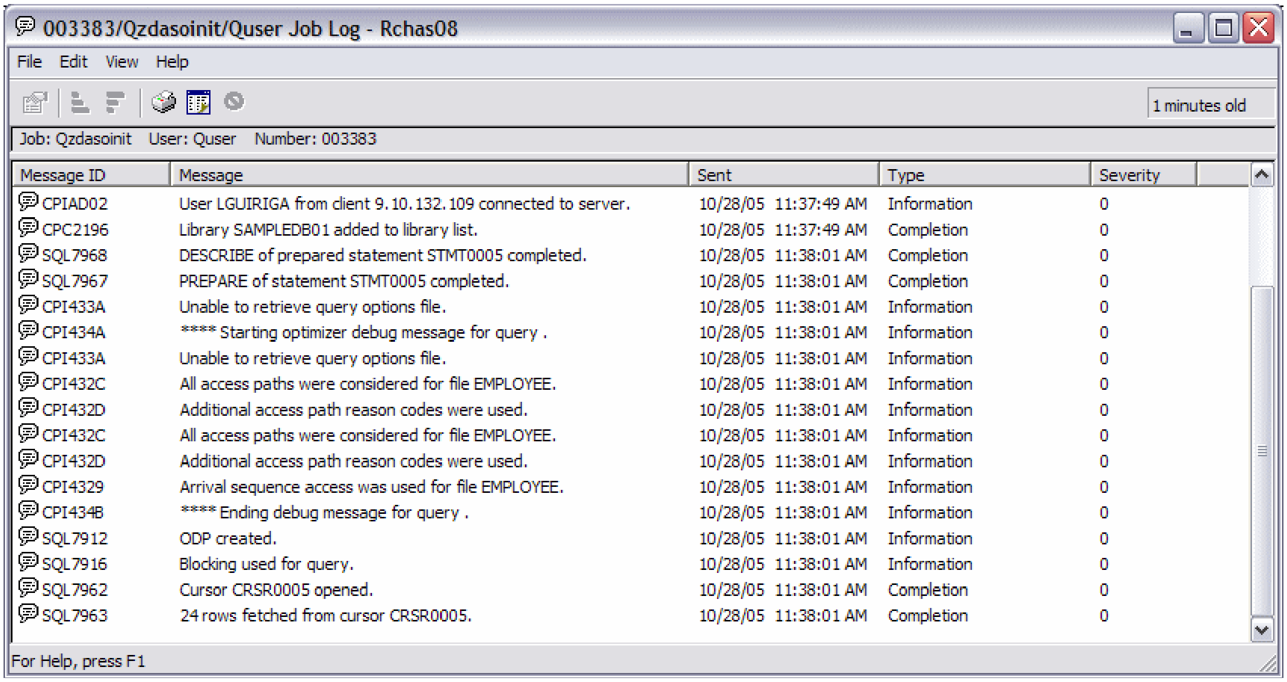
Figure 5-3 Enabling debug messages for a single job

After you run your query, view the job log. Select **View** → **Joblog** on the Run SQL Scripts window. In our example, we used the following SQL statement:

```
SELECT * FROM Employee
WHERE WorkDept BETWEEN 'A01' AND 'E01'
ORDER BY LastName
OPTIMIZE FOR ALL ROWS;
```

The detailed job log provides information that you can use to identify and analyze potential problem areas in your query. Figure 5-4 shows an example of the debug messages that are in the job log after you run the previous query.

After you enable these settings for a particular job, only debug messages that relate to queries that are 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.

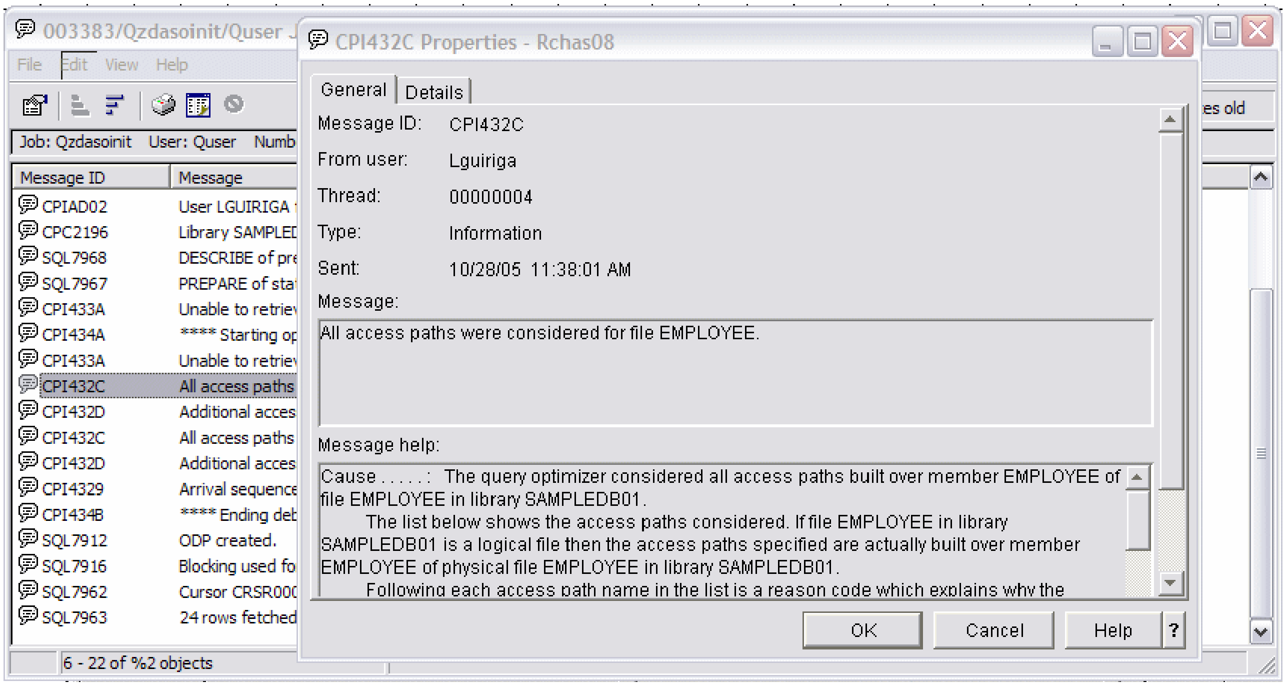


Job: Qzdasoinit User: Quser Number: 003383

Message ID	Message	Sent	Type	Severity
CPIAD02	User LGUIRIGA from client 9.10.132.109 connected to server.	10/28/05 11:37:49 AM	Information	0
CPC2196	Library SAMPLEDB01 added to library list.	10/28/05 11:37:49 AM	Completion	0
SQL7968	DESCRIBE of prepared statement STMT0005 completed.	10/28/05 11:38:01 AM	Completion	0
SQL7967	PREPARE of statement STMT0005 completed.	10/28/05 11:38:01 AM	Completion	0
CPI433A	Unable to retrieve query options file.	10/28/05 11:38:01 AM	Information	0
CPI434A	**** Starting optimizer debug message for query .	10/28/05 11:38:01 AM	Information	0
CPI433A	Unable to retrieve query options file.	10/28/05 11:38:01 AM	Information	0
CPI432C	All access paths were considered for file EMPLOYEE.	10/28/05 11:38:01 AM	Information	0
CPI432D	Additional access path reason codes were used.	10/28/05 11:38:01 AM	Information	0
CPI432C	All access paths were considered for file EMPLOYEE.	10/28/05 11:38:01 AM	Information	0
CPI432D	Additional access path reason codes were used.	10/28/05 11:38:01 AM	Information	0
CPI4329	Arrival sequence access was used for file EMPLOYEE.	10/28/05 11:38:01 AM	Information	0
CPI434B	**** Ending debug message for query .	10/28/05 11:38:01 AM	Information	0
SQL7912	ODP created.	10/28/05 11:38:01 AM	Information	0
SQL7916	Blocking used for query.	10/28/05 11:38:01 AM	Information	0
SQL7962	Cursor CRSR0005 opened.	10/28/05 11:38:01 AM	Completion	0
SQL7963	24 rows fetched from cursor CRSR0005.	10/28/05 11:38:01 AM	Completion	0

Figure 5-4 Job log debug messages

Select any of the debug messages that are displayed in the job log. You can click **File** → **Details** to obtain more detailed information about the debug message. Figure 5-5 shows an example of a detailed debug message that is displayed.



Message ID: CPI432C

From user: Lguiriga

Thread: 00000004

Type: Information

Sent: 10/28/05 11:38:01 AM

Message:

All access paths were considered for file EMPLOYEE.

Message help:

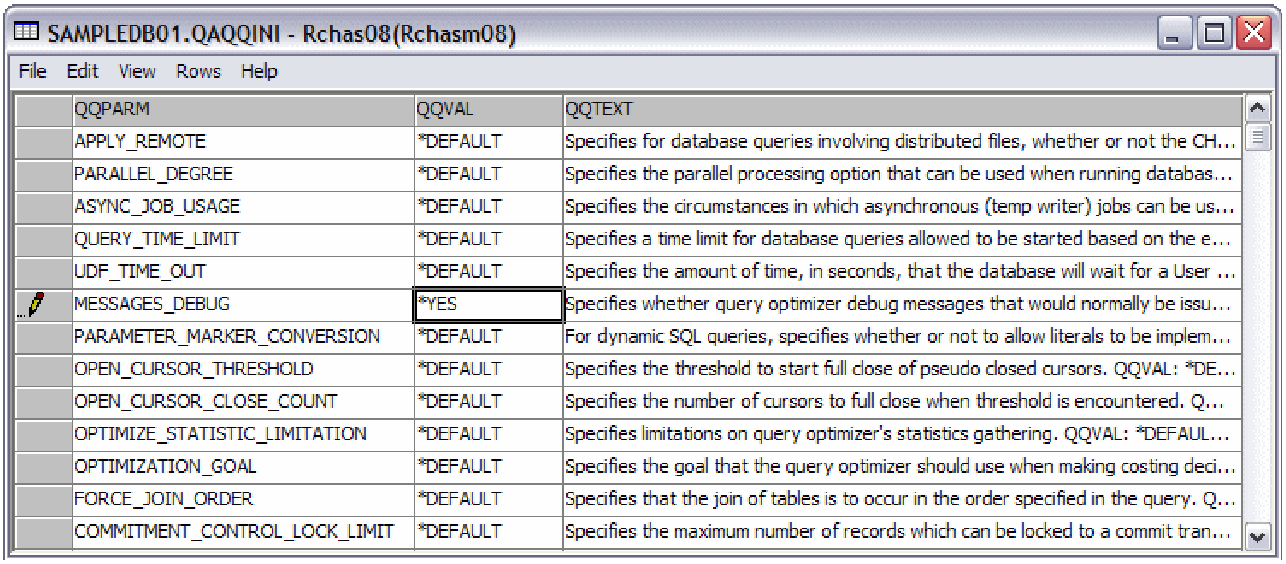
Cause . . . . . : The query optimizer considered all access paths built over member EMPLOYEE of file EMPLOYEE in library SAMPLEDB01.
The list below shows the access paths considered. If file EMPLOYEE in library SAMPLEDB01 is a logical file then the access paths specified are actually built over member EMPLOYEE of physical file EMPLOYEE in library SAMPLEDB01.
Following each access path name in the list is a reason code which explains why the

Figure 5-5 Detailed debug message information

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

1. Select the **QAQQINI** table that you want to use as explained in 5.1.8, "Query options table (QAQQINI)" on page 130.
2. In the QQAQQINI table, select the **MESSAGES_DEBUG** parameter, and change the parameter value as shown in Figure 5-6.



QQPARM	QQVAL	QQTEXT
APPLY_REMOTE	*DEFAULT	Specifies for database queries involving distributed files, whether or not the CH...
PARALLEL_DEGREE	*DEFAULT	Specifies the parallel processing option that can be used when running databas...
ASYNC_JOB_USAGE	*DEFAULT	Specifies the circumstances in which asynchronous (temp writer) jobs can be us...
QUERY_TIME_LIMIT	*DEFAULT	Specifies a time limit for database queries allowed to be started based on the e...
UDF_TIME_OUT	*DEFAULT	Specifies the amount of time, in seconds, that the database will wait for a User ...
MESSAGES_DEBUG	*YES	Specifies whether query optimizer debug messages that would normally be issu...
PARAMETER_MARKER_CONVERSION	*DEFAULT	For dynamic SQL queries, specifies whether or not to allow literals to be implem...
OPEN_CURSOR_THRESHOLD	*DEFAULT	Specifies the threshold to start full close of pseudo closed cursors. QQVAL: *DE...
OPEN_CURSOR_CLOSE_COUNT	*DEFAULT	Specifies the number of cursors to full close when threshold is encountered. Q...
OPTIMIZE_STATISTIC_LIMITATION	*DEFAULT	Specifies limitations on query optimizer's statistics gathering. QQVAL: *DEFAUL...
OPTIMIZATION_GOAL	*DEFAULT	Specifies the goal that the query optimizer should use when making costing deci...
FORCE_JOIN_ORDER	*DEFAULT	Specifies that the join of tables is to occur in the order specified in the query. Q...
COMMITMENT_CONTROL_LOCK_LIMIT	*DEFAULT	Specifies the maximum number of records which can be locked to a commit tran...

Figure 5-6 Enabling debug messages in QAQQINI

3. After you make the appropriate change, close the window.
4. In the Change Query Attributes window (Figure 5-40 on page 131), click **OK** to save your changes.

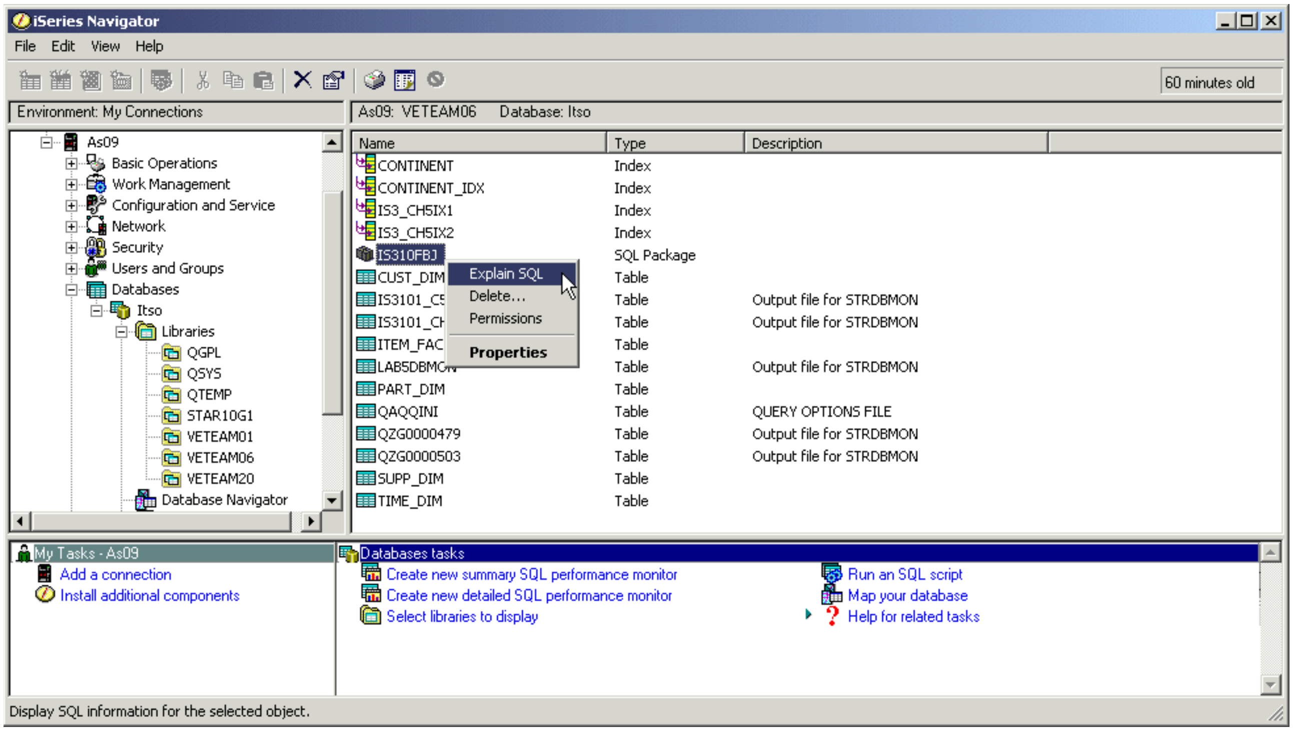
Later in this chapter, we provide additional information about managing the QAQQINI table.

> **Important:** Changes made to the QAQQINI table take effect 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.

5.1.2 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.

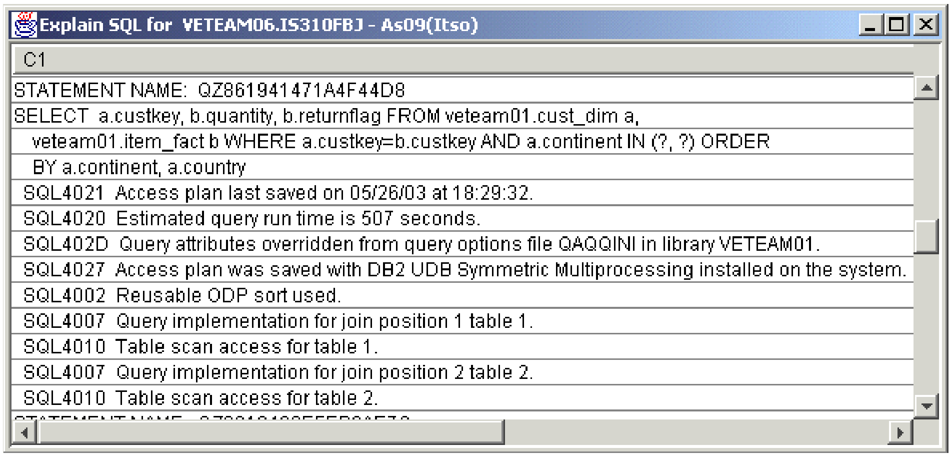
To view the information that pertains to the implementation and execution of your query, select an SQL package from iSeries Navigator. Right-click the ***SQL Package name*** and select **Explain SQL** as shown in Figure 5-7. This is equivalent to using the Print SQL Information (PRTSQLINF) CL command.



Name	Type	Description
CONTINENT	Index	
CONTINENT_IDX	Index	
IS3_CH5IX1	Index	
IS3_CH5IX2	Index	
IS310FBJ	SQL Package	
CUST_DIM	Table	
IS3101_C5	Table	Output file for STRDBMON
IS3101_CH	Table	Output file for STRDBMON
ITEM_FAC	Table	
LAB5DBMON	Table	Output file for STRDBMON
PART_DIM	Table	
QAQQINI	Table	QUERY OPTIONS FILE
QZG0000479	Table	Output file for STRDBMON
QZG0000503	Table	Output file for STRDBMON
SUPP_DIM	Table	
TIME_DIM	Table	

Figure 5-7 Selecting to Print SQL Information

The information in the SQL package is comparable to the debug messages which we discuss in 5.1.1, “Debug messages” on page 92. However, there is more detail in the first level SQLxxxx messages. Figure 5-8 shows an example of the SQL package information that can be displayed.



```
STATEMENT NAME:  QZ861941471A4F44D8
SELECT  a.custkey, b.quantity, b.returnflag FROM veteam01.cust_dim a,
  veteam01.item_fact b WHERE a.custkey=b.custkey AND a.continent IN (?, ?) ORDER
  BY a.continent, a.country
 SQL4021  Access plan last saved on 05/26/03 at 18:29:32.
 SQL4020  Estimated query run time is 507 seconds.
 SQL402D  Query attributes overridden from query options file QAQQINI in library VETEAM01.
 SQL4027  Access plan was saved with DB2 UDB Symmetric Multiprocessing installed on the system.
 SQL4002  Reusable ODP sort used.
 SQL4007  Query implementation for join position 1 table 1.
 SQL4010  Table scan access for table 1.
 SQL4007  Query implementation for join position 2 table 2.
 SQL4010  Table scan access for table 2.
```

Figure 5-8 Viewing the SQL package information

As you can see in Figure 5-8, information that pertains to the join order of the files, 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).

You can obtain this information by using the PRTSQLINF CL command as shown here:

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

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

Notes:

- The information retrieved from an SQL package may 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 re-optimized and the package or program is locked, then a new access plan is dynamically generated and placed in the plan cache (for SQE use). The version stored in the package or program is not updated.
- The information or messages that are shown with the PRTSQLINF were not enhanced for any of the new access methods used by SQE. This tool will become less and less strategic in the near future due to the more modern tools that are provided in V5R4.

5.1.3 Database monitoring tools: SQL Performance Monitors

Information that is gathered by the SQL Performance Monitors provides the most complete picture about a query's execution. You can gather performance statistics for a specific query or for every query on the server. In general, there are two categories of performance data (summary and detailed) and several ways to invoke the collection of the information. The two categories of performance data are:

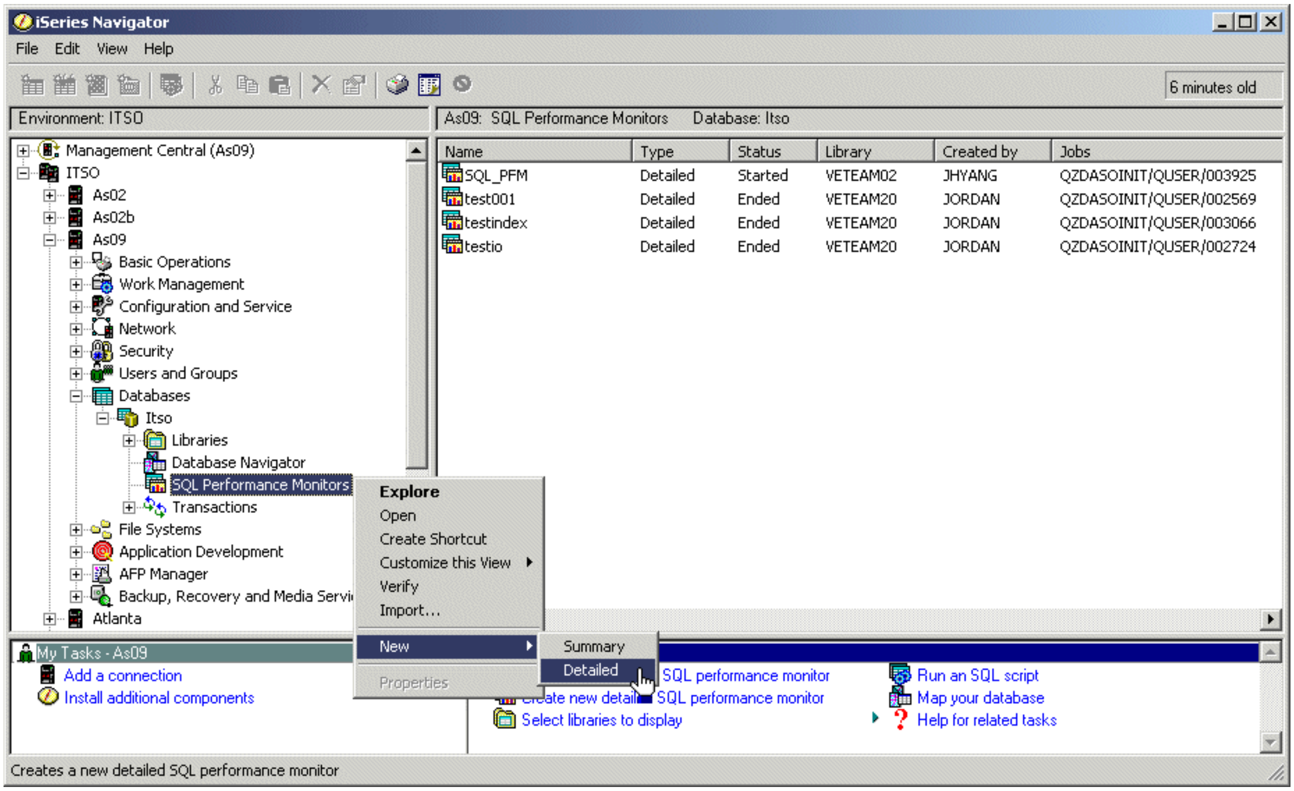
- **Summary SQL Performance Monitor**: Summary SQL performance data is collected via a memory-resident database monitoring capability that can be managed with application program interfaces (APIs) or with the iSeries Navigator. When the monitor is paused or ended, the data is written to disk so it can be analyzed. Since 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 Monitor**: Detailed SQL performance data is collected in real time and recorded immediately to a database table. The monitor does not need to be paused or ended to begin analysis of the results. Since this monitor saves the data in real time, it may have a greater impact on overall system performance, depending on how many jobs are monitored.

Running database performance monitor tools

There are two methods to start and end the collection of SQL performance data as explained in the following sections.

Using iSeries Navigator on OS/400 V5R2 and V5R3

The easiest way to start collecting performance data for a query is to use the function provided by iSeries Navigator as shown in Figure 5-9. As shown, in the left pane, select **SQL Performance Monitors**. With this function, you can select the option to create either a detailed or a summary SQL Performance Monitor. Right-click and select **New** → **Detailed**.



Name	Type	Status	Library	Created by	Jobs
SQL_PFM	Detailed	Started	VETEAM02	JHYANG	QZDASOINIT/QUSER/003925
test001	Detailed	Ended	VETEAM20	JORDAN	QZDASOINIT/QUSER/002569
testindex	Detailed	Ended	VETEAM20	JORDAN	QZDASOINIT/QUSER/003066
testio	Detailed	Ended	VETEAM20	JORDAN	QZDASOINIT/QUSER/002724

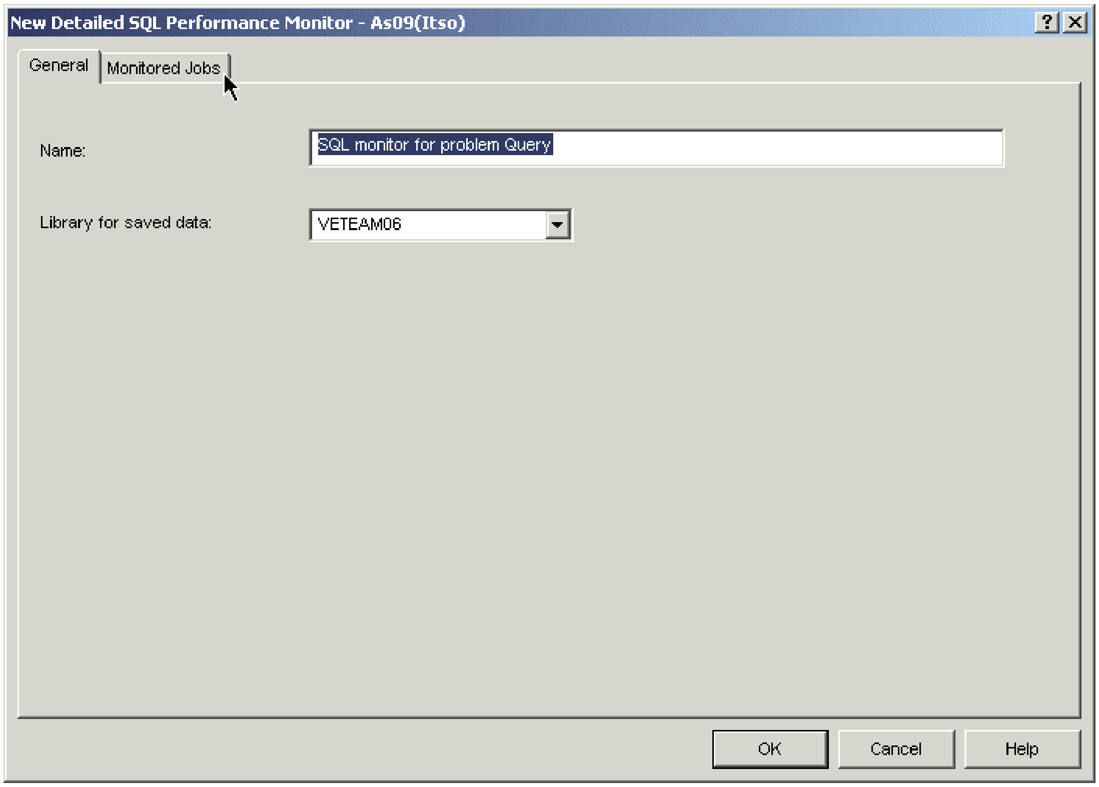
Creates a new detailed SQL performance monitor

Figure 5-9 Creating a new SQL Performance Monitor in V5R2 and V5R3

To collect detailed information about a single query job and invoke the analysis of the performance data:

1. In the New Detailed SQL Performance Monitor window (Figure 5-10), complete the following actions:

   a. On the **General** tab, type a name for the monitor and select a library name into which the performance data will be collected.



Name: SQL monitor for problem Query

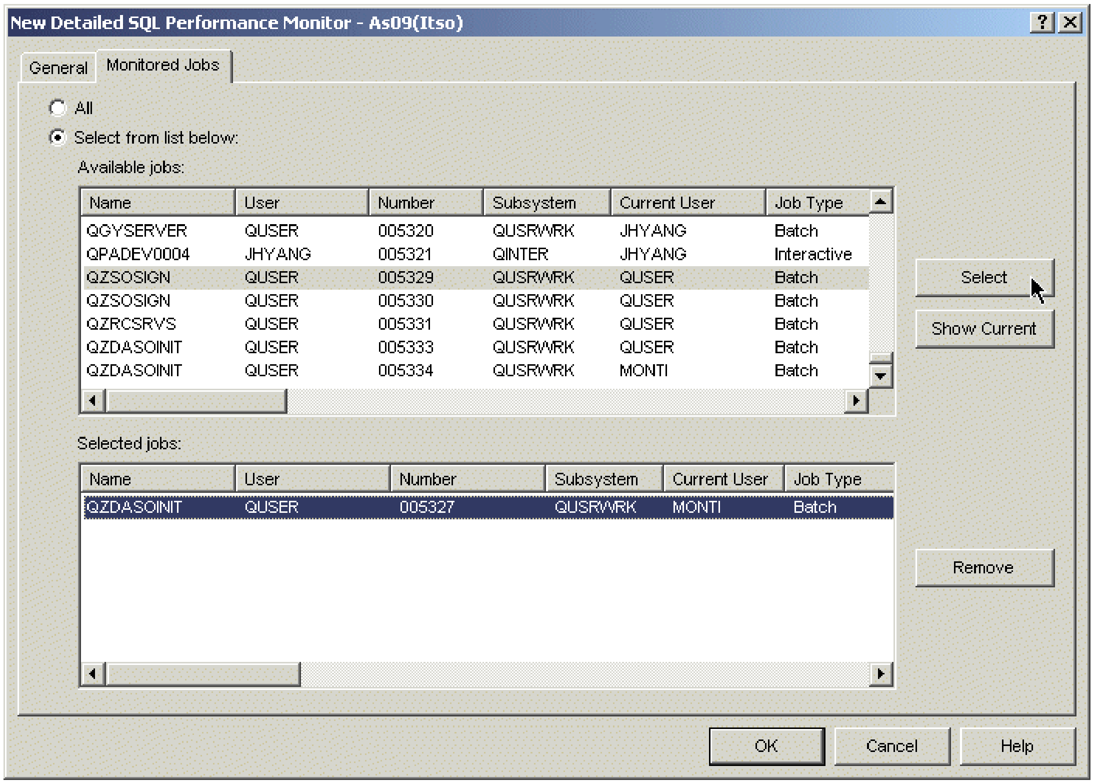
Library for saved data: VETEAM06

Figure 5-10 Specifying the name and location of the new SQL Performance Monitor

b. Click the **Monitored Jobs** tab.

c. On the Monitored Jobs page (Figure 5-11), specify the jobs for which you want to collect performance data. You see a list of all jobs that are currently active on your server. From here, you can monitor one or more jobs of your choice. Select the relevant jobs.

d. Click **OK**.



○ All

● Select from list below:

Available jobs:

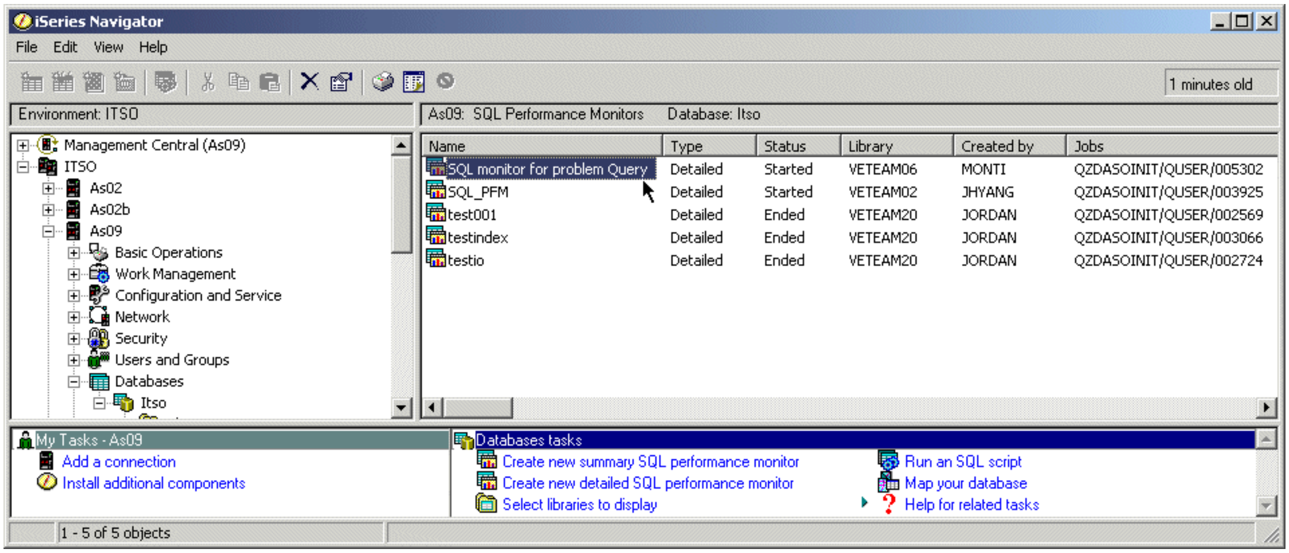
Name	User	Number	Subsystem	Current User	Job Type
QGYSERVER	QUSER	005320	QUSRWRK	JHYANG	Batch
QPADEV0004	JHYANG	005321	QINTER	JHYANG	Interactive
QZSOSIGN	QUSER	005329	QUSRWRK	QUSER	Batch
QZSOSIGN	QUSER	005330	QUSRWRK	QUSER	Batch
QZRCSRVS	QUSER	005331	QUSRWRK	QUSER	Batch
QZDASOINIT	QUSER	005333	QUSRWRK	QUSER	Batch
QZDASOINIT	QUSER	005334	QUSRWRK	MONTI	Batch

Selected jobs:

Name	User	Number	Subsystem	Current User	Job Type
QZDASOINIT	QUSER	005327	QUSRWRK	MONTI	Batch

Figure 5-11 Selecting jobs for the SQL Performance Monitor

2. In the right pane of iSeries Navigator (Figure 5-12), notice that your new SQL Performance Monitor is displayed with a status of *Started*. Your SQL Performance Monitor is now ready to start collecting the performance data of the queries running in the specified jobs.



Name	Type	Status	Library	Created by	Jobs
SQL monitor for problem Query	Detailed	Started	VETEAM06	MONTI	QZDASOINIT/QUSER/005302
SQL_PFM	Detailed	Started	VETEAM02	JHYANG	QZDASOINIT/QUSER/003925
test001	Detailed	Ended	VETEAM20	JORDAN	QZDASOINIT/QUSER/002569
testindex	Detailed	Ended	VETEAM20	JORDAN	QZDASOINIT/QUSER/003066
testio	Detailed	Ended	VETEAM20	JORDAN	QZDASOINIT/QUSER/002724

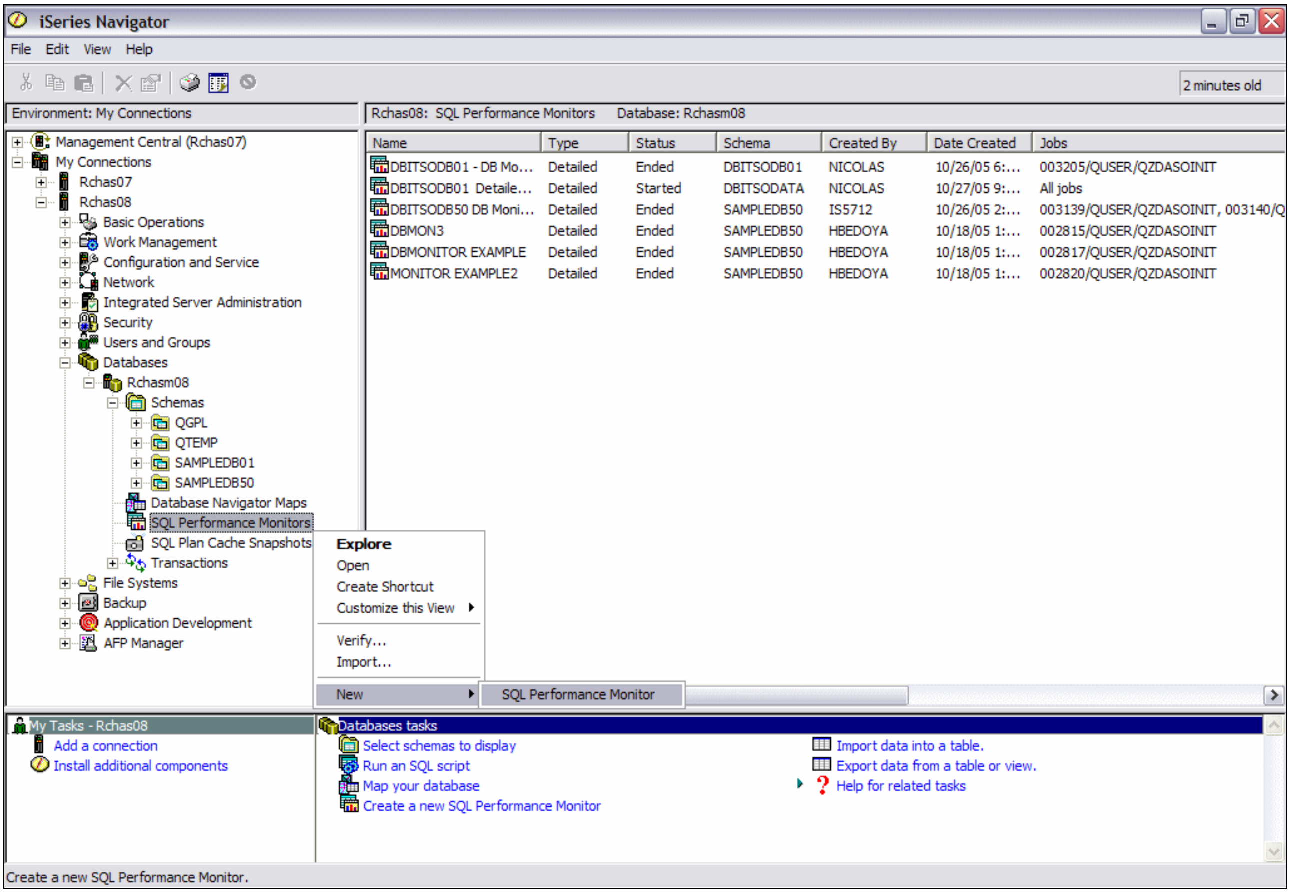
Figure 5-12 SQL Performance Monitor status

3. After you run the query, end the SQL Performance Monitor and analyze the performance data collected. To end the performance monitor, right-click the ***SQL Performance Monitor name*** and select **End**. Alternatively, you can select **File** → **End** from the menu bar.

> **Tip:** You can start, end, analyze, and delete performance data collections for your particular job directly from the Run SQL Scripts interface of iSeries Navigator. Simply use the Monitor option on the menu bar.

Using iSeries Navigator on i5/OS V5R4

In V5R4, a wizard is included to start an SQL Performance Monitor. In iSeries Navigator, you select **SQL Performance Monitors**, right-click, and select **New** → **SQL Performance Monitor** as shown in Figure 5-13.

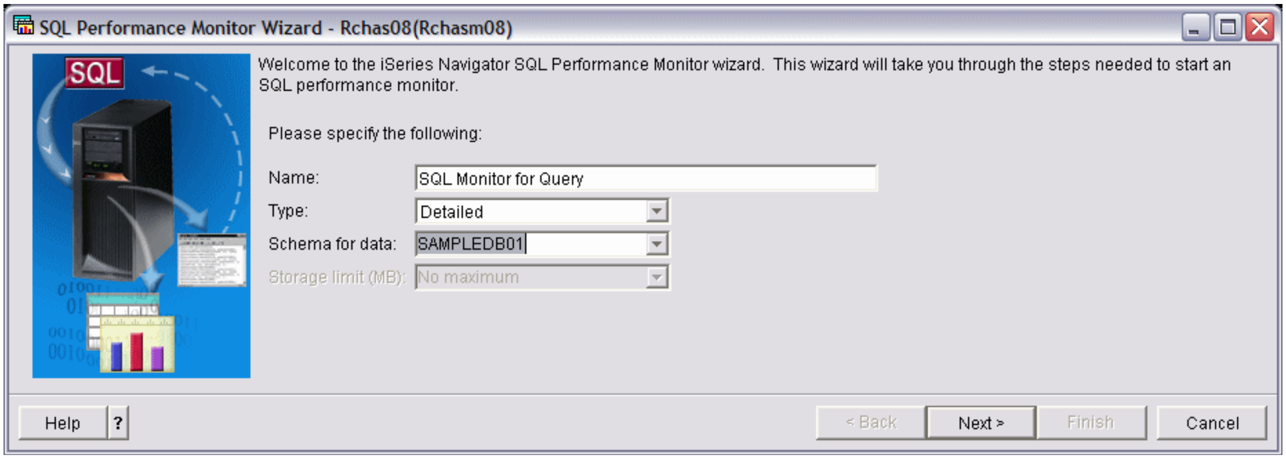


Name	Type	Status	Schema	Created By	Date Created	Jobs
DBITSODB01 - DB Mo...	Detailed	Ended	DBITSODB01	NICOLAS	10/26/05 6:...	003205/QUSER/QZDASOINIT
DBITSODB01 Detaile...	Detailed	Started	DBITSODATA	NICOLAS	10/27/05 9:...	All jobs
DBITSODB50 DB Moni...	Detailed	Ended	SAMPLEDB50	IS5712	10/26/05 2:...	003139/QUSER/QZDASOINIT, 003140/Q
DBMON3	Detailed	Ended	SAMPLEDB50	HBEDOYA	10/18/05 1:...	002815/QUSER/QZDASOINIT
DBMONITOR EXAMPLE	Detailed	Ended	SAMPLEDB50	HBEDOYA	10/18/05 1:...	002817/QUSER/QZDASOINIT
MONITOR EXAMPLE2	Detailed	Ended	SAMPLEDB50	HBEDOYA	10/18/05 1:...	002820/QUSER/QZDASOINIT

Figure 5-13 Creating a new SQL Performance Monitor in V5R4

With this function, you can create either a detailed or a summary SQL Performance Monitor. To collect detailed information about a single query job and invoke the analysis of the performance data:

1. In the SQL Performance Monitor Wizard window (Figure 5-14), specify a name for the monitor and a schema name into which the performance data will be collected. For Type, you can specify either a Detailed or Summary report from the monitor. In this example, we choose **Detailed**. Click **Next**.


SQL Performance Monitor Wizard - Rchas08(Rchasm08)

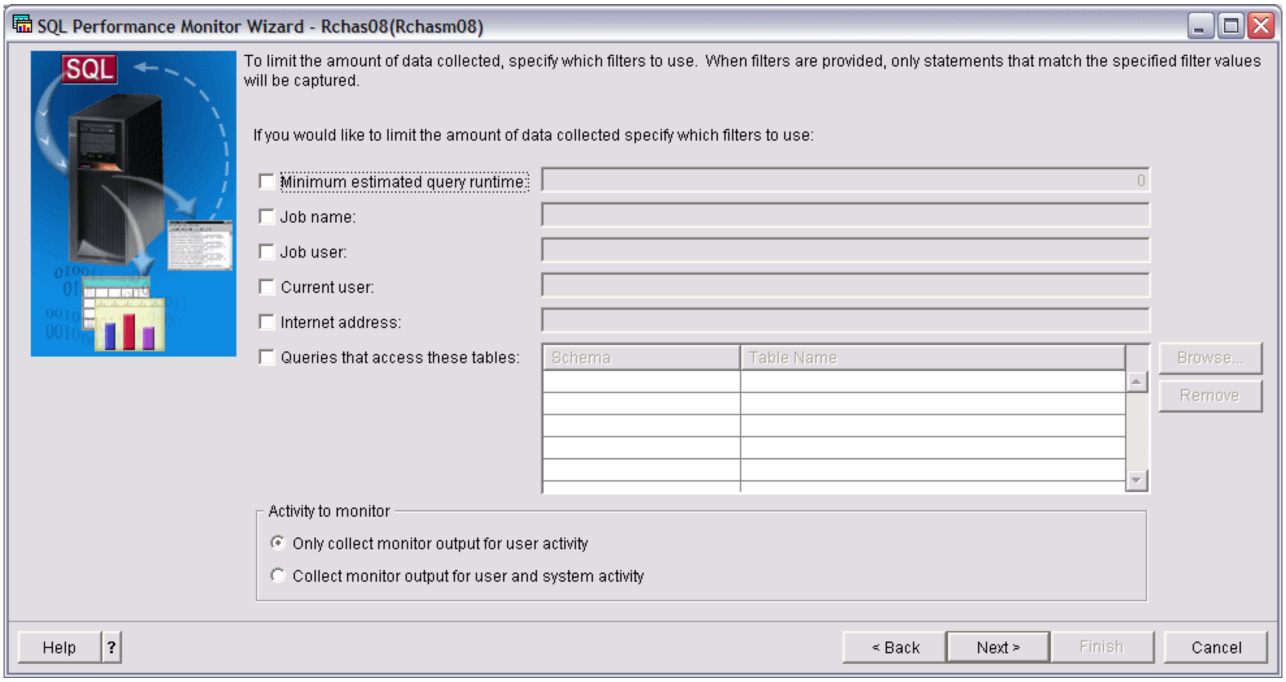
Welcome to the iSeries Navigator SQL Performance Monitor wizard. This wizard will take you through the steps needed to start an SQL performance monitor.

Please specify the following:

Name: SQL Monitor for Query
Type: Detailed
Schema for data: SAMPLEDB01
Storage limit (MB): No maximum

Figure 5-14 Specifying the name and location of the new SQL Performance Monitor

2. In the next panel (Figure 5-15), select the filters to obtain the information needed. Notice the different filters that have been added in V5R4. In V5R2 and V5R3, it was not possible to narrow the performance collection the way you can in V5R4.


SQL Performance Monitor Wizard - Rchas08(Rchasm08)

To limit the amount of data collected, specify which filters to use. When filters are provided, only statements that match the specified filter values will be captured.

If you would like to limit the amount of data collected specify which filters to use:

□ Minimum estimated query runtime: 0
□ Job name:
□ Job user:
□ Current user:
□ Internet address:
□ Queries that access these tables:

Schema	Table Name

Activity to monitor
● Only collect monitor output for user activity
○ Collect monitor output for user and system activity

Figure 5-15 Selecting filters

In V5R4, when you create a detailed monitor, you can limit the amount of data that is collected, by specifying which filter to use to capture only the information that you want. The different filters are explained here:

- Minimum estimated query runtime

  Select this filter to include queries that exceed a specified amount of time. Select a number and then a unit of time.

- Job name

  Select this option to filter by a specific job name. Specify a job name in the field. You can specify the entire ID or use a wildcard. For example, QZDAS* finds all jobs where the name starts with QZDAS.

- Job user

  Select this option to filter by job user. Specify a user ID in the field. You can specify the entire ID or use a wildcard. For example, QUSER* finds all user IDs where the name starts with QUSER.

- Current user

  Select this option to filter by the current user of the job. Specify a user ID in the field. You can specify the entire ID or use a wildcard. For example, QSYS* finds all users where the name starts with QSYS.

- Internet address

  Select this option to filter by Internet access. The format must be *nnn.nnn.nnn.nnn*, for example, `5.5.199.199`.

- Only queries that access these tables

  Select this option to filter by only queries that use certain tables. Click **Browse** to select the tables to include. To remove a table from the list, select the table and click **Remove**. You can specify a maximum of ten table names.

- Activity to monitor

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

Click **Next**.

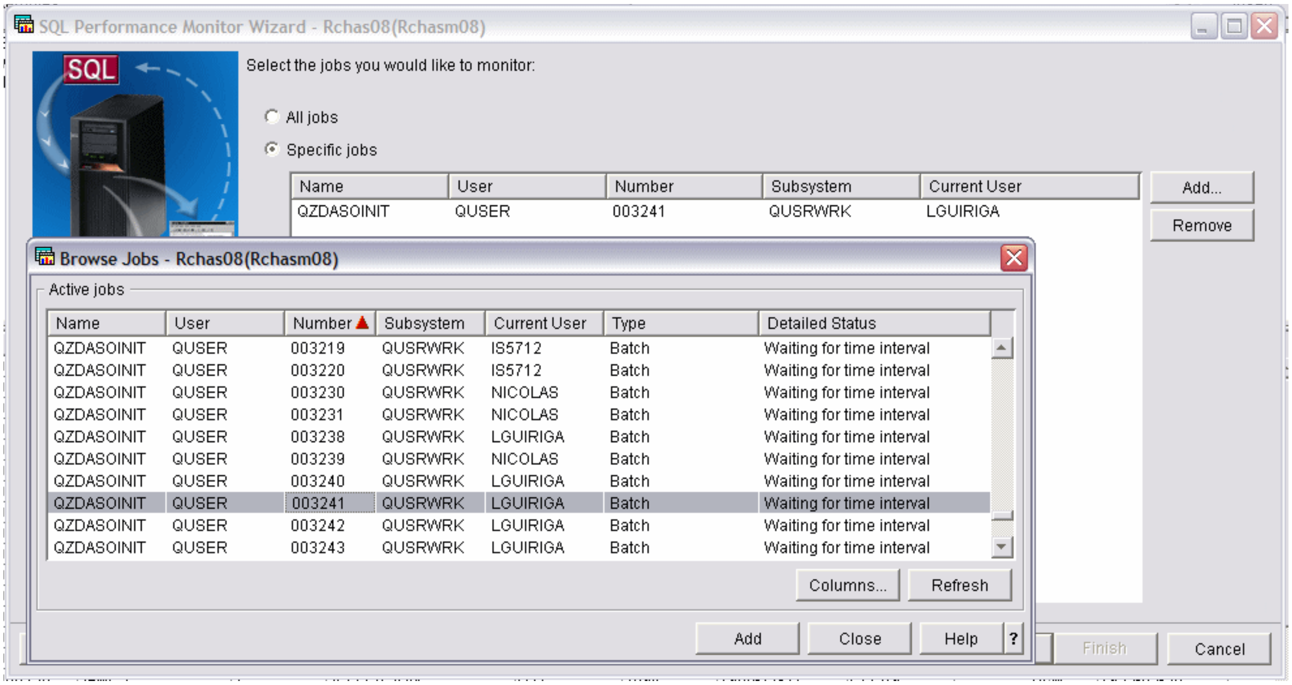
3. If you did not narrow your performance collection by using the filters, when you click the Next button, the Select the jobs panel (Figure 5-16) opens.

   a. On this panel, click the **Add** button and then the Browse Jobs window (see inset in the lower left corner of Figure 5-16) opens.

   b. Select the jobs that you want to collect performance data by clicking the job and then click the **Add** button.

   c. Click **Next**.

   If you did select any of the filters, then after you click the Next button, the details panel (Figure 5-17) opens.



SQL Performance Monitor Wizard - Rchas08(Rchasm08)
SQL

Select the jobs you would like to monitor:

○ All jobs

● Specific jobs

Name	User	Number	Subsystem	Current User
QZDASOINIT	QUSER	003241	QUSRWRK	LGUIRIGA

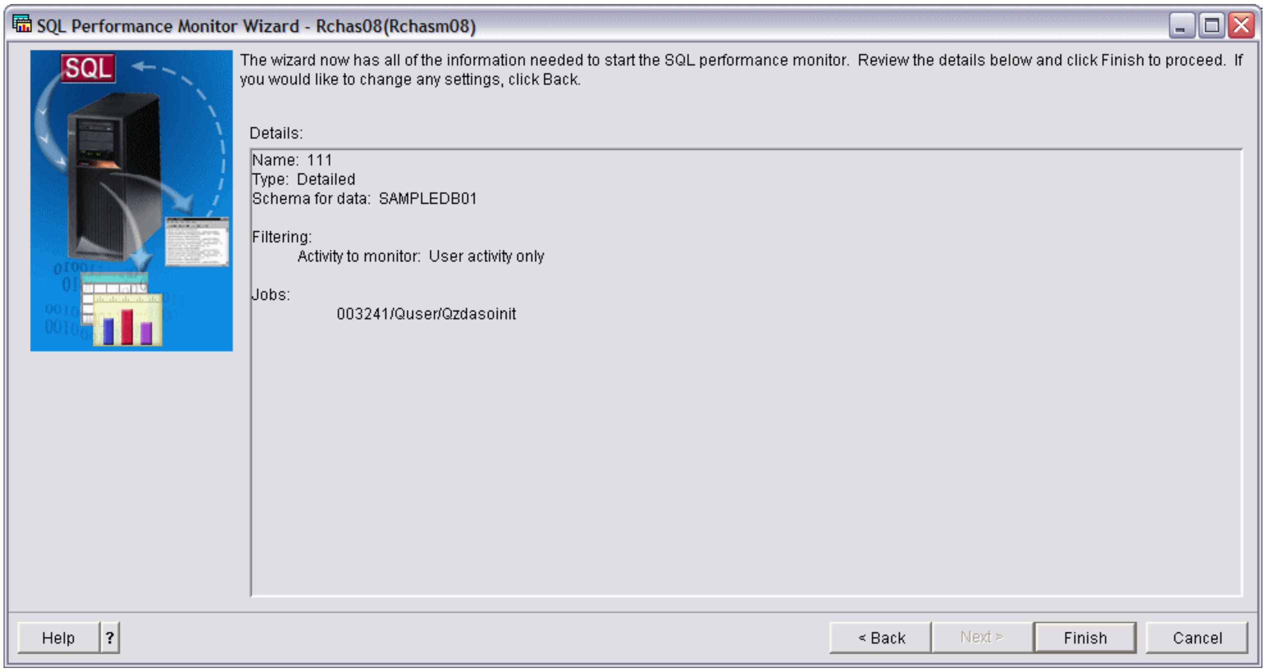
Add...
Remove
Browse Jobs - Rchas08(Rchasm08)

Active jobs

Name	User	Number ▲	Subsystem	Current User	Type	Detailed Status
QZDASOINIT	QUSER	003219	QUSRWRK	IS5712	Batch	Waiting for time interval
QZDASOINIT	QUSER	003220	QUSRWRK	IS5712	Batch	Waiting for time interval
QZDASOINIT	QUSER	003230	QUSRWRK	NICOLAS	Batch	Waiting for time interval
QZDASOINIT	QUSER	003231	QUSRWRK	NICOLAS	Batch	Waiting for time interval
QZDASOINIT	QUSER	003238	QUSRWRK	LGUIRIGA	Batch	Waiting for time interval
QZDASOINIT	QUSER	003239	QUSRWRK	NICOLAS	Batch	Waiting for time interval
QZDASOINIT	QUSER	003240	QUSRWRK	LGUIRIGA	Batch	Waiting for time interval
QZDASOINIT	QUSER	003241	QUSRWRK	LGUIRIGA	Batch	Waiting for time interval
QZDASOINIT	QUSER	003242	QUSRWRK	LGUIRIGA	Batch	Waiting for time interval
QZDASOINIT	QUSER	003243	QUSRWRK	LGUIRIGA	Batch	Waiting for time interval

Figure 5-16 Selecting jobs for SQL Performance Monitor

4. Figure 5-17 shows the details of the monitored jobs. Review this page and click **Finish**.



The wizard now has all of the information needed to start the SQL performance monitor. Review the details below and click Finish to proceed. If you would like to change any settings, click Back.

Details:

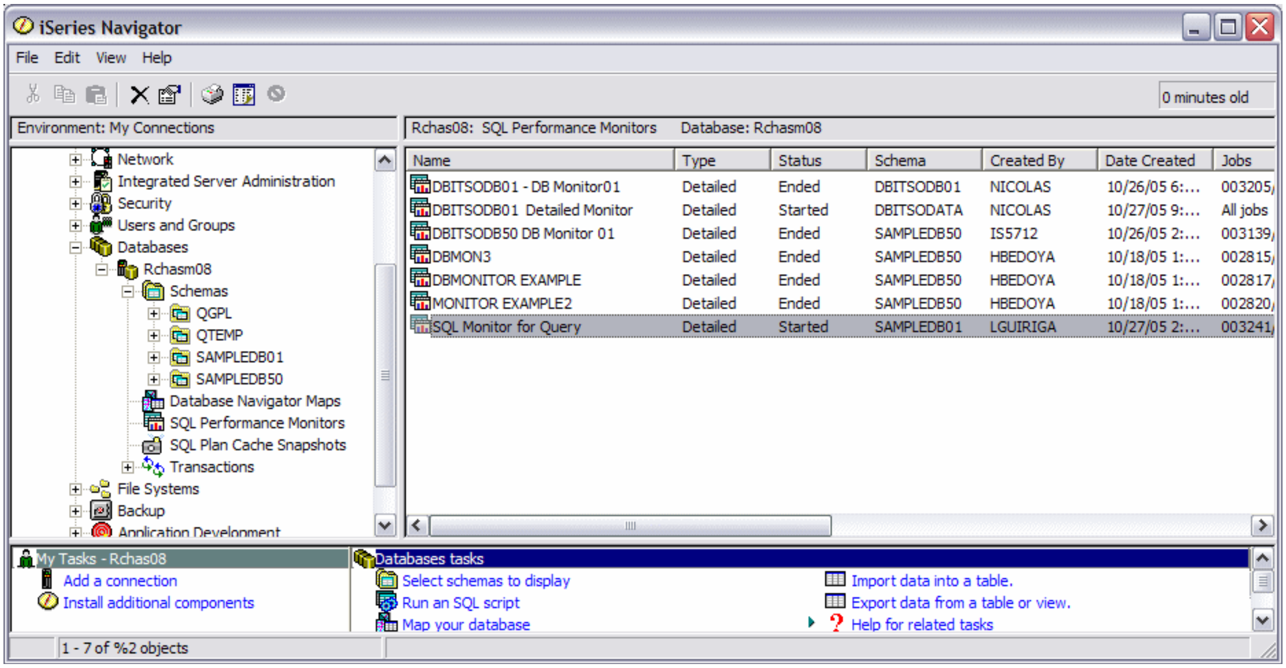
Name: 111
Type: Detailed
Schema for data: SAMPLEDB01

Filtering:
Activity to monitor: User activity only

Jobs:
003241/Quser/Qzdasoinit

Figure 5-17 Details of the monitored jobs in the SQL Performance Monitor Wizard

5. In iSeries Navigator, notice that your new SQL Performance Monitor is displayed with a status of *Started* in the right pane (Figure 5-18). Your SQL Performance Monitor is now ready to start collecting the performance data of the queries running in the specified jobs.



Rchas08: SQL Performance Monitors   Database: Rchasm08

Name	Type	Status	Schema	Created By	Date Created	Jobs
DBITSODB01 - DB Monitor01	Detailed	Ended	DBITSODB01	NICOLAS	10/26/05 6:...	003205/
DBITSODB01 Detailed Monitor	Detailed	Started	DBITSODATA	NICOLAS	10/27/05 9:...	All jobs
DBITSODB50 DB Monitor 01	Detailed	Ended	SAMPLEDB50	IS5712	10/26/05 2:...	003139/
DBMON3	Detailed	Ended	SAMPLEDB50	HBEDOYA	10/18/05 1:...	002815/
DBMONITOR EXAMPLE	Detailed	Ended	SAMPLEDB50	HBEDOYA	10/18/05 1:...	002817/
MONITOR EXAMPLE2	Detailed	Ended	SAMPLEDB50	HBEDOYA	10/18/05 1:...	002820/
SQL Monitor for Query	Detailed	Started	SAMPLEDB01	LGUIRIGA	10/27/05 2:...	003241/

Figure 5-18 SQL Performance Monitor status

6. After you run the query, end the SQL Performance Monitor and analyze the performance data collected. To end the performance monitor, right-click the ***SQL Performance Monitor name*** and select **End**. Alternatively you can select **File** → **End** from the menu bar.

Using the Start Database Monitor command

The Start Database Monitor (STRDBMON) command is an i5/OS CL command that provides another interface into the Detailed SQL Performance Monitor data collection facility. To start collecting performance data for your query, use the STRDBMON CL command as shown in the following example:

```
STRDBMON OUTFILE(SAMPLEDB01/IS3101_CH5) JOB(003241/QUSER/QZDASOINIT) TYPE(*SUMMARY)
COMMENT('Example of STRDBMON')
```

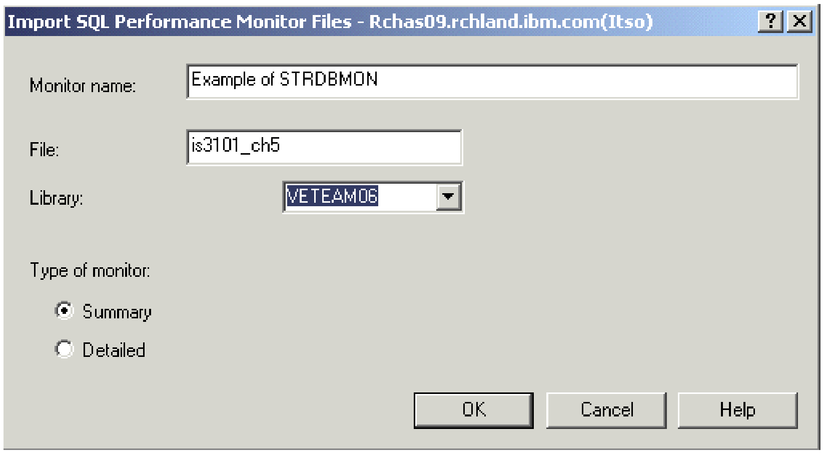
After you run your query, you can stop the collection of performance data by using the End Database Monitor (ENDDBMON) CL command, as shown here:

```
ENDDBMON JOB(003241/QUSER/QZDASOINIT)
```

You can now proceed to analyze the collected performance data. For ease of use, the structure and layout of the performance data are identical to the collection created by the SQL Performance Monitor tool that we explained in “Using iSeries Navigator on i5/OS V5R4” on page 101. This format allows you to use the predefined reports in iSeries Navigator to analyze performance data.

To analyze Database Monitor data using the predefined reports, you must import the performance data collection into SQL Performance Monitor:

1. In iSeries Navigator, select **SQL Performance Monitors**.
2. Import the data. Select **File** → **Import**.
3. Specify the details for importing your Database Monitor performance data. For V5R2 and V5R3, you do this in the Import SQL Performance Monitor Files window (Figure 5-19).



Import SQL Performance Monitor Files - Rchas09.rchland.ibm.com(Itso)

Monitor name: Example of STRDBMON

File: is3101_ch5

Library: VETEAM06

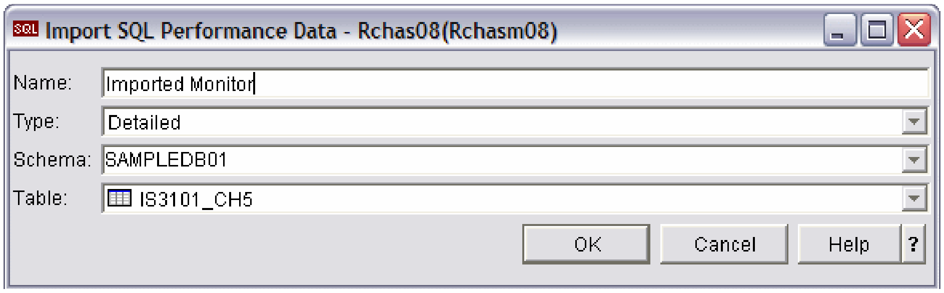
Type of monitor:

● Summary

○ Detailed

Figure 5-19 Import SQL Performance Monitor Files window in V5R2 and V5R3

For V5R4, you specify the details for importing the data in the Import SQL Performance Data window shown in Figure 5-20.



Name: Imported Monitor

Type: Detailed

Schema: SAMPLEDB01

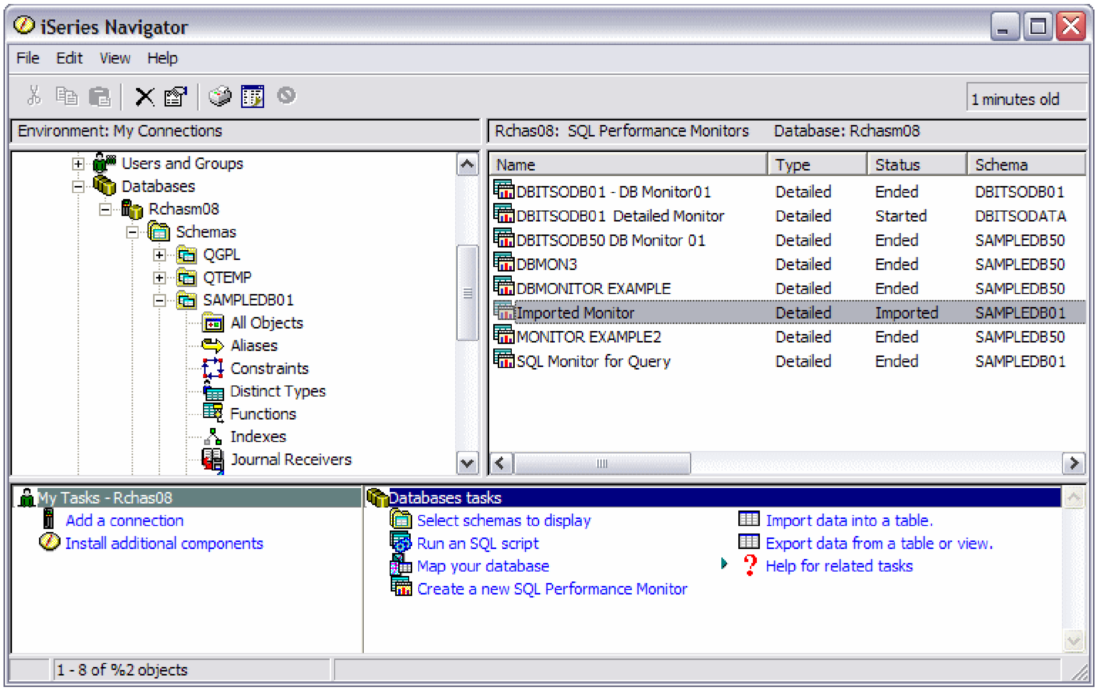
Table: IS3101_CH5

Figure 5-20 Import SQL Performance Data window in V5R4

Assign a name for your SQL Performance Monitor data collection and specify the file or table and library or schema that contains the performance data.

Back in iSeries Navigator (Figure 5-21), your newly created SQL Performance Monitor now reflects a status of *Imported*. Analyze the data using the predefined SQL Performance Monitor reports as explained in the next section.

Note: It is possible to run the Database Monitor on one System i machine and import the data collected to a separate system to perform the analysis.



Rchas08: SQL Performance Monitors Database: Rchasm08

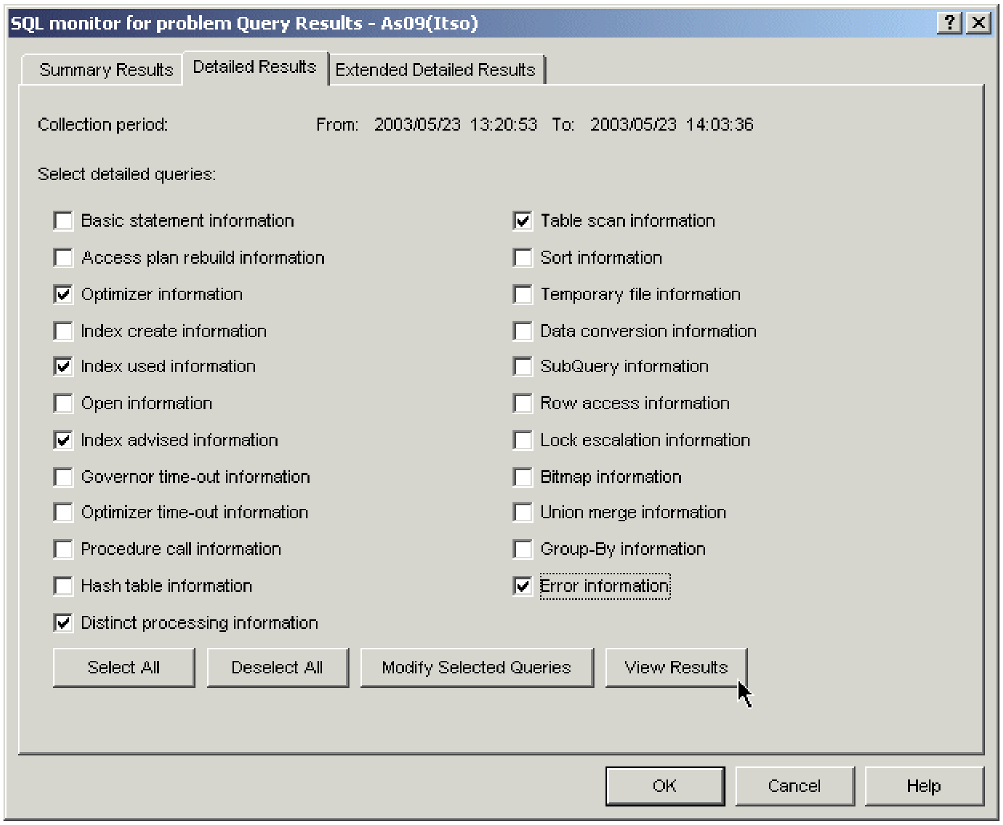
Name	Type	Status	Schema
DBITSODB01 - DB Monitor01	Detailed	Ended	DBITSODB01
DBITSODB01 Detailed Monitor	Detailed	Started	DBITSODATA
DBITSODB50 DB Monitor 01	Detailed	Ended	SAMPLEDB50
DBMON3	Detailed	Ended	SAMPLEDB50
DBMONITOR EXAMPLE	Detailed	Ended	SAMPLEDB50
Imported Monitor	Detailed	Imported	SAMPLEDB01
MONITOR EXAMPLE2	Detailed	Ended	SAMPLEDB50
SQL Monitor for Query	Detailed	Ended	SAMPLEDB01

Figure 5-21 Analyzing the Database Monitor data using SQL Performance Monitor

Analyzing SQL performance data on OS/400 V5R2 and V5R3

To access the collected performance data using iSeries Navigator:

1. Select the relevant SQL Performance Monitor. Select **File** → **Analyze Results**.
2. In the SQL monitor for problem Query Results window (Figure 5-22), you can select from a variety of predefined reports to view performance-related information about your query. Select the reports that you want to view. Click **View Results** to automatically format and display the reports.



Collection period: From: 2003/05/23 13:20:53 To: 2003/05/23 14:03:36

Select detailed queries:

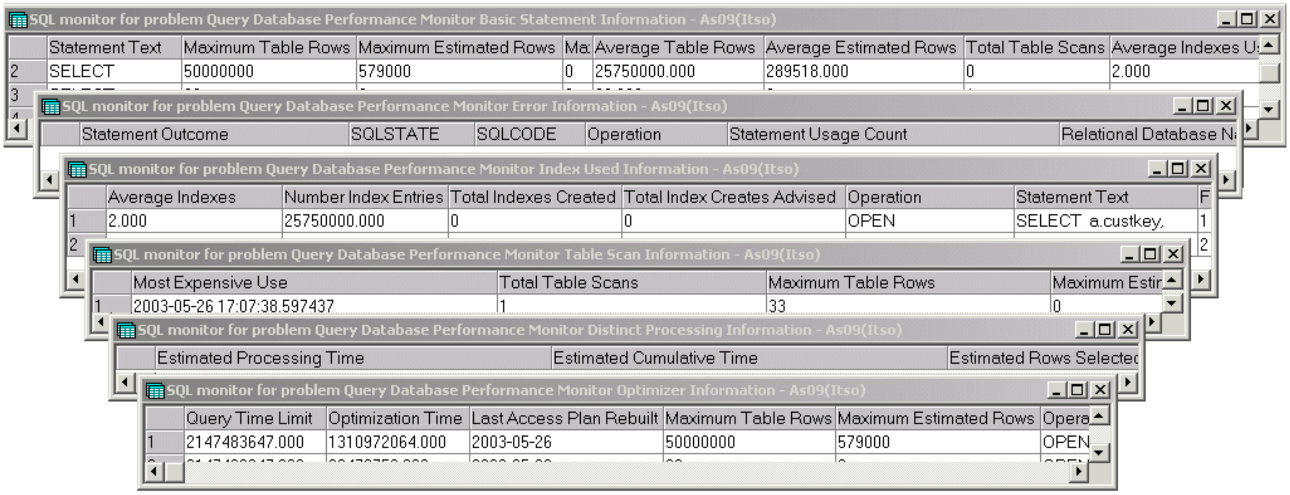
□ Basic statement information
□ Access plan rebuild information
☑ Optimizer information
□ Index create information
☑ Index used information
□ Open information
☑ Index advised information
□ Governor time-out information
□ Optimizer time-out information
□ Procedure call information
□ Hash table information
☑ Distinct processing information
☑ Table scan information
□ Sort information
□ Temporary file information
□ Data conversion information
□ SubQuery information
□ Row access information
□ Lock escalation information
□ Bitmap information
□ Union merge information
□ Group-By information
☑ Error information

Figure 5-22 Selecting SQL Performance Monitor reports

3. Figure 5-23 shows examples of some predefined reports that are currently available. By analyzing the contents of these reports, you can identify problem areas in your queries and take the appropriate steps to remedy such problems.

   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. To retrieve the SQL statements, select the report (Figure 5-22) that you need and then select **Modify Selected Queries**.

   The SQL statements are retrieved and displayed in the iSeries Navigator Run SQL Scripts window, from where you can modify and execute it.



SQL monitor for problem Query Database Performance Monitor Basic Statement Information - As09(Itso)

	Statement Text	Maximum Table Rows	Maximum Estimated Rows	Ma	Average Table Rows	Average Estimated Rows	Total Table Scans	Average Indexes U
2	SELECT	50000000	579000	0	25750000.000	289518.000	0	2.000

SQL monitor for problem Query Database Performance Monitor Error Information - As09(Itso)

	Statement Outcome	SQLSTATE	SQLCODE	Operation	Statement Usage Count	Relational Database N

SQL monitor for problem Query Database Performance Monitor Index Used Information - As09(Itso)

	Average Indexes	Number Index Entries	Total Indexes Created	Total Index Creates Advised	Operation	Statement Text	F
1	2.000	25750000.000	0	0	OPEN	SELECT a.custkey,	1

SQL monitor for problem Query Database Performance Monitor Table Scan Information - As09(Itso)

	Most Expensive Use	Total Table Scans	Maximum Table Rows	Maximum Estir
1	2003-05-26 17:07:38.597437	1	33	0

SQL monitor for problem Query Database Performance Monitor Distinct Processing Information - As09(Itso)

	Estimated Processing Time	Estimated Cumulative Time	Estimated Rows Selectec

SQL monitor for problem Query Database Performance Monitor Optimizer Information - As09(Itso)

	Query Time Limit	Optimization Time	Last Access Plan Rebuilt	Maximum Table Rows	Maximum Estimated Rows	Opera
1	2147483647.000	1310972064.000	2003-05-26	50000000	579000	OPEN

Figure 5-23 SQL Performance Monitor predefined reports

SQL performance reports information

The predefined SQL Performance Monitor Summary Results reports include the following information:

- **General Summary**: Contains information that summarizes all SQL activity. This information provides the user with a high-level indication of the nature of the SQL statements used. For example, it may indicate how much SQL is 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**: Contains a row of information for each job. Each row summarizes all SQL activity for that job. This information can be used to indicate which jobs on the system are the heaviest users of SQL, and therefore, which ones are candidates for performance tuning. The user may then want to start a separate detailed performance monitor on an individual job for more detailed information without monitoring the entire system.
- **Operation Summary**: Contains a row of summary information for each type of SQL operation. Each row summarizes all SQL activity for that type of SQL operation. This information provides the user a high-level indication of the type of SQL statements that are used. For example, it may indicate whether the applications are mainly read-only or whether there is a large amount of update, delete, or insert activity. This information can be used to try specific performance tuning techniques. For example, if a large amount of INSERT activity is occurring, you may want to use the Override Database File (OVRDBF) command to increase the blocking factor or use the QDBENCWT API.
- **Program Summary**: Contains a row of information for each program that performed SQL operations. Each row summarizes all SQL activity for that program. This information can be used to identify which programs use the most or most expensive SQL statements. The programs are then potential candidates for performance tuning. 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 Object Link Embedded (OLE) Database (DB) have a blank program name unless they result from a procedure, function, or trigger.
- **SQL Attributes Summary**: Contains a summary of the optimization attributes. This option is available only when you use a detailed SQL Performance Monitor. This information provides the user a high-level indication of some of the key SQL attributes that are used. This can help identify attributes that potentially are more costly than others. For example, in some cases, ALWCPYDTA(*YES) can allow the optimizer to run a query faster if live

data is not needed by the application. Also, *ENDMOD and *ENDPGM are much more expensive to run than *ENDJOB or *ENDACTGRP.

- **Isolation Level Summary**: Contains a summary of the number of statements that were run under each isolation level. This option is available only when you use a detailed SQL Performance Monitor. This information provides the user a high-level indication of the isolation level used. 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 may likely produce a high level of contention. The lowest level isolation level that still satisfies the application design requirement should always be used.
- **Error Summary**: Contains a summary of any SQL statement error messages or warnings that were captured by the monitor.
- **Statement Use Summary**: Contains a summary of the number of statements that are executed and the number of times they are executed during the performance monitor collection period. This option is available only when you use a detailed SQL Performance Monitor. This information provides the user a high-level indication of how often the same SQL statements are used multiple times. If the same SQL statement is used more than once, it may be cached and subsequent uses of the same statement are less expensive. It is more important to tune an SQL statement that is executed many times than to tune an SQL statement that is run only once.
- **Open Summary**: 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. This option is available only when you use a detailed SQL Performance Monitor.

  This information provides the user a high-level indication of how often an Open Data Path (ODP) is reused. The first open of a query in a job is a full open. After this, the ODP may be pseudo-closed and then reused. An open of a pseudo-closed ODP is far less expensive than a full open. The user can control when an ODP is pseudo-closed and how many pseudo-closed ODPs are allowed in a job by using the Change Query Attributes action in the Database Folder of iSeries Navigator. In rare cases, an ODP is not reusable. High usage of non-reusable ODPs may indicate that the SQL statements causing the non-reusable ODPs should be rewritten.
- **Data Access Summary**: Contains a summary of the number of SQL statements that are read-only versus those that modify data. This option is available only when you use a detailed SQL Performance Monitor. This information provides the user with a less detailed view of the type of SQL statements that are used than what is available through the Operation Summary. This information can be used to try specific performance tuning techniques. For example, if a large amount of INSERT activity is occurring, you may want to use the OVRDBF command to increase the blocking factor or use the QDBENCWT API.
- **Statement Type Summary**: Contains a summary of whether SQL statements are in extended dynamic packages, system-wide statement cache, regular dynamic, or static SQL statements. This option is available only when you use a detailed SQL Performance Monitor.

  This information provides the user a high-level indication of the number of the SQL statements that were fully parsed and optimized (dynamic) or whether the SQL statements and access plans were stored either statically in a program, procedure, function, package, or trigger. An SQL statement that must be fully parsed and optimized is more expensive than the same statement that is static, extended dynamic, or cached in the system-wide statement cache.

- **Parallel Processing Summary**: Contains a summary of the parallel processing techniques used. This option is available only when you use a detailed SQL Performance Monitor.

  This information provides the user a high-level indication of whether one of the many parallel processing techniques was used to execute the SQL statements. Most parallel processing techniques are available only if Symmetric Processing for iSeries is installed. After the option is installed, the user must specify the degree of parallelism using the Change Query Attributes action in the Database Folder of iSeries Navigator, the Change Query Attribute (CHGQRYA) CL command, or the QQRYDEGREE system value.

- **Optimizer Summary**: Contains a summary of the optimizer techniques used. This option is available only when you use a detailed SQL Performance Monitor. This information provides the user a high-level indication of the types of queries and optimizer attributes used. The user can use this information to determine whether the types of queries are complex (use of subqueries or joins) and to identify attributes that may deserve further investigation.

  For example, an access plan rebuild occurs when the prior access plan is no longer valid or if a change has occurred that identified a better access plan. If the number of access plan rebuilds is high, it may indicate that some application redesign may be necessary. Also, if the join order is forced, this may indicate that the access plan chosen is not the most efficient. However, it may also indicate that someone has already tuned the SQL statement and explicitly forced the join order because experimentation showed that a specific join order should always provide the best result. Forcing the join order should be used sparingly. It prevents the optimizer from analyzing any join order than the one that is specified.

The Detailed Results reports contain additional and more detailed information. This information includes:

- **Basic statement information**: Provides the user with basic information about each SQL statement. The most expensive SQL statements are presented first in the list, so at a glance, the user can see which statements (if any) were long running.
- **Access plan rebuild information**: Contains a row of information for each SQL statement that required the access plan to be rebuilt. Re-optimization 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. Excessive access plan rebuilds may indicate a problem.
- **Optimizer information**: Contains a row of optimization information for each subselect in an SQL statement. This information provides the user with basic optimizer information about SQL statements that involve data manipulation (selects, opens, updates, and so on). The most expensive SQL statements are presented first in the list.
- **Index create information**: Contains a row of information for each SQL statement that required an index to be created. Temporary indexes may need to be created for several reasons such as to perform a join, to support scrollable cursors, or to implement Order by or Group by.

  The indexes that are created may contain keys only for rows that satisfy the query. Such indexes are known as *sparse indexes*. In many cases, index creation may be perfectly normal and the most efficient way to perform the query. However, if the number of rows is large, or if the same index is repeatedly created, you can create a permanent index to improve performance of this query. This may be true regardless of whether an index was advised.

- **Index used information**: Contains a row of information for each permanent index that an SQL statement used. This information can be used to tell quickly if any of the permanent indexes were used to improve the performance of a query.

Permanent indexes are typically necessary to achieve optimal query performance. This information can also help determine how often a permanent index was used by in the statements that were monitored. Indexes that are never (or rarely) used should be dropped to improve the performance of inserts, updates, and deletes to a table. Before dropping the index, you may want to look at the last used date in the description information for the index.

- **Open information**: Contains a row of information for each open activity for each SQL statement. The first time (or times) that an open occurs 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 input/output (I/O) operation. An ODP may be cached at close time so that, if the SQL statement is run again during the job, the ODP is reused. Such an open is called a *pseudo open*, which is less expensive than a full open.
- **Index advised information**: Provides information about advised indexes. This option is available only when you use a detailed SQL Performance Monitor. It 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 usually improves performance, this is not a guarantee. After the index is created, more accurate estimates of the actual costs are available. The optimizer may decide, based on this new information, that the cost of using the index is too high. Even if the optimizer does not use the index to implement the query, the new estimates available from the new index provide more detailed information to the optimizer that may produce better performance.
- **Governor timeout information**: Provides information about all optimizer timeouts. This option is available only when you use a detailed SQL Performance Monitor. This can be used to determine how often users attempt to run queries that would exceed the governor timeout value. A large number of governor timeouts may indicate that the timeout value is set too low.
- **Optimizer timeout information**: 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 may decide that further time spent optimizing is no longer reasonable and use the best access plan up to that point. This may not be the best plan available.

  If the SQL statement is run 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 may be possible by enabling extended dynamic package support or by using static SQL in a procedure or program. Since many dynamic SQL statements can be cached in the system-wide statement cache, optimizer timeouts are not common.
- **Procedure call information**: Provides information about procedure call usage. This option is 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 returning results, rather than to send each individual SQL statement to the server.
- **Hash table information**: Provides information about any temporary hash tables that were used. This option is available only when you use a detailed SQL Performance Monitor.

Hash join and hash grouping may 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 may become a problem.

- **Distinct processing information**: 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 may be a time consuming operation because a final sort may be needed 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.
- **Table scan**: Contains a row of information for each subselect that required records to be processed in arrival sequence order. Table scans of large tables can be time consuming. A long running SQL statement may indicate that an index is required to improve performance.
- **Sort information**: Contains a row of information for each sort that an SQL statement performed. Sorts of large result sets in an SQL statement may be a time-consuming operation. In some cases, an index can be created that eliminates the need for a sort.
- **Temporary file information**: Contains a row of information for each SQL statement that required a temporary result. Temporary results are sometimes necessary based on the SQL statement. If the result set inserted into a temporary result is large, you may want to investigate why the temporary result is necessary. In some cases, 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 then sees changes as they are occur in the database tables.
- **Data conversion information**: Contains a row of information for each SQL statement that required data conversion. For example, if a result column has an attribute of INTEGER, but the variable 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.
- **Subquery information**: Contains a row of subquery information. This information can indicate which subquery in a complex SQL statement is the most expensive.
- **Row access information**: Contains information about the rows returned and I/Os performed. This option is 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 may be used to pre-bring some of the data into main memory.
- **Lock escalation information**: Provides information about any lock escalation. This option is 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 that is modifying the table and the reader of the table. A large number of lock escalation entries may indicate a contention performance problem.
- **Bitmap information**: Provides information about any bitmap 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**: 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.
- **Group By information**: Provides information about any Group by operations. This option is available only when you use a detailed SQL Performance Monitor.
- **Error information**: Provides information about any SQL statement error messages and warnings that were captured by the monitor.
- **Start and end monitor information**: Provides information about any start and end Database Monitor operations. This option is available only when you use a detailed SQL Performance Monitor.

The Extended Detailed Results reports provide essentially the same information as these reports, but with added levels of detail.

For a complete definition of the various performance collection tables and a description of each of the table columns, see the *DB2 Universal Database for iSeries - Database Performance and Query Optimization* manual in the iSeries Information Center at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp

When you reach this site, click **Database**. In the right panel, under DB2 manuals, select **Performance and Query Optimization**.

Tip: For a quick of way to determine whether a query was processed by SQE or CQE, you can run the following SQL statement on your performance data:

```
select QQC16 from library.performance_data_table where QQRID = 3014
```

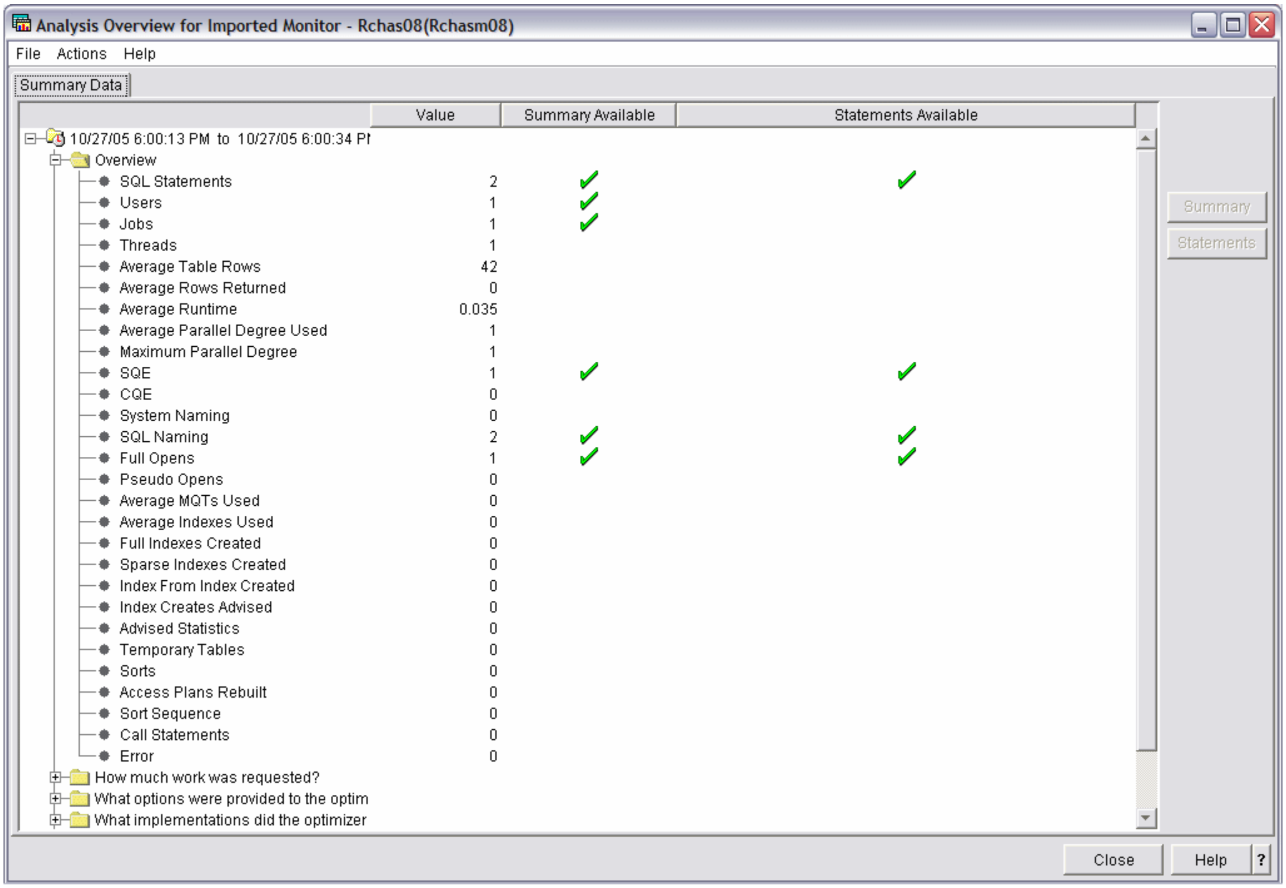
If column QQC16 contains Y, then your query was processed by SQE.

Analyzing SQL performance data on V5R4

The reporting capability of the SQL Performance Monitor in V5R4 has been greatly enhanced with a more flexible and intuitive drill-down reporting tool.

To access the reporting tool, using iSeries Navigator, select the relevant SQL Performance Monitor. Select **File** → **Analyze Results**. The Analysis Overview for Imported Monitor window (Figure 5-24) opens. Note that this is the enhanced interface, called the *dashboard*, for the reports provided in V5R4.

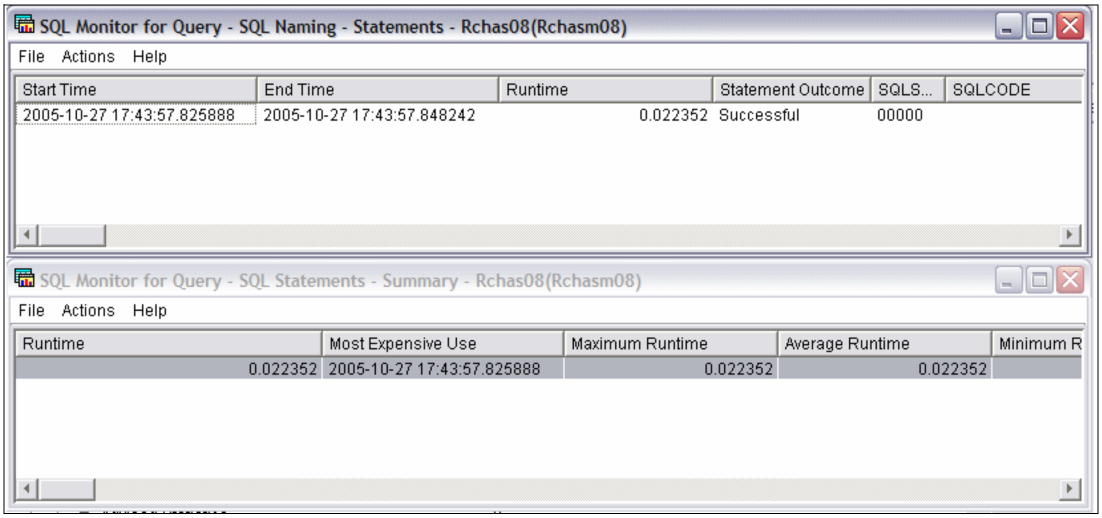
In addition, when a green check mark is displayed under the Summary or Statements column, as shown in Figure 5-24, you can select that row and click the Summary or Statements button to view information about that row type.



	Value	Summary Available	Statements Available
10/27/05 6:00:13 PM to 10/27/05 6:00:34 PI			
Overview			
SQL Statements	2	✓	✓
Users	1	✓	
Jobs	1	✓	
Threads	1		
Average Table Rows	42		
Average Rows Returned	0		
Average Runtime	0.035		
Average Parallel Degree Used	1		
Maximum Parallel Degree	1		
SQE	1	✓	✓
CQE	0		
System Naming	0		
SQL Naming	2	✓	✓
Full Opens	1	✓	✓
Pseudo Opens	0		
Average MQTs Used	0		
Average Indexes Used	0		
Full Indexes Created	0		
Sparse Indexes Created	0		
Index From Index Created	0		
Index Creates Advised	0		
Advised Statistics	0		
Temporary Tables	0		
Sorts	0		
Access Plans Rebuilt	0		
Sort Sequence	0		
Call Statements	0		
Error	0		
How much work was requested?			
What options were provided to the optim			
What implementations did the optimizer			

Figure 5-24 Selecting SQL Performance Monitor reports

The information is displayed in the SQL Monitor for Query - SQL Naming - Statements window (Figure 5-25).


SQL Monitor for Query - SQL Naming - Statements - Rchas08(Rchasm08)
File Actions Help

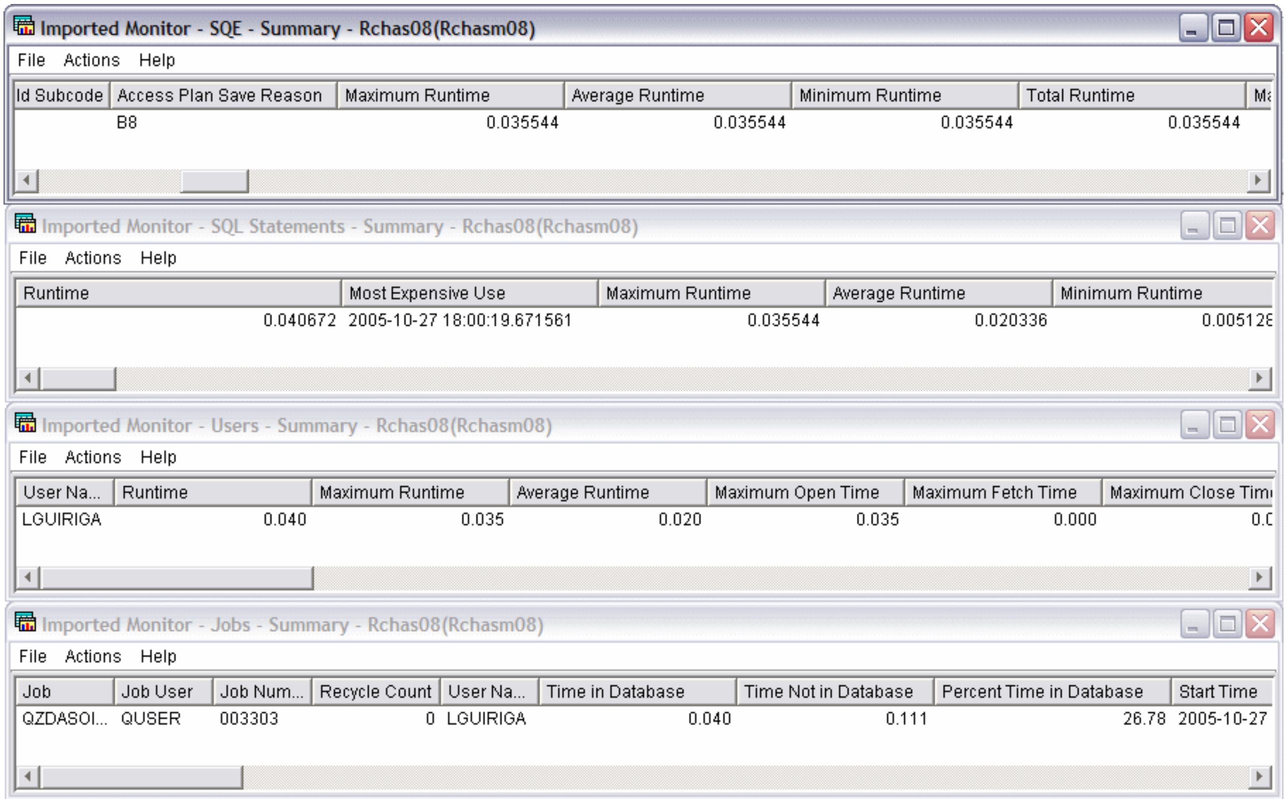
Start Time	End Time	Runtime	Statement Outcome	SQLS...	SQLCODE
2005-10-27 17:43:57.825888	2005-10-27 17:43:57.848242	0.022352	Successful	00000	

SQL Monitor for Query - SQL Statements - Summary - Rchas08(Rchasm08)
File Actions Help

Runtime	Most Expensive Use	Maximum Runtime	Average Runtime	Minimum R
0.022352	2005-10-27 17:43:57.825888	0.022352	0.022352	

Figure 5-25 Statements and Summary window

Figure 5-26 shows additional examples of some of the reports that are currently available. By analyzing the contents of these reports, you can identify problem areas in your queries and take the appropriate steps to remedy such problems.



Imported Monitor - SQE - Summary - Rchas08(Rchasm08)

Id Subcode	Access Plan Save Reason	Maximum Runtime	Average Runtime	Minimum Runtime	Total Runtime	Ma
	B8	0.035544	0.035544	0.035544	0.035544	

Imported Monitor - SQL Statements - Summary - Rchas08(Rchasm08)

Runtime	Most Expensive Use	Maximum Runtime	Average Runtime	Minimum Runtime
0.040672	2005-10-27 18:00:19.671561	0.035544	0.020336	0.00512

Imported Monitor - Users - Summary - Rchas08(Rchasm08)

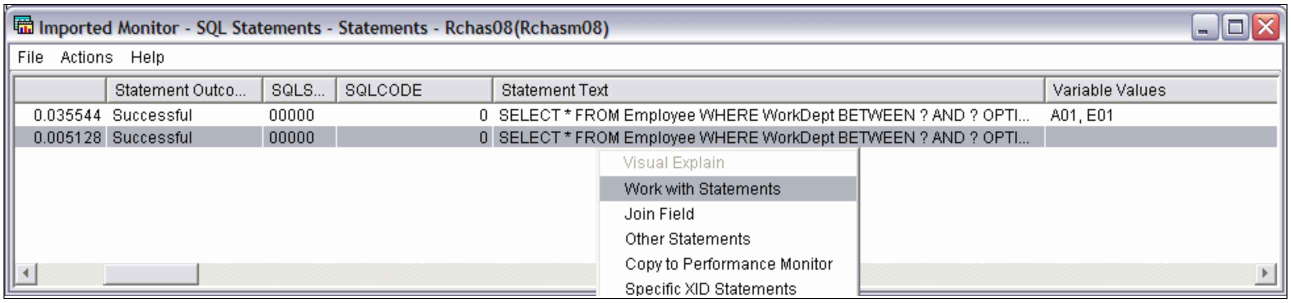
User Na...	Runtime	Maximum Runtime	Average Runtime	Maximum Open Time	Maximum Fetch Time	Maximum Close Tim
LGUIRIGA	0.040	0.035	0.020	0.035	0.000	0.0

Imported Monitor - Jobs - Summary - Rchas08(Rchasm08)

Job	Job User	Job Num...	Recycle Count	User Na...	Time in Database	Time Not in Database	Percent Time in Database	Start Time
QZDASOI...	QUSER	003303	0	LGUIRIGA	0.040	0.111	26.78	2005-10-27

Figure 5-26 SQL Performance Monitor predefined reports

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. To retrieve the SQL statements, select the report that you need. Then select the statement, right-click it, and select **Work with Statements** (Figure 5-27). The SQL statements are retrieved and displayed in the iSeries Navigator Run SQL Scripts window, from where you can execute it, modify it, and use Visual Explain to explain it.



Imported Monitor - SQL Statements - Statements - Rchas08(Rchasm08)

	Statement Outco...	SQLS...	SQLCODE	Statement Text	Variable Values
0.035544	Successful	00000	0	SELECT * FROM Employee WHERE WorkDept BETWEEN ? AND ? OPTI...	A01, E01
0.005128	Successful	00000	0	SELECT * FROM Employee WHERE WorkDept BETWEEN ? AND ? OPTI...	

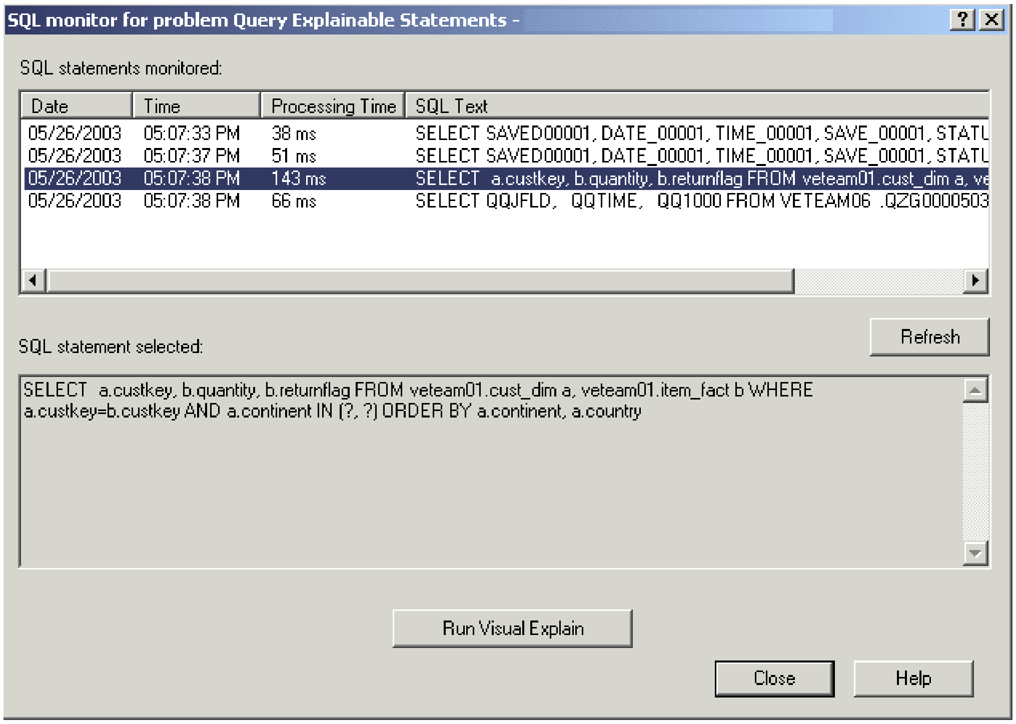
Figure 5-27 Working with statements

Running Visual Explain on the SQL Performance Monitor data

From an SQL Performance Monitor, it is useful to select an SQL statement and to explain it using Visual Explain. You can do this in OS/400 V5R2 and V5R3, as well as in i5/OS V5R4. The difference is that, in V5R4, the interface was changed. We explain both interfaces in this section.

From the SQL Performance Monitors status panel (Figure 5-18 page 105), select a monitor row and right-click it. In V5R2 and V5R3, there is an option called *List Explainable Statements*, and in V5R4, it is called *Show Statements*.

In V5R2 and V5R3, you are prompted with the SQL monitor for problem Query Explainable Statement window (Figure 5-28).

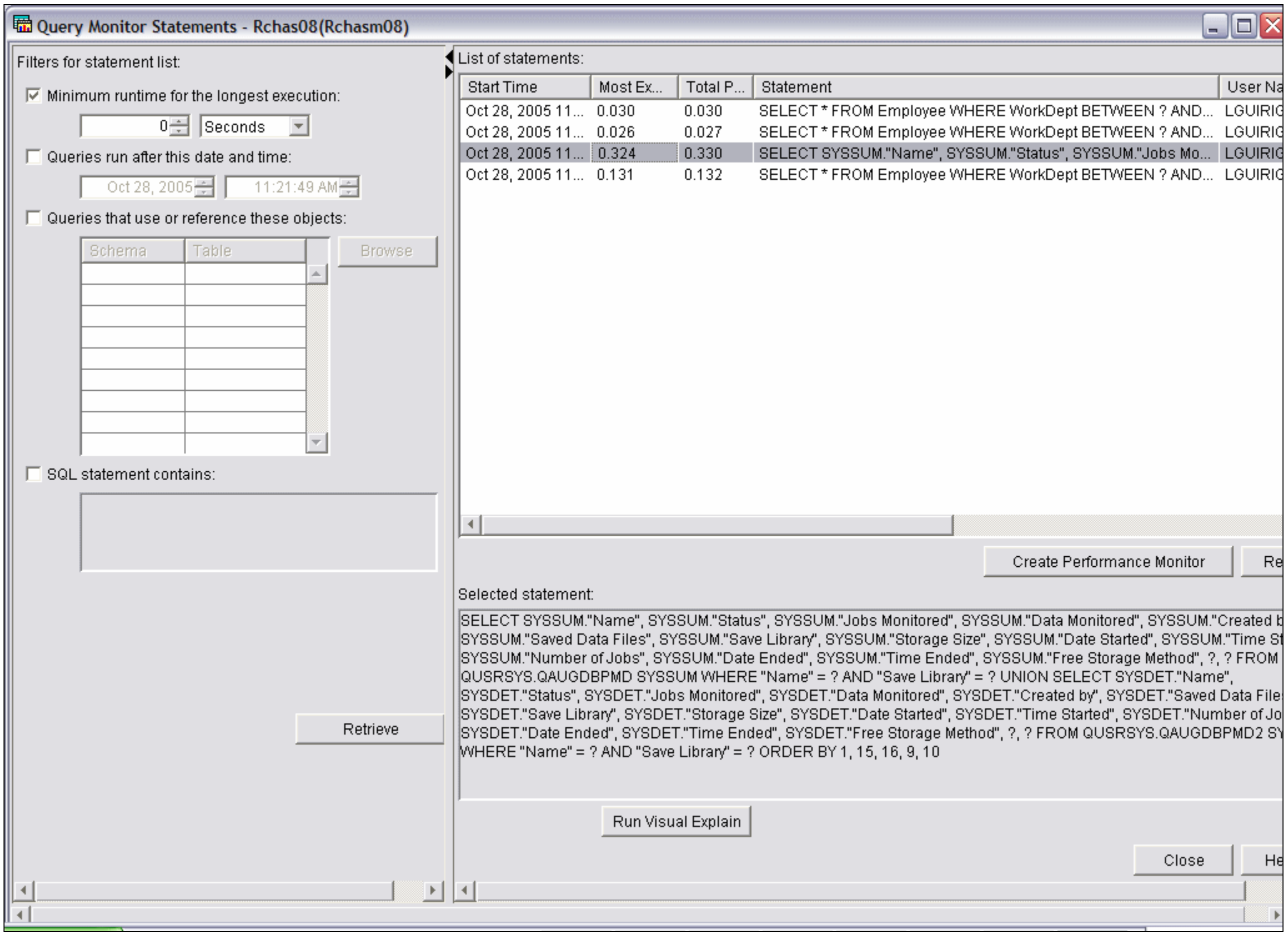

SQL monitor for problem Query Explainable Statements -
?
SQL statements monitored:

Date	Time	Processing Time	SQL Text
05/26/2003	05:07:33 PM	38 ms	SELECT SAVED00001, DATE_00001, TIME_00001, SAVE_00001, STATU
05/26/2003	05:07:37 PM	51 ms	SELECT SAVED00001, DATE_00001, TIME_00001, SAVE_00001, STATU
05/26/2003	05:07:38 PM	143 ms	SELECT a.custkey, b.quantity, b.returnflag FROM veteam01.cust_dim a, ve
05/26/2003	05:07:38 PM	66 ms	SELECT QQJFLD, QQTIME, QQ1000 FROM VETEAM06 .QZG0000503

```
SELECT  a.custkey, b.quantity, b.returnflag FROM veteam01.cust_dim a, veteam01.item_fact b WHERE
a.custkey=b.custkey AND a.continent IN (?, ?) ORDER BY a.continent, a.country
```

Figure 5-28 List Explainable Statements in V5R2 and V5R3

In V5R4, you are prompted with the Query Monitor Statements window (Figure 5-29). From this window, you can select an SQL statement and click the **Run Visual Explain** button. This takes you directly to the Visual Explain panel (Figure 5-30). Also, you can run a separate monitor for the statement with performance problems.



Query Monitor Statements - Rchas08(Rchasm08)

Filters for statement list:

☑ Minimum runtime for the longest execution: 0 Seconds

☐ Queries run after this date and time: Oct 28, 2005 11:21:49 AM

☐ Queries that use or reference these objects:

Schema	Table

☐ SQL statement contains:

List of statements:

Start Time	Most Ex...	Total P...	Statement	User Na
Oct 28, 2005 11...	0.030	0.030	SELECT * FROM Employee WHERE WorkDept BETWEEN ? AND...	LGUIRIG
Oct 28, 2005 11...	0.026	0.027	SELECT * FROM Employee WHERE WorkDept BETWEEN ? AND...	LGUIRIG
Oct 28, 2005 11...	0.324	0.330	SELECT SYSSUM."Name", SYSSUM."Status", SYSSUM."Jobs Mo...	LGUIRIG
Oct 28, 2005 11...	0.131	0.132	SELECT * FROM Employee WHERE WorkDept BETWEEN ? AND...	LGUIRIG

Selected statement:

```
SELECT SYSSUM."Name", SYSSUM."Status", SYSSUM."Jobs Monitored", SYSSUM."Data Monitored", SYSSUM."Created b
SYSSUM."Saved Data Files", SYSSUM."Save Library", SYSSUM."Storage Size", SYSSUM."Date Started", SYSSUM."Time St
SYSSUM."Number of Jobs", SYSSUM."Date Ended", SYSSUM."Time Ended", SYSSUM."Free Storage Method", ?, ? FROM
QUSRSYS.QAUGDBPMD SYSSUM WHERE "Name" = ? AND "Save Library" = ? UNION SELECT SYSDET."Name",
SYSDET."Status", SYSDET."Jobs Monitored", SYSDET."Data Monitored", SYSDET."Created by", SYSDET."Saved Data File
SYSDET."Save Library", SYSDET."Storage Size", SYSDET."Date Started", SYSDET."Time Started", SYSDET."Number of Jo
SYSDET."Date Ended", SYSDET."Time Ended", SYSDET."Free Storage Method", ?, ? FROM QUSRSYS.QAUGDBPMD2 SY
WHERE "Name" = ? AND "Save Library" = ? ORDER BY 1, 15, 16, 9, 10
```

Figure 5-29 Statements from a monitor in V5R4

5.1.4 Visual Explain

One of the easiest ways to gather information about the execution of an SQL statement is through the Visual Explain interface. Visual Explain is a component of iSeries Navigator. It provides a full-color graphical representation of how the optimizer implemented your query. It presents a tree-based diagram that shows which tables and indexes were used in the query, how they were joined (where applicable), and the data flow through the query process until the final selection or output step. In addition to the graphical overview, it provides detailed information, presented in separate window panes, that describes how the optimizer implemented the query.

Another added advantage of Visual Explain is its ability to explain the SQL statement, with associated detailed information, through a simulation process, without actually running the statement. This is especially useful for analyzing queries and simulating changes in your production environment. You can select this feature after you invoke the SQL Script Center by selecting **Visual Explain** → **Explain** on the menu bar.

These features make Visual Explain a great tool for architecting and modeling new queries, as well as for analyzing and tuning existing queries. For more information about Visual

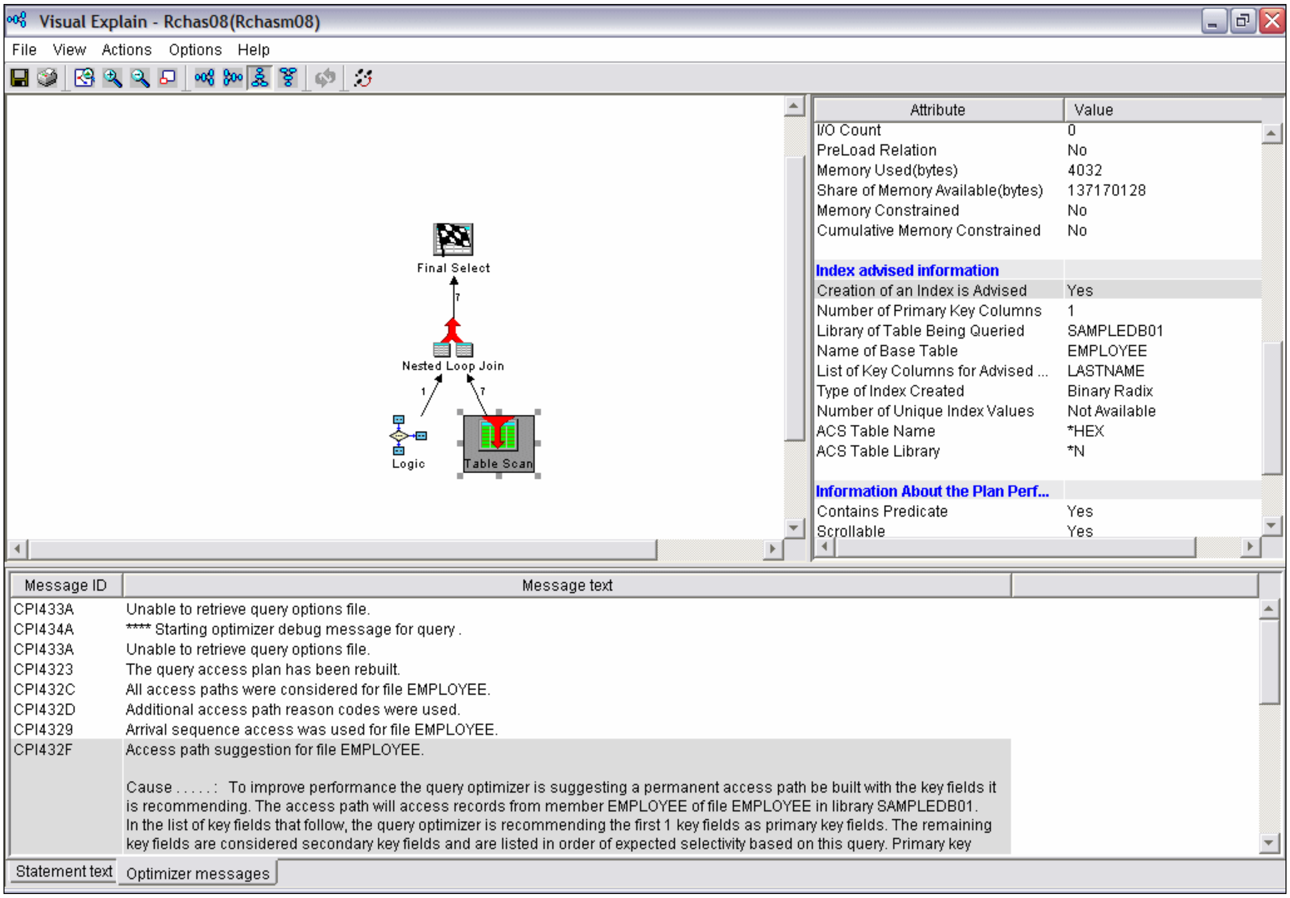
Explain, refer to the IBM Redbook *Advanced Functions and Administration on DB2 Universal Database for iSeries*, SG24-4249. Also refer to the IBM Redbook *SQL Performance Diagnosis on IBM DB2 Universal Database for iSeries*, SG24-6654.

Figure 5-30 shows an example of the information presented by Visual Explain when selecting either **Visual Explain** → **Explain** or **Visual Explain** → **Run and Explain**. In the examples that follow, we used the following SQL statement:

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

Notice the optimizer messages displayed in the bottom pane of the window. These are similar to the debug messages shown in Figure 5-4 page 93. If you double-click any of these messages, you receive additional second-level text messages and extended detailed information as shown in Figure 5-30.

The Visual Explain interface also details information about the tables and indexes used in your query. When you select a table or index icon on the left hand graphical layout, Visual Explain presents you with detailed information about the object in the right pane of the window. As you can see in Figure 5-30, details are displayed that pertain to the table size, the number of rows in the table, the number of rows selected from the table, and suggestions for creating indexes. You can also get an indication of whether your query primarily used more central processing unit (CPU) resources or I/O resources. These are all factors that you should consider when devising a strategy to make your query more efficient.



Attribute	Value
I/O Count	0
PreLoad Relation	No
Memory Used(bytes)	4032
Share of Memory Available(bytes)	137170128
Memory Constrained	No
Cumulative Memory Constrained	No
Index advised information	
Creation of an Index is Advised	Yes
Number of Primary Key Columns	1
Library of Table Being Queried	SAMPLEDB01
Name of Base Table	EMPLOYEE
List of Key Columns for Advised ...	LASTNAME
Type of Index Created	Binary Radix
Number of Unique Index Values	Not Available
ACS Table Name	*HEX
ACS Table Library	*N
Information About the Plan Perf...	
Contains Predicate	Yes
Scrollable	Yes

Message ID	Message text
CPI433A	Unable to retrieve query options file.
CPI434A	**** Starting optimizer debug message for query .
CPI433A	Unable to retrieve query options file.
CPI4323	The query access plan has been rebuilt.
CPI432C	All access paths were considered for file EMPLOYEE.
CPI432D	Additional access path reason codes were used.
CPI4329	Arrival sequence access was used for file EMPLOYEE.
CPI432F	Access path suggestion for file EMPLOYEE. 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 EMPLOYEE of file EMPLOYEE in library SAMPLEDB01. 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

Figure 5-30 Visual Explain debug messages

You can see examples of using the Index and Statistics Advisor functions of Visual Explain in “Using the Statistics and Index Advisor” on page 137.

Visual Explain can assist you in understanding how your query is running. It can also help you determine the changes that you can make to improve the query’s performance.

5.1.5 Index Advised: System wide (V5R4)

In V5R4, a new feature is available that is capable of showing, at the system wide, schema or table level, all the indexes that have been advised by the optimizer. You can display the index advised information from the optimizer by using iSeries Navigator. This information is based on and located in the QSYS2/SYSIXADV system table.

To review the Index Advisor, select **Database** → **Index Advisor** → **Index Advisor** in iSeries Navigator as shown in Figure 5-31.

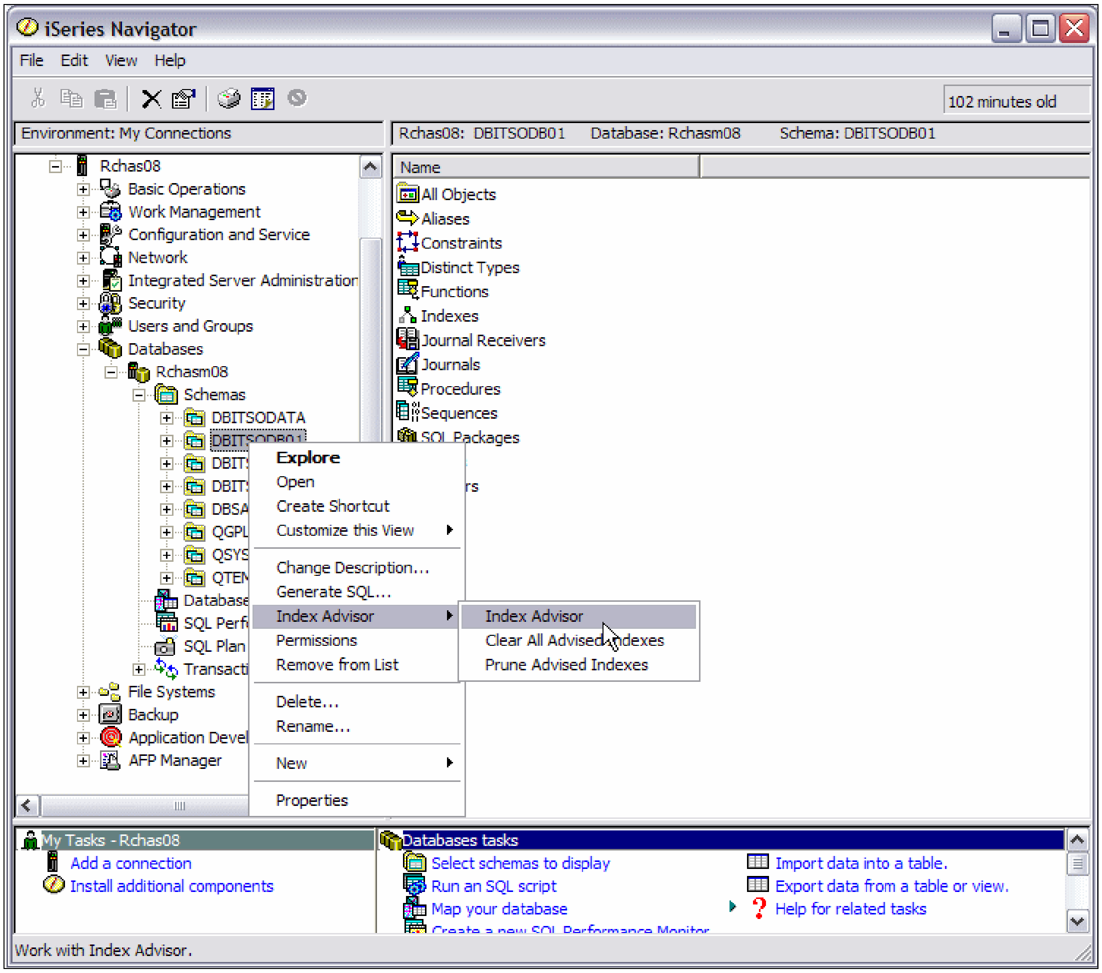
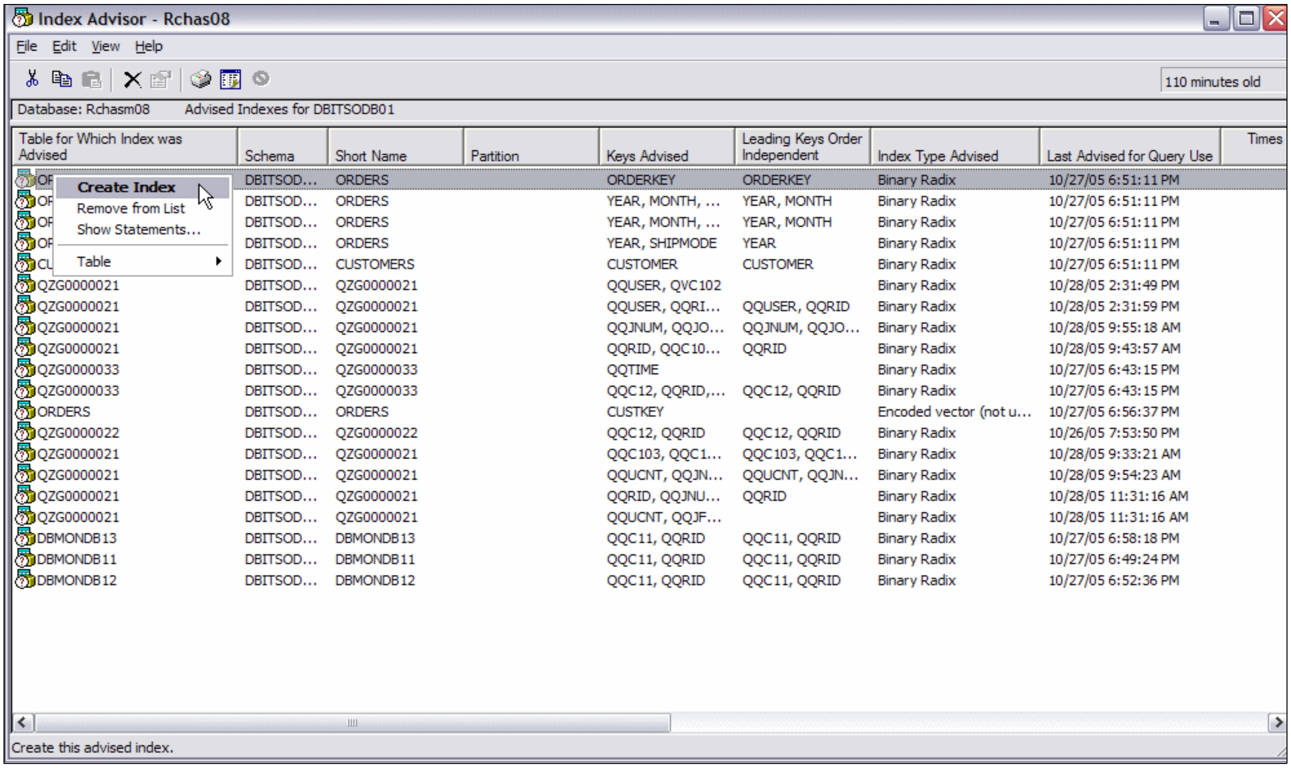


Figure 5-31 Index Advisor

You can use the Index Advisor to manage the creation of recommended indexes that the optimizer suggests to improve the performance of your query.

In the Index Advisor window, you can see the list of indexes recommended by the optimizer. The resulting row is sequenced upon the table for which index was advised, but you can sort the information by the column that you want and click the column name, as shown in Figure 5-32.



Database: Rchasm08 Advised Indexes for DBITSODB01

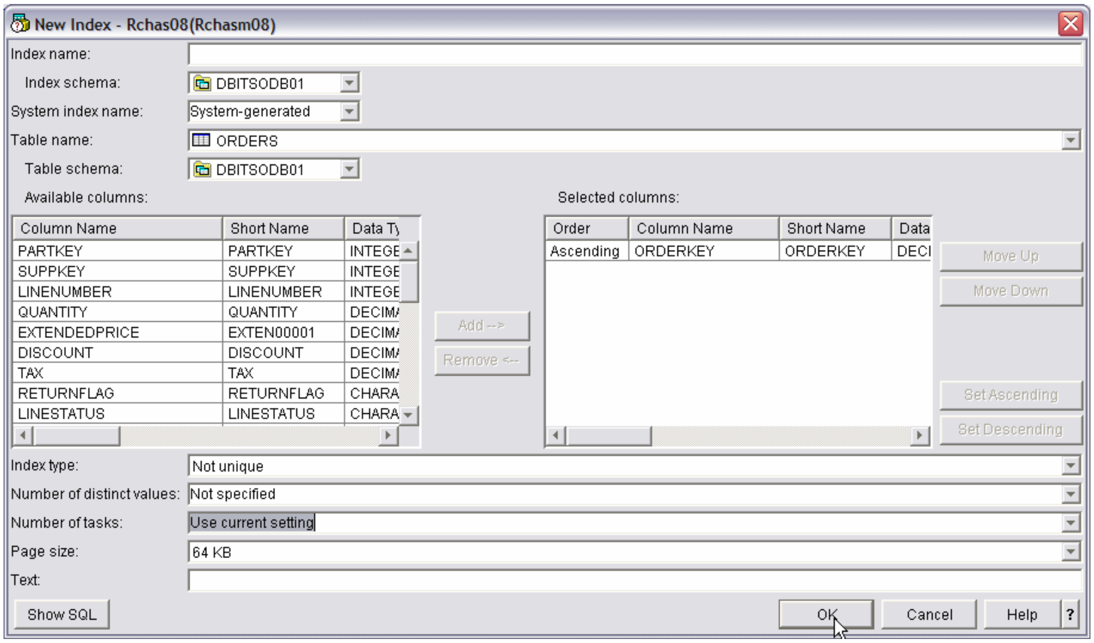
Table for Which Index was Advised	Schema	Short Name	Partition	Keys Advised	Leading Keys Order Independent	Index Type Advised	Last Advised for Query Use	Times
OF	DBITSOD...	ORDERS		ORDERKEY	ORDERKEY	Binary Radix	10/27/05 6:51:11 PM	
OF	DBITSOD...	ORDERS		YEAR, MONTH, ...	YEAR, MONTH	Binary Radix	10/27/05 6:51:11 PM	
OF	DBITSOD...	ORDERS		YEAR, MONTH, ...	YEAR, MONTH	Binary Radix	10/27/05 6:51:11 PM	
OF	DBITSOD...	ORDERS		YEAR, SHIPMODE	YEAR	Binary Radix	10/27/05 6:51:11 PM	
CU	DBITSOD...	CUSTOMERS		CUSTOMER	CUSTOMER	Binary Radix	10/27/05 6:51:11 PM	
QZG0000021	DBITSOD...	QZG0000021		QQUSER, QVC102		Binary Radix	10/28/05 2:31:49 PM	
QZG0000021	DBITSOD...	QZG0000021		QQUSER, QQRI...	QQUSER, QQRID	Binary Radix	10/28/05 2:31:59 PM	
QZG0000021	DBITSOD...	QZG0000021		QQJNUM, QQJO...	QQJNUM, QQJO...	Binary Radix	10/28/05 9:55:18 AM	
QZG0000021	DBITSOD...	QZG0000021		QQRID, QQC10...	QQRID	Binary Radix	10/28/05 9:43:57 AM	
QZG0000033	DBITSOD...	QZG0000033		QQTIME		Binary Radix	10/27/05 6:43:15 PM	
QZG0000033	DBITSOD...	QZG0000033		QQC12, QQRID,...	QQC12, QQRID	Binary Radix	10/27/05 6:43:15 PM	
ORDERS	DBITSOD...	ORDERS		CUSTKEY		Encoded vector (not u...	10/27/05 6:56:37 PM	
QZG0000022	DBITSOD...	QZG0000022		QQC12, QQRID	QQC12, QQRID	Binary Radix	10/26/05 7:53:50 PM	
QZG0000021	DBITSOD...	QZG0000021		QQC103, QQC1...	QQC103, QQC1...	Binary Radix	10/28/05 9:33:21 AM	
QZG0000021	DBITSOD...	QZG0000021		QQUCNT, QQJN...	QQUCNT, QQJN...	Binary Radix	10/28/05 9:54:23 AM	
QZG0000021	DBITSOD...	QZG0000021		QQRID, QQJNU...	QQRID	Binary Radix	10/28/05 11:31:16 AM	
QZG0000021	DBITSOD...	QZG0000021		QQUCNT, QQJF...		Binary Radix	10/28/05 11:31:16 AM	
DBMONDB13	DBITSOD...	DBMONDB13		QQC11, QQRID	QQC11, QQRID	Binary Radix	10/27/05 6:58:18 PM	
DBMONDB11	DBITSOD...	DBMONDB11		QQC11, QQRID	QQC11, QQRID	Binary Radix	10/27/05 6:49:24 PM	
DBMONDB12	DBITSOD...	DBMONDB12		QQC11, QQRID	QQC11, QQRID	Binary Radix	10/27/05 6:52:36 PM	

Figure 5-32 Index Advisor window

Figure 5-33 shows the New Index window from which you can immediately create the suggested index. Notice the new parameter *Number of tasks* for specifying to create the index in parallel. The possible values are:

- Use current setting
- No parallel processing
- Managed by the optimizer
- Maximized by optimizer
- Use system value
- Integer value

Note: You must have the system feature DB2 Symmetric Multiprocessing installed to specify the number of tasks.



New Index - Rchas08(Rchasm08)

Index name:

Index schema: DBITSODB01

System index name: System-generated

Table name: ORDERS

Table schema: DBITSODB01

Available columns:

Column Name	Short Name	Data Ty
PARTKEY	PARTKEY	INTEGE
SUPPKEY	SUPPKEY	INTEGE
LINENUMBER	LINENUMBER	INTEGE
QUANTITY	QUANTITY	DECIMA
EXTENDEDPRICE	EXTEN00001	DECIMA
DISCOUNT	DISCOUNT	DECIMA
TAX	TAX	DECIMA
RETURNFLAG	RETURNFLAG	CHARA
LINESTATUS	LINESTATUS	CHARA

Selected columns:

Order	Column Name	Short Name	Data
Ascending	ORDERKEY	ORDERKEY	DECI

Index type: Not unique

Number of distinct values: Not specified

Number of tasks: Use current setting

Page size: 64 KB

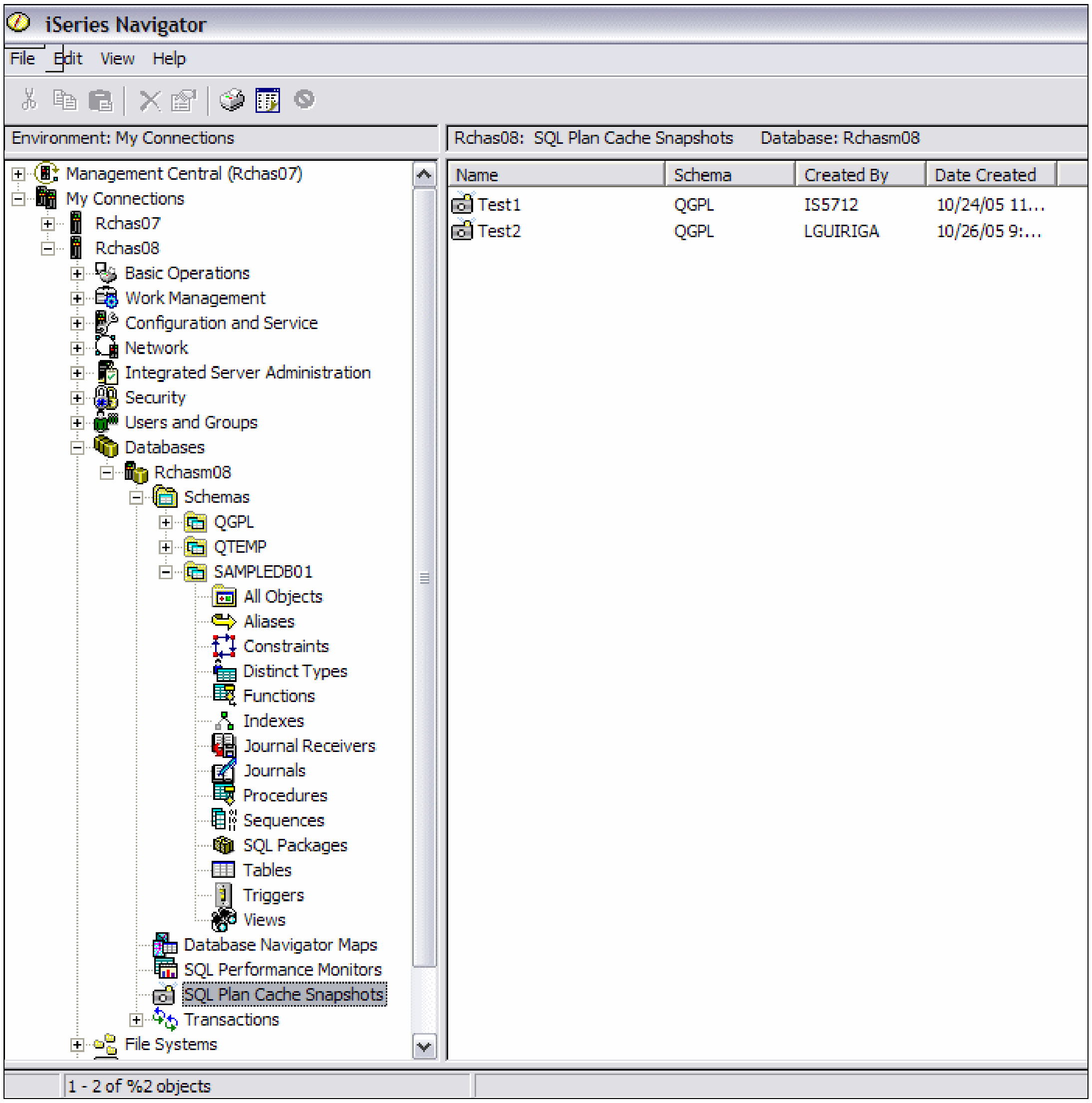
Text:

Figure 5-33 Creating a new index from the Index Advisor

5.1.6 Viewing the plan cache (V5R4)

The plan cache contains a wealth of information about the SQE SQL statements that are run through the database. Its contents are viewable through iSeries Navigator.

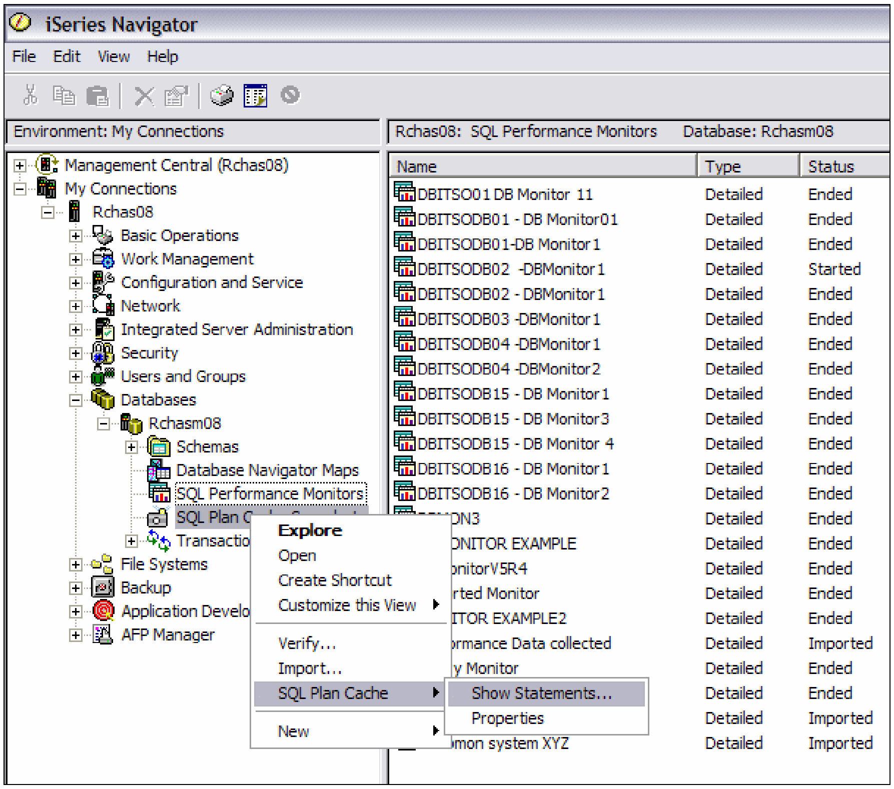
The plan cache interface provides a window into the database query operations on the system. In iSeries Navigator, you can find it by selecting ***system name*** → **Databases** → ***database name*** → **SQL Plan Cache Snapshots** as shown in Figure 5-34.



Name	Schema	Created By	Date Created
Test1	QGPL	IS5712	10/24/05 11:...
Test2	QGPL	LGUIRIGA	10/26/05 9:...

Figure 5-34 Viewing the plan cache using iSeries Navigator V5R4

When you right-click the SQL Plan Cache Snapshot, you can choose from a series of options to allow different views of the current plan cache of the database. In the example shown in Figure 5-35, we select **SQL Plan Cache → Show Statements**.



Name	Type	Status
DBITSO01 DB Monitor 11	Detailed	Ended
DBITSODB01 - DB Monitor01	Detailed	Ended
DBITSODB01-DB Monitor1	Detailed	Ended
DBITSODB02 -DBMonitor1	Detailed	Started
DBITSODB02 - DBMonitor1	Detailed	Ended
DBITSODB03 -DBMonitor1	Detailed	Ended
DBITSODB04 -DBMonitor1	Detailed	Ended
DBITSODB04 -DBMonitor2	Detailed	Ended
DBITSODB15 - DB Monitor1	Detailed	Ended
DBITSODB15 - DB Monitor3	Detailed	Ended
DBITSODB15 - DB Monitor 4	Detailed	Ended
DBITSODB16 - DB Monitor1	Detailed	Ended
DBITSODB16 - DB Monitor2	Detailed	Ended
ON3	Detailed	Ended
ONITOR EXAMPLE	Detailed	Ended
onitorV5R4	Detailed	Ended
rted Monitor	Detailed	Ended
ITOR EXAMPLE2	Detailed	Ended
rmance Data collected	Detailed	Imported
y Monitor	Detailed	Ended
	Detailed	Ended
	Detailed	Imported
mon system XYZ	Detailed	Imported

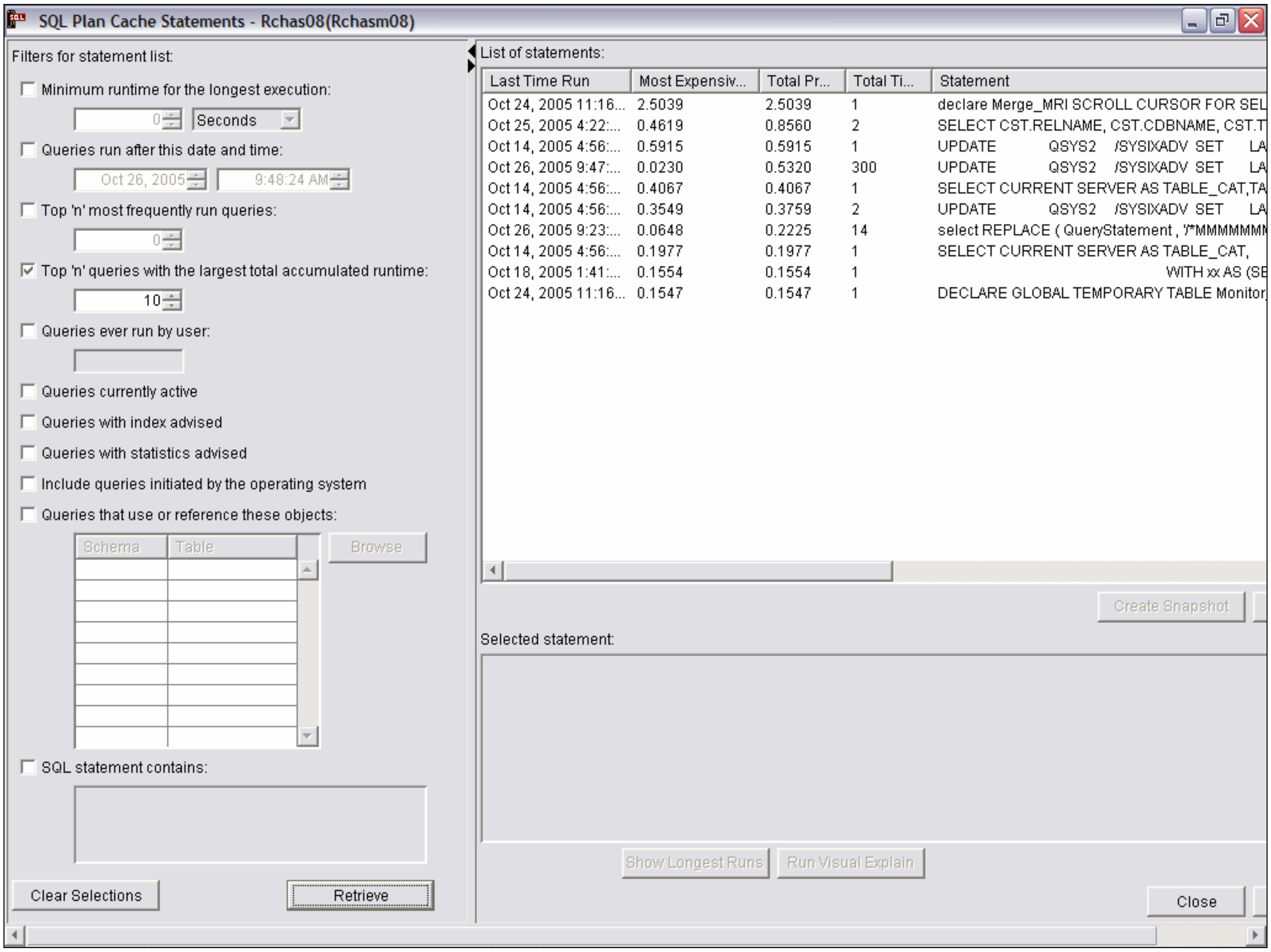
Figure 5-35 Viewing the plan cache

The SQL Plan Cache Statements window (Figure 5-36) opens. This window provides a direct view of the current plan cache on the system and filtering capability. It also provides filtering options that allow the user to more quickly isolate specific criteria of interest. No filters are required to be specified (the default), although adding filtering shortens the time it takes to show the results.

The list of queries that is returned is ordered by default so that those who are consuming the most processing time are shown at the top. You can reorder the results by clicking the column heading for which you want the list ordered. Repeated clicking toggles the order from ascending to descending.

When you choose an individual entry, you can see more detailed information about that entry. Show Longest Runs shows details of up to ten of the longest running instances of that query. Run Visual Explain can also be performed against the chosen query to show the details of the query plan. Finally, if one or more entries are highlighted, a snapshot (database performance monitor file) for those selected entries can be generated.

Important: You must perform the retrieve action (push) to fill the display.



Filters for statement list:

- ☐ Minimum runtime for the longest execution: 0 Seconds
- ☐ Queries run after this date and time: Oct 26, 2005 9:48:24 AM
- ☐ Top 'n' most frequently run queries: 0
- ☑ Top 'n' queries with the largest total accumulated runtime: 10
- ☐ Queries ever run by user:
- ☐ Queries currently active
- ☐ Queries with index advised
- ☐ Queries with statistics advised
- ☐ Include queries initiated by the operating system
- ☐ Queries that use or reference these objects: (Schema | Table)
- ☐ SQL statement contains:

List of statements:

Last Time Run	Most Expensiv...	Total Pr...	Total Ti...	Statement
Oct 24, 2005 11:16...	2.5039	2.5039	1	declare Merge_MRI SCROLL CURSOR FOR SEL
Oct 25, 2005 4:22:...	0.4619	0.8560	2	SELECT CST.RELNAME, CST.CDBNAME, CST.T
Oct 14, 2005 4:56:...	0.5915	0.5915	1	UPDATE QSYS2 /SYSIXADV SET LA
Oct 26, 2005 9:47:...	0.0230	0.5320	300	UPDATE QSYS2 /SYSIXADV SET LA
Oct 14, 2005 4:56:...	0.4067	0.4067	1	SELECT CURRENT SERVER AS TABLE_CAT,TA
Oct 14, 2005 4:56:...	0.3549	0.3759	2	UPDATE QSYS2 /SYSIXADV SET LA
Oct 26, 2005 9:23:...	0.0648	0.2225	14	select REPLACE (QueryStatement , '/*MMMMMMM
Oct 14, 2005 4:56:...	0.1977	0.1977	1	SELECT CURRENT SERVER AS TABLE_CAT,
Oct 18, 2005 1:41:...	0.1554	0.1554	1	WITH xx AS (SE
Oct 24, 2005 11:16...	0.1547	0.1547	1	DECLARE GLOBAL TEMPORARY TABLE Monitor

Selected statement:

Figure 5-36 Plan cache

The information in Figure 5-36 shows the SQL query text, the last time the query was run, the most expensive single instance run of the query, total processing time consumed by the query, total number of times the query has been run, and information about the user and job that first created the plan entry. It also shows the number of times (if any) that the database engine was able to reuse the results of a prior run of the query to avoid rerunning the entire query.

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

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

The filtering options provide a way to focus on a particular area of interest. The following filters can be specified:

- Minimum runtime for the longest execution

  This option filters those queries with at least one long individual query instance runtime.

- Queries run after this date and time

  This option filters those queries that have been run recently

- Top ‘n’ most frequently run queries

  This option finds those queries that are run most often.

- Top ‘n’ queries with the largest total accumulated runtime

  This option shows the top resource consumers. This equates to the first *n* entries shown by default when no filtering is given. Specifying a value for *n* improves the performance of getting the first screen of entries, although the total entries displayed is limited to *n*.

- Queries ever run by user

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

- Queries currently active

  This option shows the list of cached entries associated with queries that are still running or are in pseudo close mode. As with user filtering, the user and job name information shown in the resulting entries still reflects the originator of the cached entry, which is not necessarily the same as the user currently running the query. Multiple users may be running the query.

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

- Queries with index advised

  This option limits the list to those queries where an index was advised by the optimizer to improve performance.

- Queries with statistics advised

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

- Include queries initiated by the operating system

  This option includes the “hidden” queries that were initiated by the database itself behind the scenes to process a request. By default, the list only includes user initiated queries.

- Queries that use or reference these objects

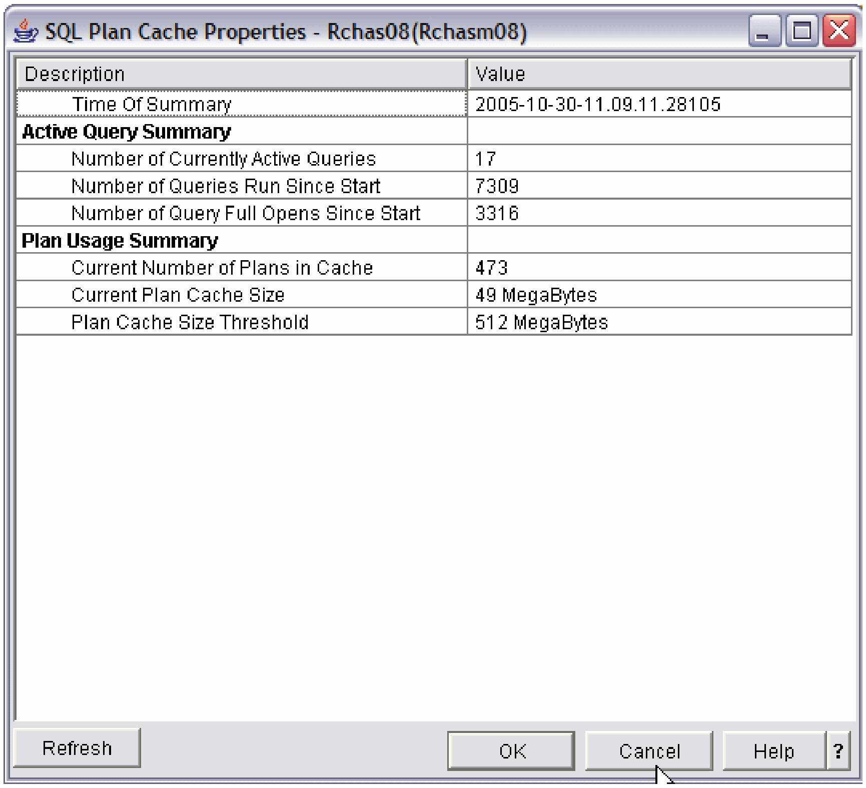
  This option provides a way to limit the entries to those that referenced or use the table or tables (or indexes) that are specified.

- SQL statement contains

  This option provides a wildcard search capability on the SQL text itself. It is useful for finding particular types of queries. For example, queries with a FETCH FIRST clause can be found by specifying “fetch”. The search is case insensitive for ease of use. For example, the string FETCH finds the same entries as the search string “fetch”.

Multiple filter options can be specified. Note that in a multifilter case, the candidate entries for each filter are computed independently, and only those entries that are present in all the candidate lists are shown. For example, if you specified the *Top ‘n’ most frequently run queries* and *Queries ever run by user* options, you are shown those most run entries in the cache that ran at some point by the specified user. You are not necessarily shown the most frequently run queries run by the user, unless those queries also happen to be the most frequently run queries in the entire cache.

You can also see the properties of the plan cache. Right-click **SQL Plan Cache Snapshot** and select **SQL Plan Cache** → **Properties**. Then you see high level information about the cache, such as the cache size, number of active queries, number of plans, and number of full opens (see Figure 5-37).



SQL Plan Cache Properties - Rchas08(Rchasm08)

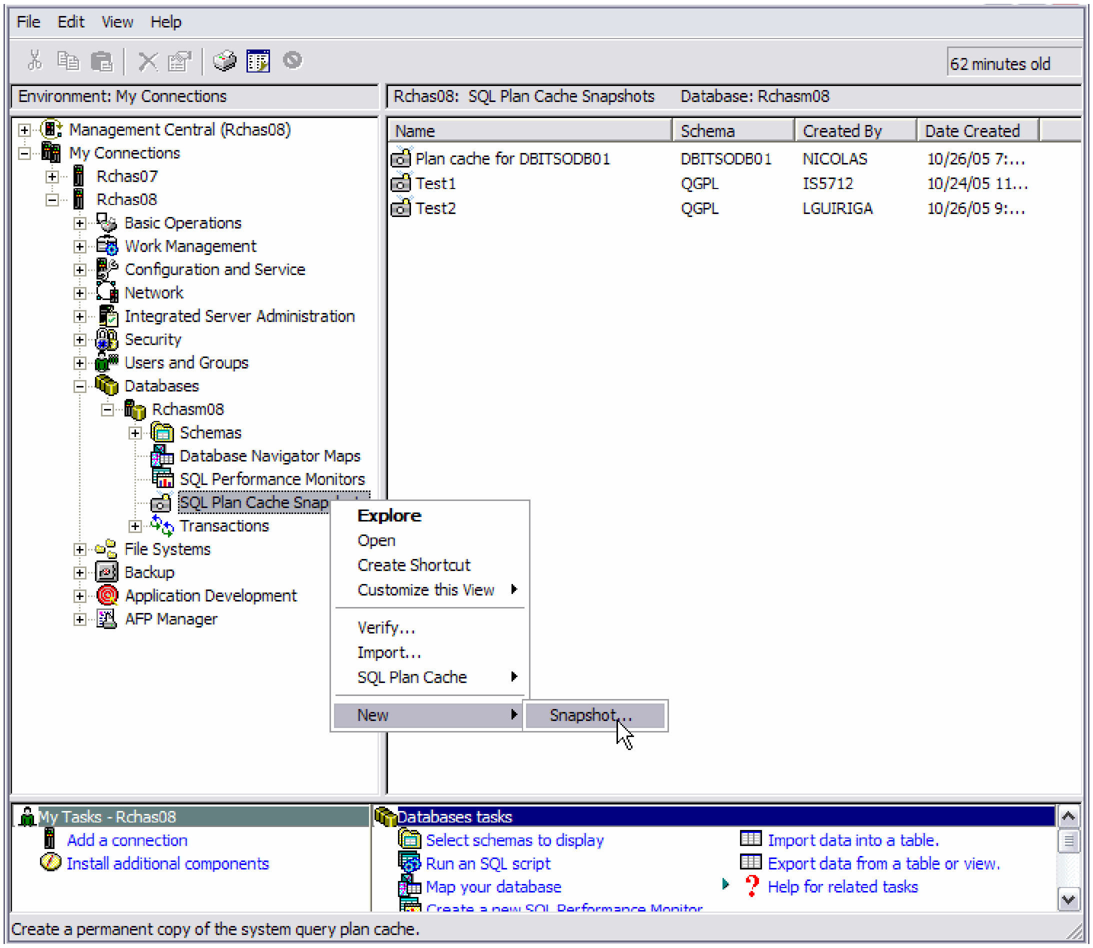
Description	Value
Time Of Summary	2005-10-30-11.09.11.28105
Active Query Summary	
Number of Currently Active Queries	17
Number of Queries Run Since Start	7309
Number of Query Full Opens Since Start	3316
Plan Usage Summary	
Current Number of Plans in Cache	473
Current Plan Cache Size	49 MegaBytes
Plan Cache Size Threshold	512 MegaBytes

Figure 5-37 SQL Plan Cache Properties

You can use this information to view overall database activity. If tracked over time, it provides trends to help you better understand the database utilization peaks and valleys throughout the day and week.

5.1.7 Plan Cache Snapshot

It is important to keep in mind that the plan cache is cleared by an initial program load (IPL). For this reason, it is important to have a way to dump or store the contents of the plan cache into a permanent or persistent object. This can be done by creating a Plan Cache Snapshot. To create a snapshot in iSeries Navigator, you select **SQL Plan Cache** → **New** → **Snapshot** as shown in Figure 5-38.

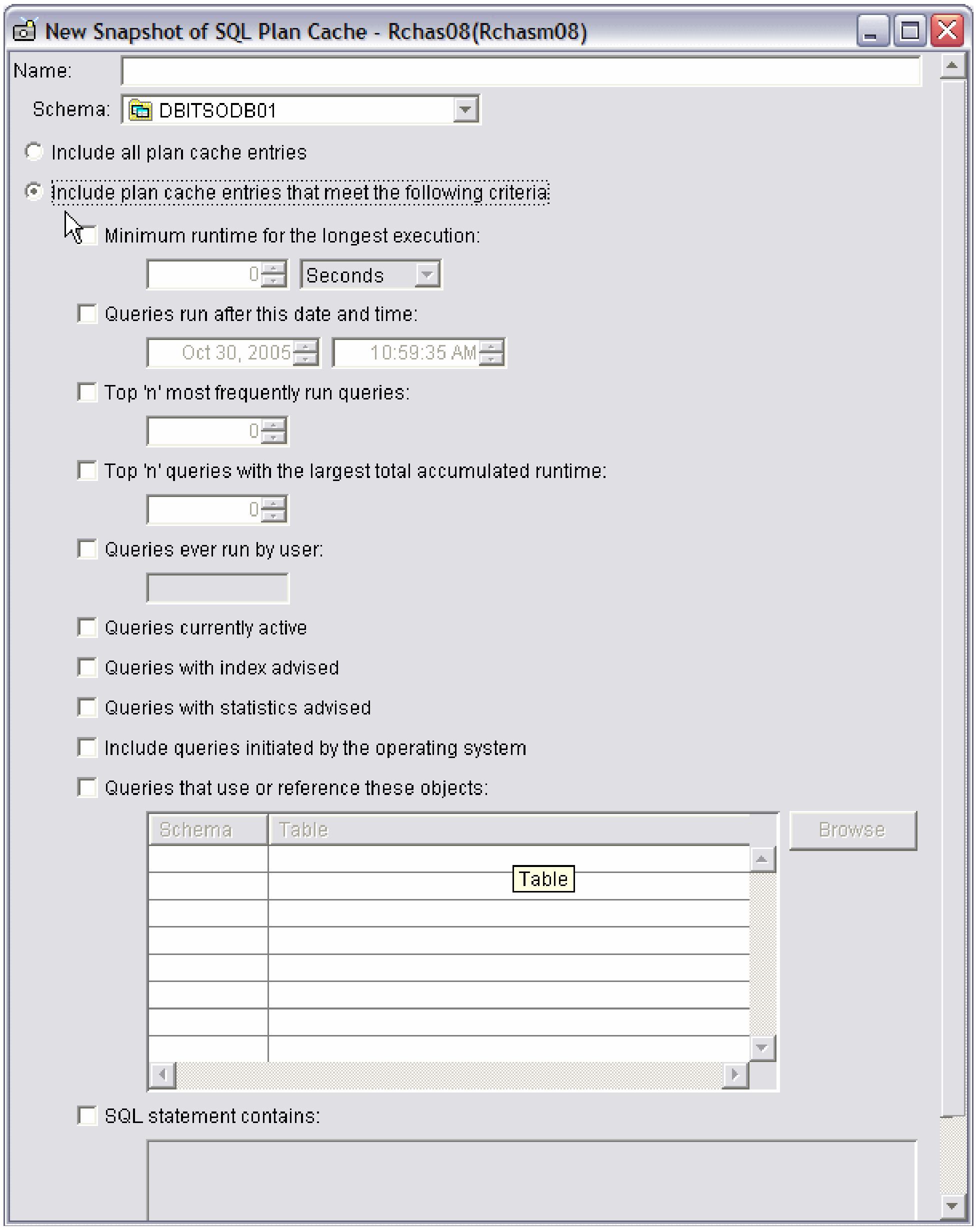


Name	Schema	Created By	Date Created
Plan cache for DBITSODB01	DBITSODB01	NICOLAS	10/26/05 7:...
Test1	QGPL	IS5712	10/24/05 11...
Test2	QGPL	LGUIRIGA	10/26/05 9:...

Create a permanent copy of the system query plan cache.

Figure 5-38 Creating a snapshot from SQL Plan Cache

When you create a snapshot from the plan cache, you can choose some filtering capabilities as shown in Figure 5-39.



New Snapshot of SQL Plan Cache - Rchas08(Rchasm08)

Name:

Schema: DBITSODB01

○ Include all plan cache entries

● Include plan cache entries that meet the following criteria

□ Minimum runtime for the longest execution:

0 Seconds

□ Queries run after this date and time:

Oct 30, 2005 10:59:35 AM

□ Top 'n' most frequently run queries:

0

□ Top 'n' queries with the largest total accumulated runtime:

0

□ Queries ever run by user:

□ Queries currently active

□ Queries with index advised

□ Queries with statistics advised

□ Include queries initiated by the operating system

□ Queries that use or reference these objects:

Schema	Table

□ SQL statement contains:

Figure 5-39 New Snapshot of SQL Plan Cache

The stored procedure, qsys2.dump_plan_cache, provides the simplest way to create a database monitor file output (snapshot) from the plan cache. The dump_plan_cache procedure takes two parameters, library name and file name, to identify the resulting database monitor file. If the file does not exist, it is created. For example, to dump the plan cache to a database performance monitor file in library QGPL, you enter:

```
CALL qsys2.dump_plan_cache('QGPL','SNAPSHOT1');
```

The plan cache is an actively changing cache. Therefore, it is important to realize that it contains timely information. If information over long periods of time is of interest, consider implementing a method of performing periodic snapshots of the plan cache to capture trends and heavy usage periods. You can use the APIs described previously, in conjunction with job scheduling (for example), to programmatically perform periodic snapshots.

5.1.8 Query options table (QAQQINI)

The QAQQINI query options table provides a facility to dynamically modify or override some of the system environment values that were used when you ran your queries. Every System i installation contains a template QAQQINI table that is located in the QSYS library. We recommend that you *do not* modify this default QAQQINI table. Instead, create a copy of this table in library QUSRSYS. Then you can modify this table to define system-wide default values for queries without a unique QAQQINI table assigned.

To create a QAQQINI table in QUSRSYS or in other user libraries (for additional variations on the query environment), use the Create Duplicate Object (CRTDUPOBJ) CL command or select **Edit** → **Copy/Paste** using iSeries Navigator. This duplicates the triggers that are in place to handle changes to the QAQQINI table to maintain its integrity. The following example shows the CL command to create a copy of the QAQQINI table:

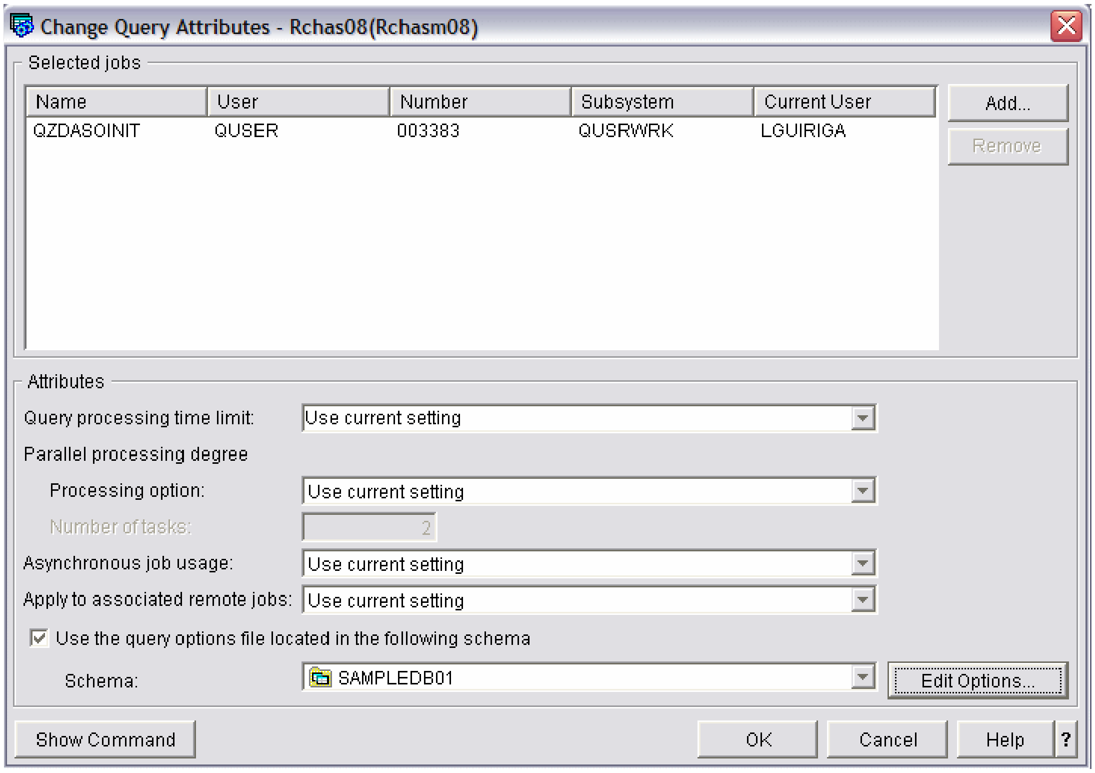
```
CRTDUPOBJ OBJ(QAQQINI) FROMLIB(QSYS) OBJTYPE(*FILE) TOLIB(MYLIB) DATA(*YES)
```

After you create your own copy of the QAQQINI table, you can direct your queries to use this table by changing your query runtime attributes.

To change your query runtime attributes, in the Run SQL Scripts window in iSeries Navigator, select **Options** → **Change Query Attributes**. Next, specify the name of the library that contains the QAQQINI table that you want to use for running your query, as shown in Figure 5-40.

Important: The default QSYS/QAQQINI table is used by certain system processes. Changing this table may affect these processes. Also objects in library QSYS may automatically be replaced when applying IBM PTFs or when upgrading your operating system. This can result in user changes to the QSYS/QAQQINI table being reset to the IBM-supplied default values.

You can set various parameters in the QAQQINI table that allow you to monitor the performance of your queries. When you click the Edit Options button (Figure 5-40), you can gain access to these parameters.



Change Query Attributes - Rchas08(Rchasm08)

Selected jobs

Name	User	Number	Subsystem	Current User
QZDASOINIT	QUSER	003383	QUSRWRK	LGUIRIGA

Attributes

Query processing time limit: Use current setting

Parallel processing degree

Processing option: Use current setting

Number of tasks: 2

Asynchronous job usage: Use current setting

Apply to associated remote jobs: Use current setting

☑ Use the query options file located in the following schema

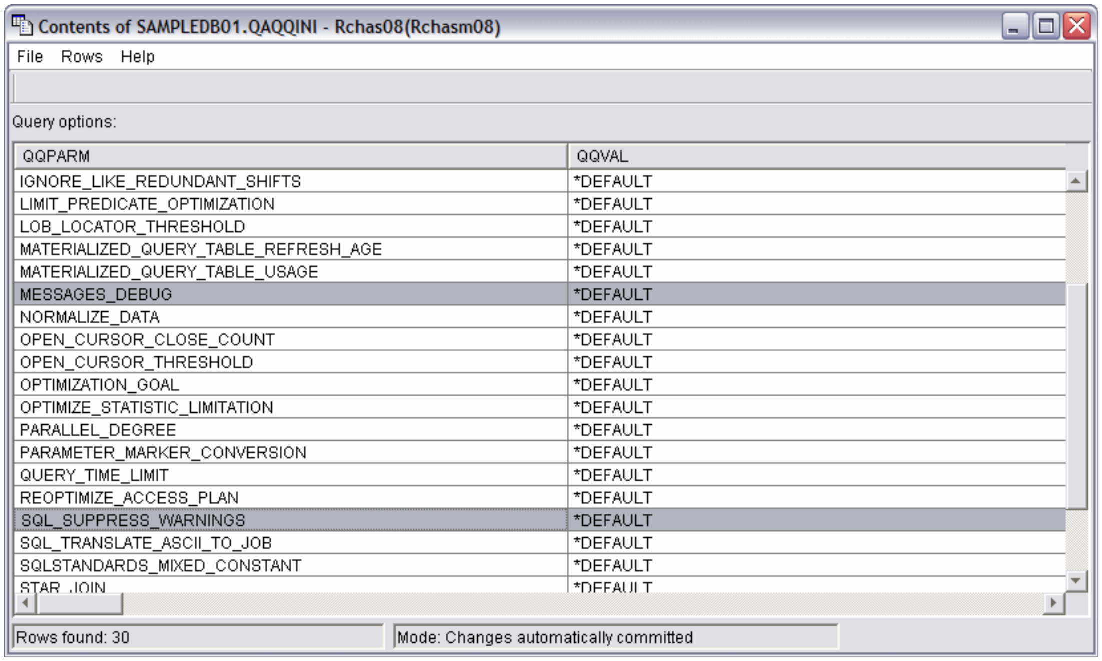
Schema: SAMPLEDB01

Figure 5-40 Specifying the location of the QAQQINI table

Figure 5-41 shows an example of some of these parameters in the QAQQINI table. You can control the amount of detailed information that you want to see about your query execution by setting the appropriate parameter values (QQVAL), for example:

- The MESSAGES_DEBUG parameter controls whether debug messages are written into your job log.
- The SQL_SUPPRESS_WARNINGS parameter is used to determine if you want to see SQL warnings when executing your queries.

> **Tip:** You can also direct your SQL query to use a specific QAQQINI table or temporarily change certain QAQQINI parameters by including the relevant CL command in your SQL script as shown in the following examples:
>
> ```
> CL: chgqrya degree(*optimize)
> CL: chgqrya qryoptlib(library_name)
> ```



Query options:

QQPARM	QQVAL
IGNORE_LIKE_REDUNDANT_SHIFTS	*DEFAULT
LIMIT_PREDICATE_OPTIMIZATION	*DEFAULT
LOB_LOCATOR_THRESHOLD	*DEFAULT
MATERIALIZED_QUERY_TABLE_REFRESH_AGE	*DEFAULT
MATERIALIZED_QUERY_TABLE_USAGE	*DEFAULT
MESSAGES_DEBUG	*DEFAULT
NORMALIZE_DATA	*DEFAULT
OPEN_CURSOR_CLOSE_COUNT	*DEFAULT
OPEN_CURSOR_THRESHOLD	*DEFAULT
OPTIMIZATION_GOAL	*DEFAULT
OPTIMIZE_STATISTIC_LIMITATION	*DEFAULT
PARALLEL_DEGREE	*DEFAULT
PARAMETER_MARKER_CONVERSION	*DEFAULT
QUERY_TIME_LIMIT	*DEFAULT
REOPTIMIZE_ACCESS_PLAN	*DEFAULT
SQL_SUPPRESS_WARNINGS	*DEFAULT
SQL_TRANSLATE_ASCII_TO_JOB	*DEFAULT
SQLSTANDARDS_MIXED_CONSTANT	*DEFAULT
STAR_JOIN	*DEFAULT

Rows found: 30

Mode: Changes automatically committed

Figure 5-41 QAQQINI parameters for monitoring query performance

For more information about valid parameter settings for the QAQQINI table, see Informational APAR II13320. Also refer to (search on) “Creating the QAQQINI query options file” in the iSeries Information Center at the following address:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp

5.2 Performance tuning

The fundamental basis of performance tuning has not changed with the introduction of SQE. Performance tuning can be summarized as:

- Ensuring that sufficient system resources are available for the anticipated query environment
  - CPU power (speed) and number of CPUs
  - Main storage
  - Disk arms
- Paying attention to the indexing strategy for the application environment
  - Proactive creation of indexes to meet known application requirements
  - Reactive creation of indexes in response to advice given by the optimizer after the application is in test or production

SQE introduces additional considerations, which are discussed in the following sections.

5.2.1 System environment

There are several considerations for system configuration in the SQE environment.

Processors

Processing capacity is still important to ensure that complex queries are processed in a timely and efficient manner. For the complex queries that do not use SQE starting in OS/400 V5R2, the CPU power needed by CQE-handled queries is similar to V5R1. For SQE, some queries see an improvement in processor utilization.

Multiple processors (iSeries n-way support) are an important consideration. When SQE uses SQL non-indexed access methods starting in V5R2, multiple processors are a factor because SQE can run all or part of an access plan tree in parallel. Since SQE uses threads instead of tasks for splitting up a job, it uses fewer tasks. In addition to SQE using multiple processors for executing the threads within an access plan, SMP can also be used by the statistics engine to collect statistics for SQE.

Note: Changing the number of CPUs (whole or fractions using logical partition (LPAR)) that are available to a query results in rebuilding the access plan.

Main storage

SQE uses a slightly smaller main storage footprint for a query compared with CQE. This reduction is not significant enough to encourage smaller configurations to be deployed.

However, there is a change in the way that the optimizer calculates a *fair share* of memory for a query. This is the maximum amount of memory that can be allocated to a query without having a detrimental effect on other jobs running in the same memory pool.

CQE continues to compute a job’s fair share of memory by dividing the memory pool size by the maximum activity level of that pool. SQE, in contrast, tries to compute the fair share of memory calculation by dividing by the average number of active jobs instead of the maximum activity level. SQE can make this computation only when the associated memory pool is defined with a pool paging option of *CALC (also known as *expert cache*). If the pool paging option is not *CALC, then it too divides by the maximum activity level. Because of the way SQE calculates the fair share of memory, memory-constrained configurations may degrade performance more severely than may be the case with CQE.

The degree of parallelism is controlled by the DEGREE parameter of the CHGQRYA command or by the PARALLEL_DEGREE value within the query attribute table (QAQQINI) in use. This value controls parallelism as follows:

- *OPTIMIZE
  - Use the fair share of memory as calculated previously.
  - CPU bound queries use Degree up to the number of processors available.
- *MAX
  - Use all memory available in the pool.
  - CPU bound queries use Degree up to twice the number of processors available.
- *NBRTASKS
  - Use the fair share of memory as calculated previously.
  - Force the Degree specified in DEGREE value:
    - CQE considers this as a maximum value for parallelism.
    - SQE forces the use of this value for parallelism.

SQE Optimizer responds differently to changes in the memory pool size in terms of rebuilding an access plan. CQE Optimizer rebuilds an access plan if there is a two-fold change in size. SQE looks for a 10-fold change. While this seems like a dramatic change, remember that the SQE computes the job's fair share of memory differently from CQE.

Disk storage

We continually emphasize the need for a balanced system by ensuring that sufficient disk arms are configured. SQE performs asynchronous I/O far more aggressively than CQE and fully uses parallel pre-fetching of data. Therefore, any under configuration of the number of disk drives on a system is accentuated by queries that use SQE, particularly those that access a large number of table rows.

5.2.2 Indexing basics

DB2 for i5/OS has two kinds of persistent indexes. Binary-radix tree indexes became available when the AS/400 systems (earlier model of the System i family) began shipping in 1988. Encoded-vector indexes (EVIs) became available in 1998 with Version 4 Release 3. Both types of indexes are useful in improving performance for certain kinds of queries.

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. These nodes 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.

The binary-radix tree structure (Figure 5-42) 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 finding 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 it needs and perform a minimal number of I/Os.

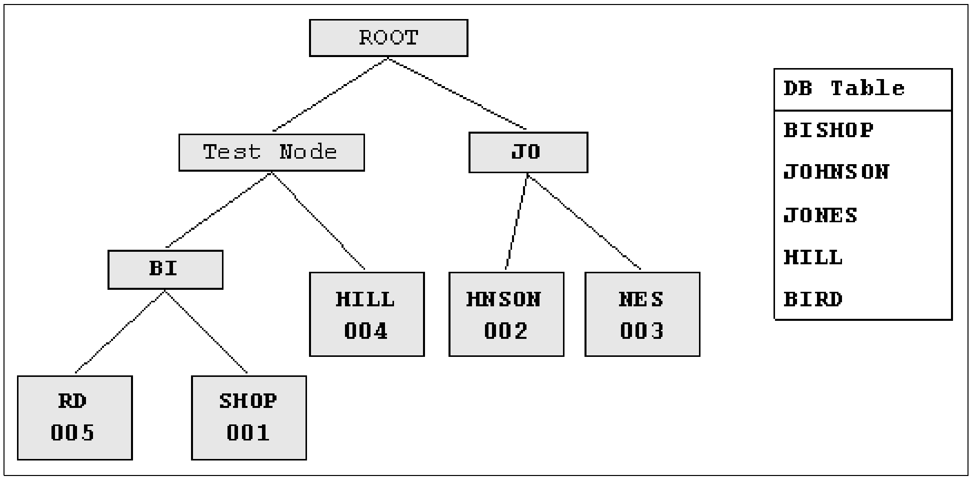


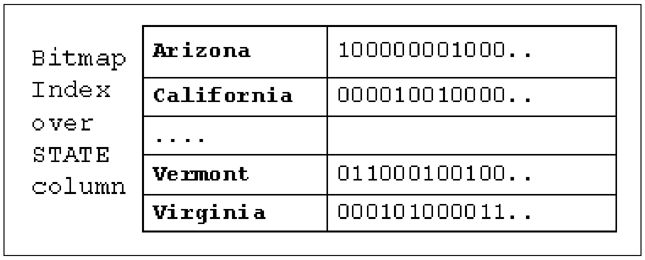
Figure 5-42 Binary-radix tree index structure

In Business Intelligence environments, database analysts do not always have the same level of predictability. Increasingly, users want ad hoc access to the detail data underlying their data marts. For example, they may run a report every week to look at sales data and then drill down for more information related to a particular problem area they found in the report. In this scenario, database analysts cannot write all the queries in advance on behalf of users. Without knowing what queries will be run, it is impossible to build the perfect index.

Encoded-vector indexes

To understand EVIs, you should have a basic knowledge of bitmap indexing. DB2 for i5/OS does not create permanent bitmaps. SQL creates dynamic bitmaps temporarily for query optimization.

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. See Figure 5-43.


Bitmap Index over STATE column

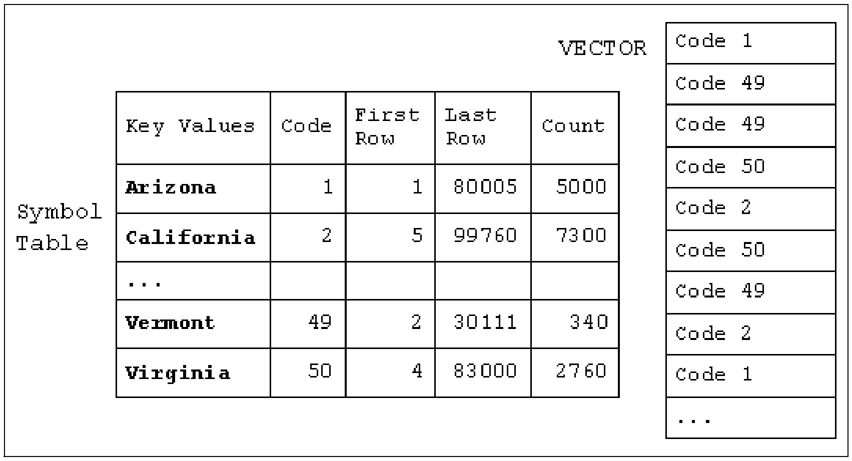
Arizona	100000001000..
California	000010010000..
....	
Vermont	011000100100..
Virginia	000101000011..

Figure 5-43 Bitmap index structure

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 ungainly size. For example, in a one billion row table, you may 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 some sort of compression algorithm to help alleviate this growth problem.

In addition, maintenance of very large bitmap indexes can be problematic. Every time the database is updated, the system must update each bitmap. This is a potentially tedious process, for example, if there are 1,000 unique values in a one billion row table. When adding a new distinct key value, an entire bitmap must be generated. These issues usually result in the database being used as “read only”.

EVI is a new solution developed by IBM Research to support dynamic bitmap indexing. It is a data structure that is stored as basically two components: the symbol table and vector (Figure 5-44).



Key Values	Code	First Row	Last Row	Count
Arizona	1	1	80005	5000
California	2	5	99760	7300
...				
Vermont	49	2	30111	340
Virginia	50	4	83000	2760

VECTOR
Code 1
Code 49
Code 49
Code 50
Code 2
Code 50
Code 49
Code 2
Code 1
...

Figure 5-44 Two components of EVI

The symbol table contains a distinct key list, along with statistical and descriptive information about each distinct key value in the index. The symbol 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 byte code.

The vector contains a byte code value for each row in the table. This byte code represents the actual key value found in the symbol table and the respective row in the database table. The byte codes 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.

The optimizer can use the symbol table to obtain statistical information about the data and key values represented in the EVI. If the optimizer decides to use an EVI to process the local selection of the query, the database engine uses the vector to build a dynamic bitmap, which contains one bit for each row in the table. The bits in the bitmap are in the same ordinal position as the row it represents in the table. If the row satisfies the query, the bit is set *on*. If the row does not satisfy the query, the bit is set *off*. The database engine can also derive a list of relative record numbers (RRNs) from the EVI. These RRNs represent the rows that match the selection criteria, without needing a bitmap.

Like a traditional bitmap index, the DB2 Universal Database dynamic bitmaps or RRN lists can be ANDed and ORed together to satisfy an ad hoc query. For example, if a user wants to see sales data for a certain region during a certain time period, the database analyst can define an EVI over the Region column and the Quarter column of the database. When the query runs, the database engine builds dynamic bitmaps using the two EVIs. Then it uses ANDing on both bitmaps to produce a bitmap that represents all the local selection (bits turned on for only the relevant rows). This ANDing capability effectively uses more than one index to drastically reduce the number of rows that the database engine must retrieve and process.

Note: In OS/400 V5R2, changes were made to the algorithms to handle the Symbol Table overflow area for values added since the last rebuild. These changes resulted in significant performance improvements for EVI maintenance, particularly for EVIs based on columns with a high number of additions and changes.

Indexes and the optimizer

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

The CQE Optimizer attempts to examine most, if not all, indexes built over a table unless or until it times out. However, the SQE Optimizer only considers those indexes returned to it by the Statistics Manager. With OS/400 V5R2, these include only indexes that the Statistics Manager considers useful in performing local selection, based on the “where” clause predicates. Consequently the SQE Optimizer does not time out.

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

5.2.3 Indexing and statistics strategy

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

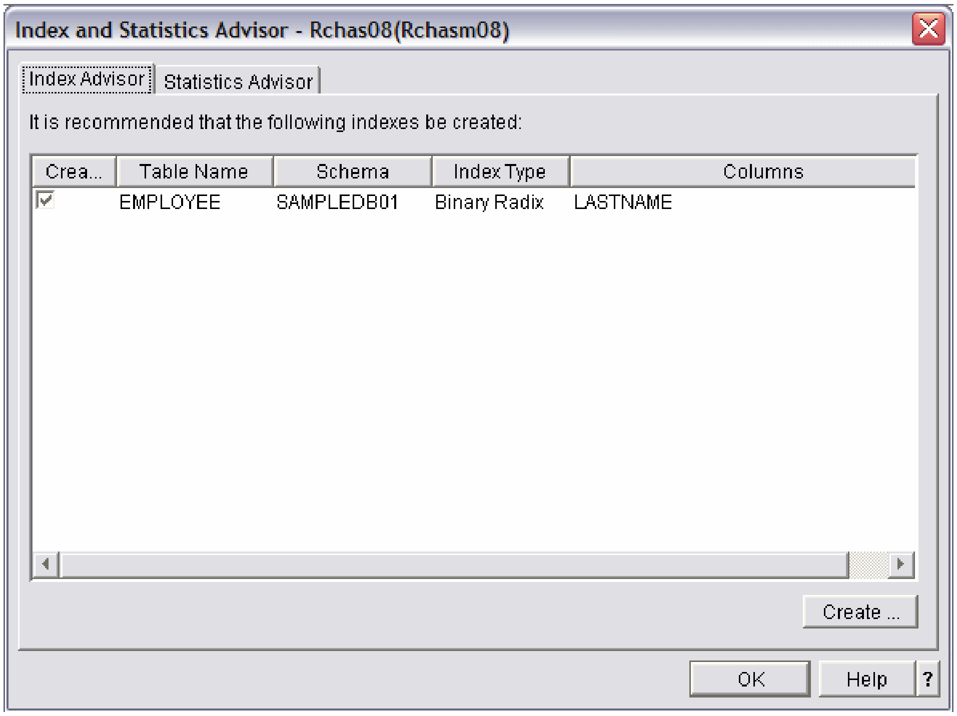
Using the Statistics and Index Advisor

If you are using Visual Explain (see 5.1.4, “Visual Explain” on page 118), you can use the Index and Statistics Advisor as follows to manage the creation of recommended indexes and column statistics.

Consider the SQL statement and Visual Explain chart (Figure 5-30 page 119) from 5.1.4, “Visual Explain” on page 118. When you select the **Actions** → **Advisor**, Visual Explain presents all the optimizer suggestions for creating indexes and column statistics to improve the performance of your query.

Index Advisor

Figure 5-45 shows the Index Advisor page of the Visual Explain Statistics and Index Advisor window. In this case, the optimizer suggests that this query will benefit from a binary-radix index on the EMPLOYEE table, with the LASTNAME column as the key. You can create the index directly from here by clicking the Create button.

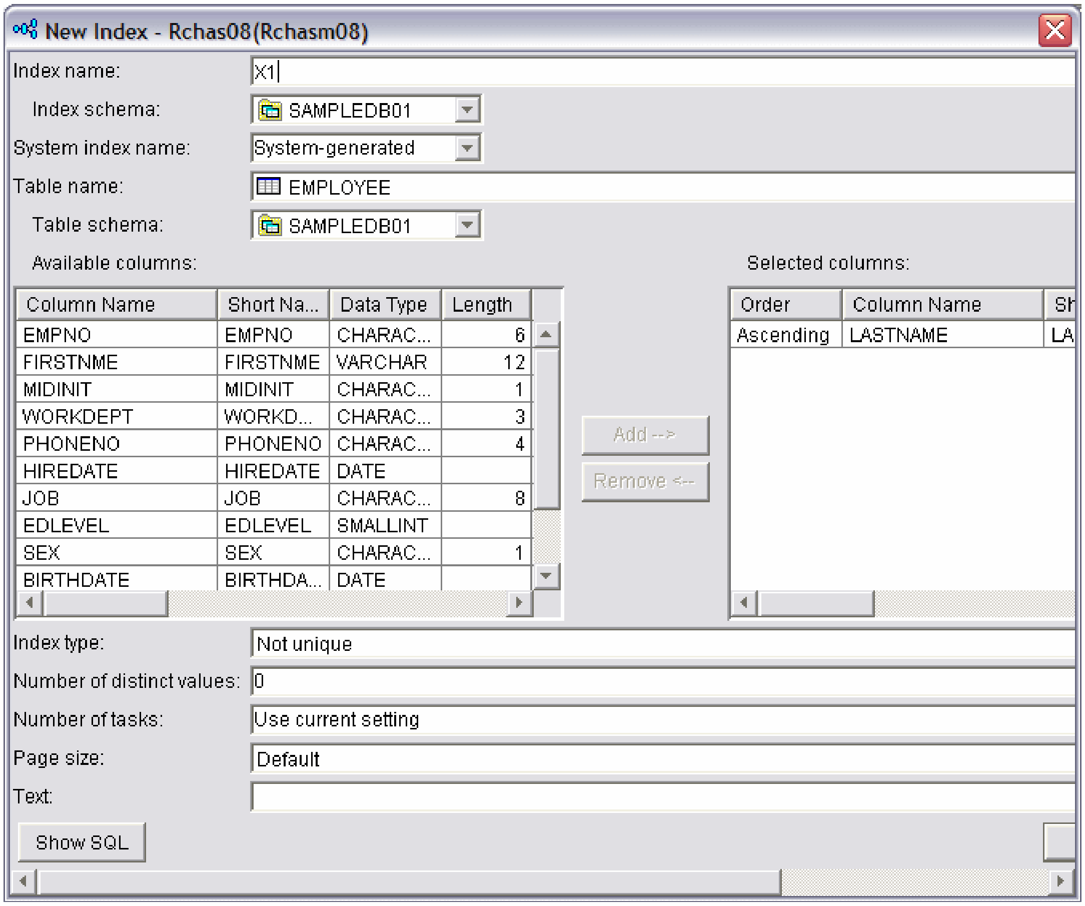


It is recommended that the following indexes be created:

Crea...	Table Name	Schema	Index Type	Columns
☑	EMPLOYEE	SAMPLEDB01	Binary Radix	LASTNAME

Figure 5-45 Visual Explain Index Advisor page

Figure 5-46 shows the New Index window, from which you can immediately create the suggested index. Notice that the suggested key column and ordering sequence is already filled in for you. You simply specify a name and location for your index.



New Index - Rchas08(Rchasm08)

Index name:	X1
Index schema:	SAMPLEDB01
System index name:	System-generated
Table name:	EMPLOYEE
Table schema:	SAMPLEDB01

Available columns:

Column Name	Short Na...	Data Type	Length
EMPNO	EMPNO	CHARAC...	6
FIRSTNME	FIRSTNME	VARCHAR	12
MIDINIT	MIDINIT	CHARAC...	1
WORKDEPT	WORKD...	CHARAC...	3
PHONENO	PHONENO	CHARAC...	4
HIREDATE	HIREDATE	DATE	
JOB	JOB	CHARAC...	8
EDLEVEL	EDLEVEL	SMALLINT	
SEX	SEX	CHARAC...	1
BIRTHDATE	BIRTHDA...	DATE	

Selected columns:

Order	Column Name	Sh
Ascending	LASTNAME	LA

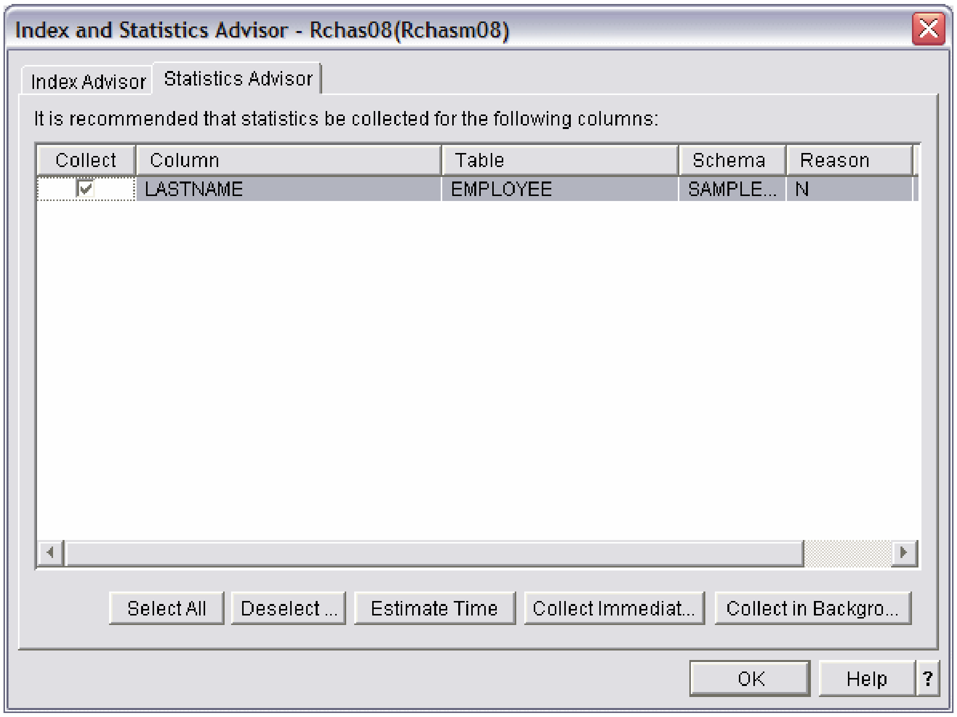
Index type:	Not unique
Number of distinct values:	0
Number of tasks:	Use current setting
Page size:	Default
Text:	

Figure 5-46 Creating a new index

Statistics Advisor

For this query, the optimizer also suggests that query performance can be improved if statistics are available in certain table columns. Figure 5-47 shows an example of the Statistics Advisor page.

You can create these suggested statistics directly from this window. However, if you set the QDBFSTCCOL system value to either *SYSTEM or *ALL, statistics are automatically collected by the QDBFSTCCOL system job whenever the optimizer suggests that statistics are useful.



It is recommended that statistics be collected for the following columns:

Collect	Column	Table	Schema	Reason
☑	LASTNAME	EMPLOYEE	SAMPLE...	N

Figure 5-47 Visual Explain Statistics Advisor page

5.2.4 Optimization goal starting in V5R2

Optimization can be biased toward returning the first few rows in the result set, usually sufficient to display the first panel for the user. Alternatively, it can be biased toward returning every row of the result set.

This bias is controlled by different means depending on the query interface that is used. For example, CLI uses the default *ALLIO. SQL uses the optional “OPTIMIZE FOR n ROWS” clause. If this clause is not specified, the optimizer uses the default value as specified in Table 5-1. If *FIRSTIO is selected, the optimizer tends to favor table scans or use existing indexes. In the case of CQE, optimizing for all rows or for a number of rows greater than the number in the result set may make the creation of a temporary index more likely if no satisfactory permanent indexes existed.

While SQE does not build temporary indexes, the SQE Optimizer still uses the optimization goal as a “costing” decision. However, when the clause is omitted from the SQL statement, SQE uses a different default value for the number of rows for which to optimize. The CQE default is 3% of the total number of rows anticipated in the result set. SQE defaults to 30 rows.

Table 5-1 Default optimization goals for each interface (as of V5R2)

Interfaces	Default optimization goals
Embedded SQL	*ALLIO
Dynamic Embedded SQL	*FIRSTIO
Interactive SQL	*FIRSTIO
CLI	*ALLIO
ODBC,OLE DB, .NET Provider	*FIRSTIO (*ALLIO with extended dynamic SQLPKG)
JDBC	*FIRSTIO (*ALLIO with extended dynamic SQLPKG)
Native JDBC Driver	*ALLIO
Net.Data®	*FIRSTIO
Data Transfer (iSeries Access for Windows)	*ALLIO

5.2.5 Other considerations

There are additional considerations that you must take into account when doing performance tuning. We explore some of those considerations in the following sections.

Plan cache and access plans

Starting in V5R2, a system IPL causes the plan cache to be cleared. Therefore, you can expect a query's performance to be affected when it is run for the first time after an IPL because the access plan needs to be rebuilt. However, because of the need to perform ODP creations after an IPL, the access plan rebuild overhead may not be noticeable.

There are several other situations that cause a query's access plan to be rebuilt in addition to those situations described in 5.2.1, "System environment" on page 133. These include:

- Creating or deleting indexes
- Table size changing by 10%
- Creating a new statistic, automatically or manually
- Refreshing a statistic, automatically or manually
- Removing a stale statistic
- Applying database related PTFs

Notice that access plans are marked as invalid after an OS/400 or i5/OS release upgrade.

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.

Handing queries back from SQE to CQE

You can minimize performance overhead that is incurred as a result of the optimizer handing queries back from SQE to CQE (see 2.3.1, “The Query Dispatcher” on page 16) by avoiding those functions that cause this behavior. For example, avoid the presence of any of the following characteristics on tables that you query:

- Logical files with select or omit keys
- Logical files with derived keys: Perform intermediate mapping of keys
  - Renaming fields
  - Alternate collating sequence
  - Translation
- Index or logical file with National Language Sort Sequence (NLSS) applied
- Keyed logical files built across multi-member physical files

An exception applies only if your query has no local selection, grouping, or ordering, and is forced unconditionally to perform a table scan.

6

Practical experience

This chapter provides information about practical experiences gained so far. It shows performance comparisons for queries running under OS/400 V5R2 using Classic Query Engine (CQE) to the same queries running under OS/400 V5R2 that take advantage of Structured Query Language (SQL) Query Engine (SQE). The tests were performed on identical systems and identical configurations. They give a close approximation of performance gains that can be expected between OS/400 V5R1 and OS/400 V5R2.

The queries that were chosen to be measured were selected from numerous sources such as customer applications, IBM Business Partners, and industry benchmarks.

6.1 General performance findings

Figure 6-1 shows response time improvements measured in an environment with 119 different, longer running Business Intelligence (BI) type queries. In this specific environment, response time with SQE was less than half of the response time with CQE for most of the queries. To help you understand the complexity of the queries that were used in this test, the original runtime was between 2 and 600 seconds per single query.

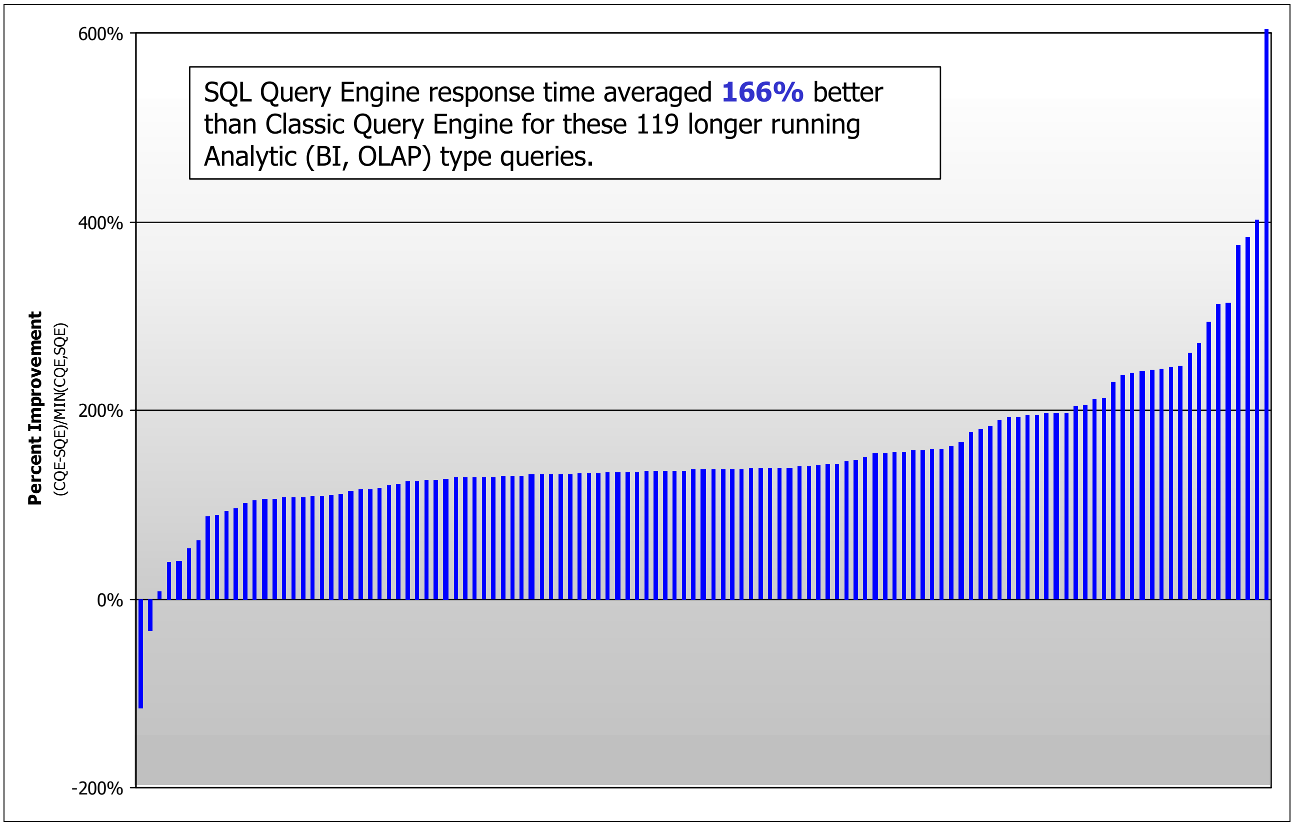


Figure 6-1 Response time improvements in a BI environment

In addition to achieving mainly better performance in this BI type of environment, we saw improvements in a mixed environment where query execution time was in the range from 1 to 15 seconds. See Figure 6-2.

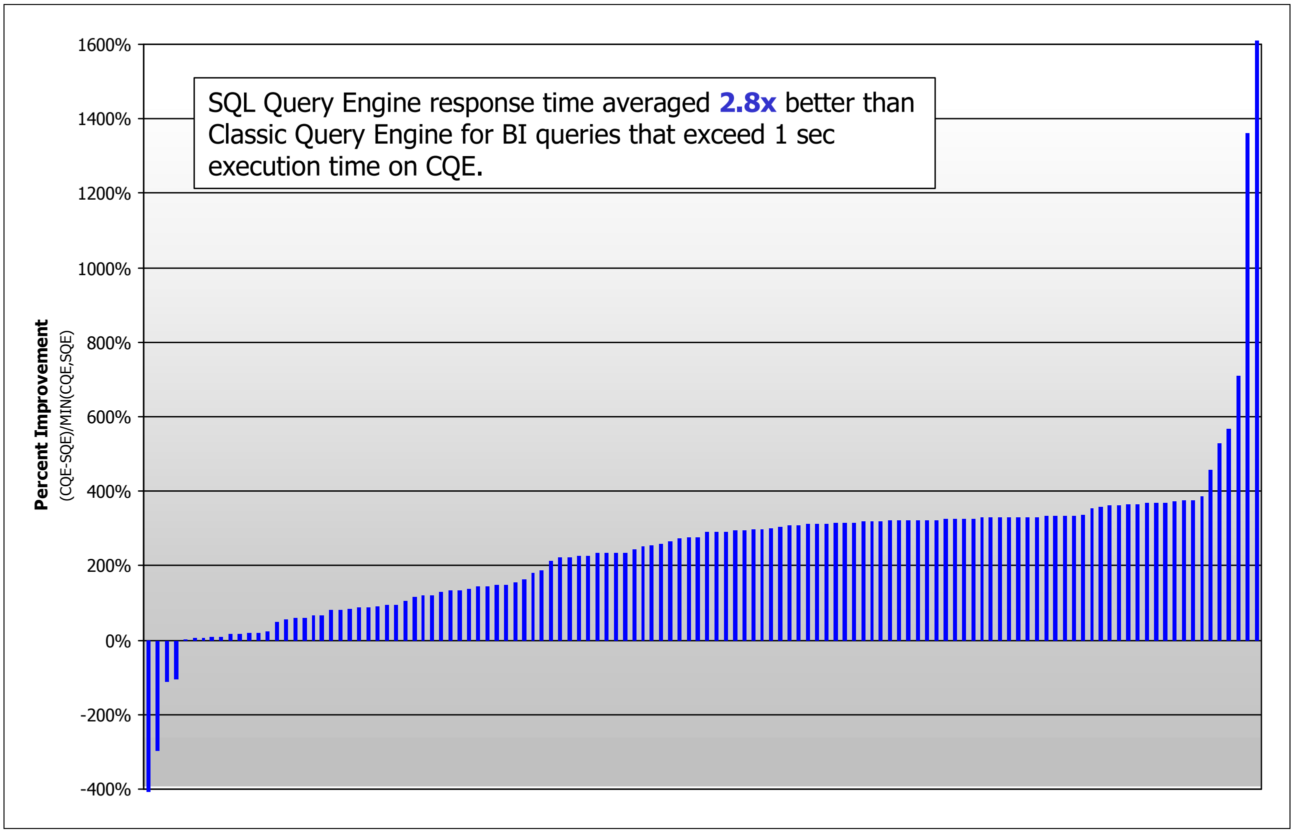
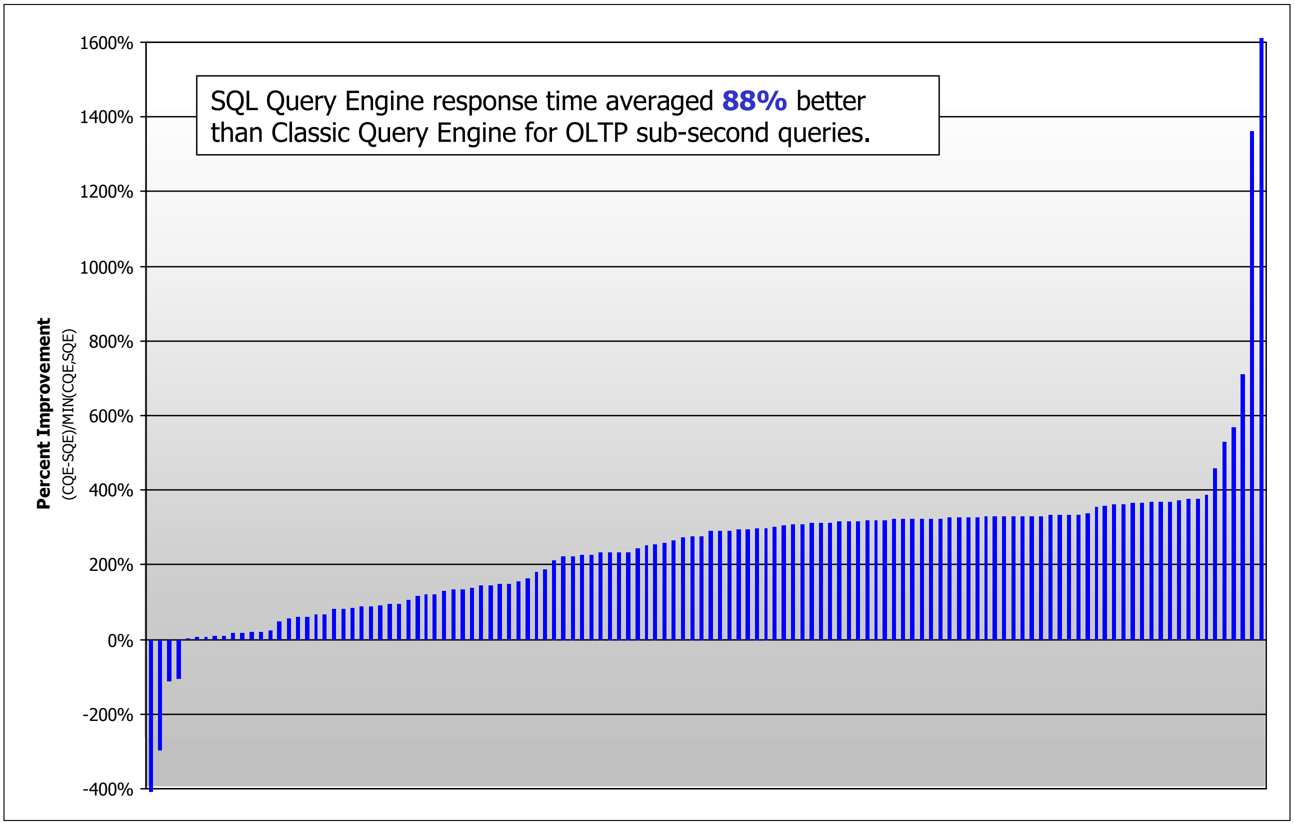


Figure 6-2 Response time improvements in a mixed environment

Figure 6-3 shows another example of the performance improvements. This chart compares query execution times for rather short running queries. All of them show subsecond execution times.



SQL Query Engine response time averaged **88%** better than Classic Query Engine for OLTP sub-second queries.

Figure 6-3 Response time improvements for short running queries

You should be aware that response time improvements vary in different customer environments.

6.2 Join performance enhancements

Join performance is an area that has demonstrated great performance results using SQE as compared to CQE. This is due to the fact that SQE's implementation of temporary result objects is much more efficient than CQE. Running a given set of 20 queries that looked like the one shown in Example 6-1, the overall runtime was 15 times faster using SQE than using CQE. These results were achieved using a database of 100 GB with symmetric multiprocessing (SMP) activated on a 12-way system with 120 GB of main memory.

Example 6-1 Join query

```
SELECT   AL1.CUSTOMER, AL5.YEAR, AL5.MONTHNAME, SUM(AL2.QUANTITY)
FROM     ITEM_FACT AL2, CUST_DIM AL1, TIME_DIM AL5
WHERE    (AL2.CUSTKEY=AL1.CUSTKEY AND AL5.DATEKEY=AL2.SHIPDATE)
AND      (AL5.YEAR=1998 AND AL5.MONTHNAME='October')
GROUP BY AL1.CUSTOMER, AL5.YEAR, AL5.MONTHNAME
ORDER BY 4 DESC
```

6.3 DISTINCT performance enhancements

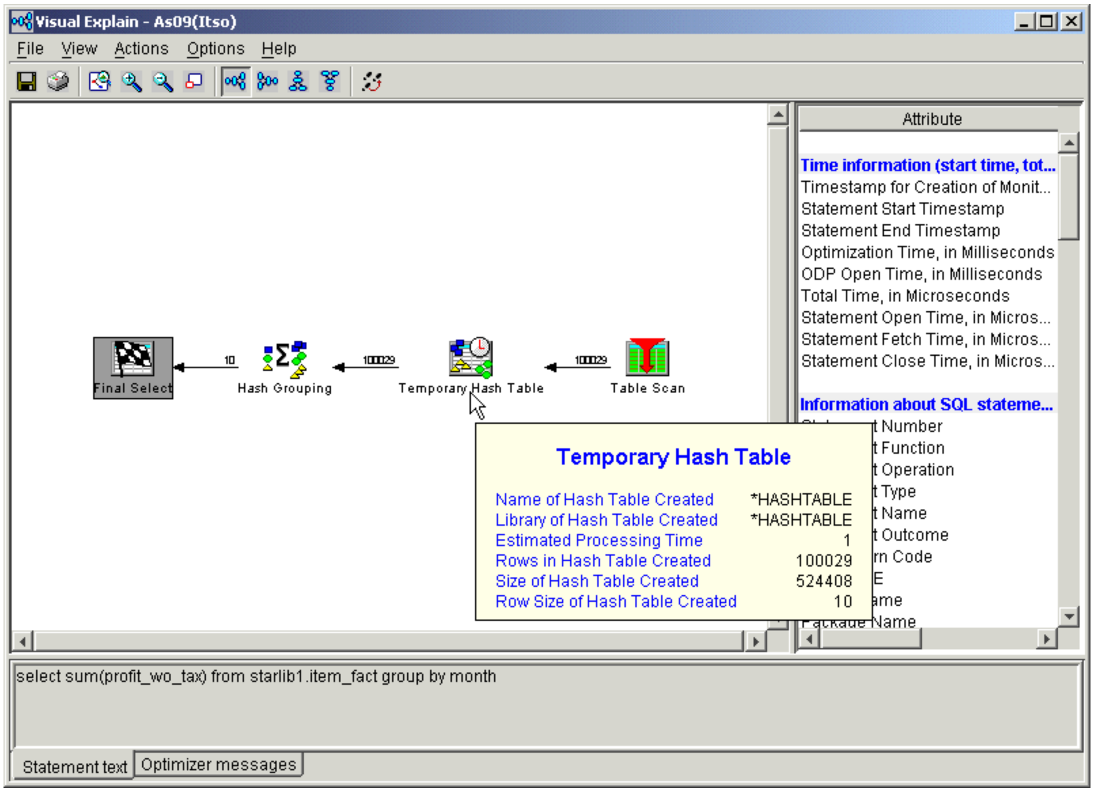
Another area where we observed large performance improvements is in grouping and the use of DISTINCT. Again tests were performed running a set of 10 different queries comparable to the one shown in Example 6-2. The overall runtime was six times faster when using SQE compared to CQE. These results were achieved using a database of 10 GB with SMP activated on a 12-way system with 120 GB of main memory.

Example 6-2 Query using DISTINCT

```
SELECT   YEAR, QUARTER, MONTH, CUSTKEY,
         SUM(REVENUE_WO_TAX)                     AS TOTAL_REVENUE_WO_TAX,
         SUM(REVENUE_W_TAX)                      AS TOTAL_REVENUE_W_TAX,
         SUM(PROFIT_WO_TAX)                      AS TOTAL_PROFIT_WO_TAX,
         SUM(PROFIT_W_TAX)                       AS TOTAL_PROFIT_W_TAX,
         SUM(QUANTITY)                           AS TOTAL_QUANTITY,
         SUM(SUPPLYCOST)                         AS TOTAL_SUPPLYCOST,
         COUNT(DISTINCT ORDERKEY)                AS TOTAL_NO_OF_ORDERS,
         COUNT(DISTINCT (ORDERKEY + LINENUMBER)) AS TOTAL_NO_OF_LINES,
         AVG(BIGINT(DAYS_ORDER_TO_SHIP))         AS AVG_DAYS_ORDER_TO_SHIP,
         AVG(BIGINT(DAYS_ORDER_TO_RECEIPT))      AS AVG_DAYS_ORDER_TO_RECEIPT,
         AVG(BIGINT(DAYS_SHIP_TO_RECEIPT))       AS AVG_DAYS_SHIP_TO_RECEIPT,
         AVG(BIGINT(DAYS_COMMIT_TO_RECEIPT))     AS AVG_DAYS_COMMIT_TO_RECEIPT
FROM     ITEM_FACT
GROUP BY YEAR, QUARTER, MONTH, CUSTKEY
```

6.4 Influence of column statistics

Having column statistics available may help reducing optimization time. It may also help in reducing query execution time. An example of this is grouping over columns that do not have indexes available. The example in Figure 6-4 shows that the query is implemented using hash grouping. Since the Statistics Manager can only use default values for the cardinality of the *month* column, the generated hash table is much larger than it actually needs to be. This results in longer query execution time.



Time information (start time, tot...
Timestamp for Creation of Monit...
Statement Start Timestamp
Statement End Timestamp
Optimization Time, in Milliseconds
ODP Open Time, in Milliseconds
Total Time, in Microseconds
Statement Open Time, in Micros...
Statement Fetch Time, in Micros...
Statement Close Time, in Micros...

Information about SQL stateme...
t Number
t Function
t Operation
t Type
t Name
t Outcome
rn Code
E
ame
Package Name

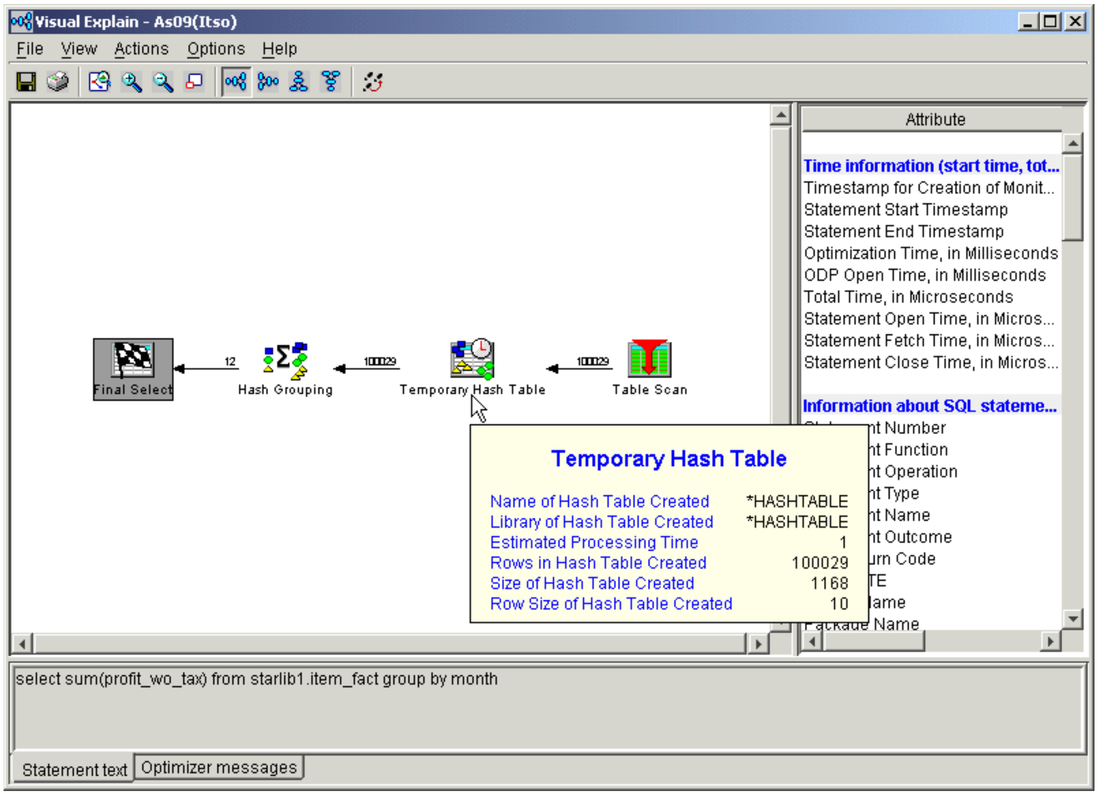
Temporary Hash Table

Name of Hash Table Created	*HASHTABLE
Library of Hash Table Created	*HASHTABLE
Estimated Processing Time	1
Rows in Hash Table Created	100029
Size of Hash Table Created	524408
Row Size of Hash Table Created	10

```
select sum(profit_wo_tax) from starlib1.item_fact group by month
```

Figure 6-4 Optimization without column statistics

Consider the example in Figure 6-5. The same query was given to the optimizer. Only this time, column statistics for *month* are available. The optimizer comes back with the same access plan as before, but the size of the temporary hash table is much smaller now. This is because the Statistics Manager can provide a more accurate answer for the cardinality of that column.



Temporary Hash Table

Name of Hash Table Created	*HASHTABLE
Library of Hash Table Created	*HASHTABLE
Estimated Processing Time	1
Rows in Hash Table Created	100029
Size of Hash Table Created	1168
Row Size of Hash Table Created	10

Time information (start time, tot...
Timestamp for Creation of Monit...
Statement Start Timestamp
Statement End Timestamp
Optimization Time, in Milliseconds
ODP Open Time, in Milliseconds
Total Time, in Microseconds
Statement Open Time, in Micros...
Statement Fetch Time, in Micros...
Statement Close Time, in Micros...

Information about SQL stateme...
nt Number
nt Function
nt Operation
nt Type
nt Name
nt Outcome
urn Code
TE
lame
Package Name

```
select sum(profit_wo_tax) from starlib1.item_fact group by month
```

Figure 6-5 Optimization with column statistics

6.5 Fenced versus unfenced user-defined functions

With V5R2, when you create a user-defined function (UDF), you can decide whether to make the UDF a fenced UDF or an unfenced UDF. By default, UDFs are created as fenced UDFs.

Fenced indicates that the database should run the UDF in a separate thread. For complex UDFs, this separation is meaningful, because it avoids potential problems such as generating unique SQL cursor names. Not having to be concerned about resource conflicts is one reason to use the default and create the UDF as a fenced UDF.

A UDF created with the NOT FENCED option indicates to the database that the user is requesting that the UDF can run within the same thread that initiated the UDF. Unfenced is a suggestion to the database that can still decide to run the UDF in the same manner as a fenced UDF.

Example 6-3 shows creating a fenced UDF versus an unfenced UDF.

Example 6-3 Creating a fenced UDF versus creating an unfenced UDF

```
CREATE FUNCTION QGPL.FENCED (parameter1 INTEGER)
RETURNS INTEGER LANGUAGE SQL
BEGIN
RETURN parameter1 * 3;
END;

CREATE FUNCTION QGPL.UNFENCED1 (parameter1 INTEGER)
RETURNS INTEGER LANGUAGE SQL NOT FENCED
-- Build the UDF to request faster execution via the NOT FENCED option
BEGIN
RETURN parameter1 * 3;
END;
```

The use of unfenced versus fenced UDFs provides better performance since the original query and the UDF can run within the same thread. We ran tests using 56 different statements of the form shown in Example 6-4 within one job. We gradually increased the number of concurrent jobs on the system. This test was run on a 12-way system with 15 GB of main memory and 135 disk arms. The underlying database table contains nearly 200,000 records and has a size of nearly 220 MB. The UDF DB2TX.CONTAINS was implemented once as a fenced UDF and once as an unfenced UDF.

Example 6-4 SQL statement using text extender UDFs

```
SELECT MOVIE_ID, FORMAT_CODE, MOVIE_YEAR, MOVIE_NAME, MOVIE_RATING
FROM EMOVIES.MOVIES
WHERE DB2TX.CONTAINS ( H_PLOT_L, '("Yinshi","Creek","Riding")' ) = 1
ORDER BY FORMAT_CODE, MOVIE_YEAR desc, MOVIE_RATING ;
```

The results are shown in Figure 6-6 and Figure 6-7. They clearly show that the elapsed time for executing the 56 statements is much better when using unfenced UDFs, especially since the number of jobs on the system increases. This implementation requires greater central processing unit (CPU) resource utilization.

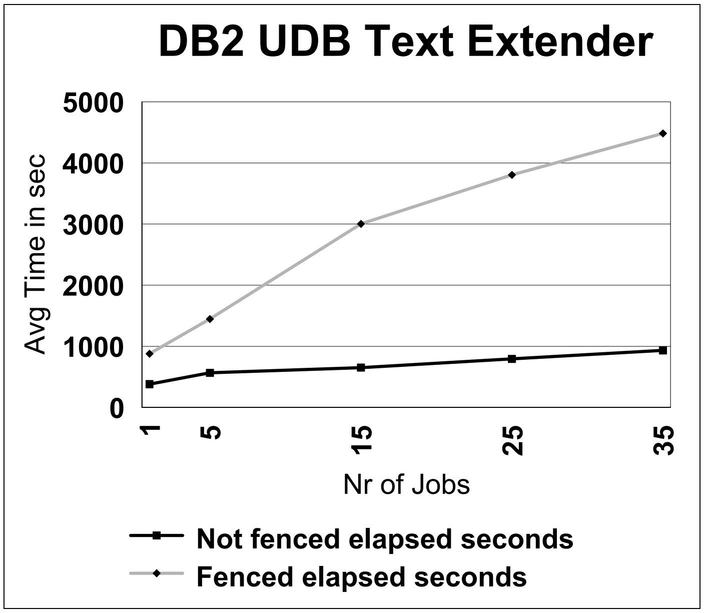


Figure 6-6 Elapsed time using fenced versus unfenced UDFs

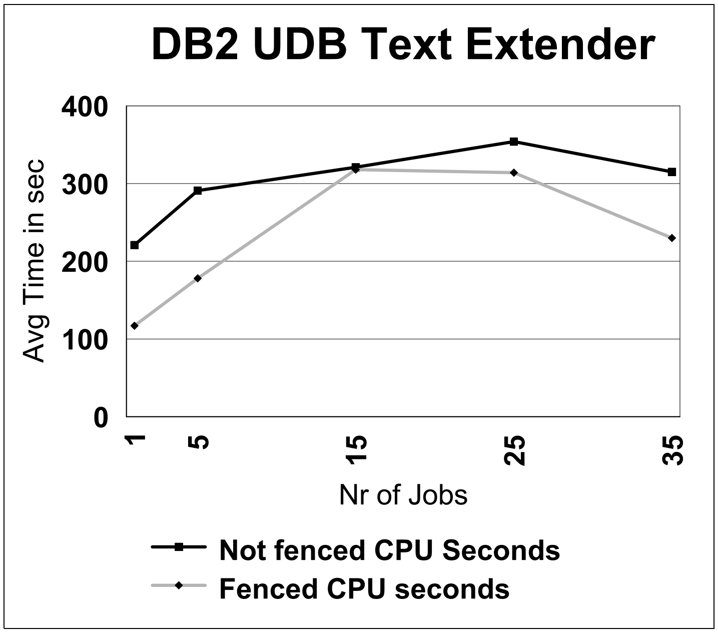


Figure 6-7 CPU seconds using fenced versus unfenced UDFs

6.6 Reading through an index backwards

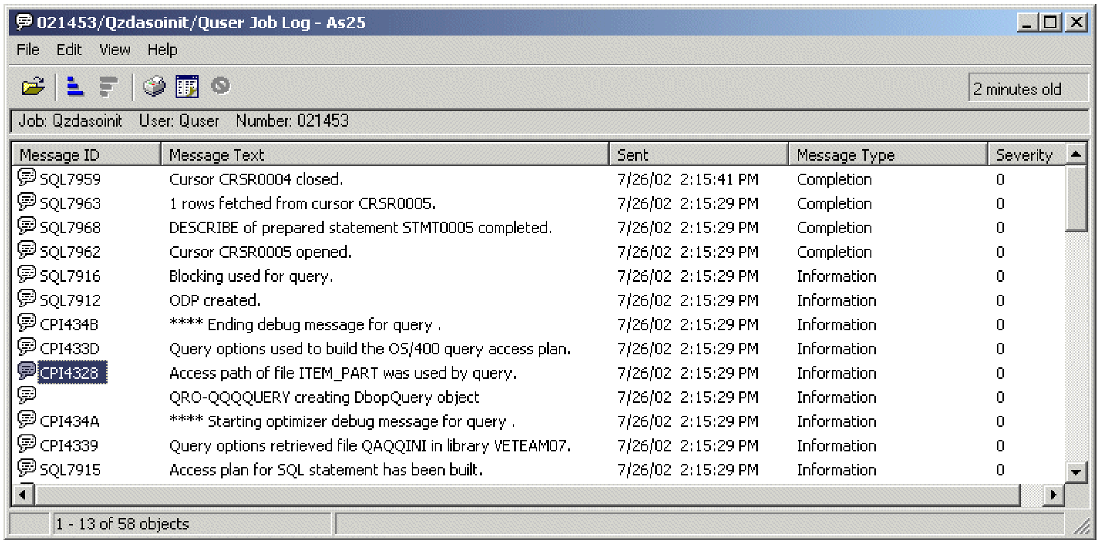
Before OS/400 V5R2, the system could only read through an index forward for ordering purposes. Starting with V5R2, you can read backwards through an index for ordering. This means that an index in ascending order can be used to execute a query with ordering or grouping in descending sequence.

We tested the following example:

```
CREATE INDEX ITEM_PART ON ITEM_FACT (PARTKEY ASC)
SELECT MAX(PARTKEY) FROM ITEM_FACT
```

In this example, it was only necessary to read the last entry in the ascending index to find the maximum value. Prior to OS/400 V5R2, you needed to either read through the entire table or create an index in descending order.

As you can see in Figure 6-8, the index was actually used to implement the query. This resulted in substantially lower I/O and execution time.



Job: Qzdasoinit User: Quser Number: 021453

Message ID	Message Text	Sent	Message Type	Severity
SQL7959	Cursor CRSR0004 closed.	7/26/02 2:15:41 PM	Completion	0
SQL7963	1 rows fetched from cursor CRSR0005.	7/26/02 2:15:29 PM	Completion	0
SQL7968	DESCRIBE of prepared statement STMT0005 completed.	7/26/02 2:15:29 PM	Completion	0
SQL7962	Cursor CRSR0005 opened.	7/26/02 2:15:29 PM	Completion	0
SQL7916	Blocking used for query.	7/26/02 2:15:29 PM	Information	0
SQL7912	ODP created.	7/26/02 2:15:29 PM	Information	0
CPI434B	**** Ending debug message for query .	7/26/02 2:15:29 PM	Information	0
CPI433D	Query options used to build the OS/400 query access plan.	7/26/02 2:15:29 PM	Information	0
CPI4328	Access path of file ITEM_PART was used by query.	7/26/02 2:15:29 PM	Information	0
	QRO-QQQQUERY creating DbopQuery object	7/26/02 2:15:29 PM	Information	0
CPI434A	**** Starting optimizer debug message for query .	7/26/02 2:15:29 PM	Information	0
CPI4339	Query options retrieved file QAQQINI in library VETEAM07.	7/26/02 2:15:29 PM	Information	0
SQL7915	Access plan for SQL statement has been built.	7/26/02 2:15:29 PM	Information	0

1 - 13 of 58 objects

Figure 6-8 Debug messages for using index

6.7 VARCHAR

When a column is defined as a VARCHAR data type, it is given an allocated length, either explicitly or by default. When data is inserted into the column, and the data’s length exceeds the allocated length, the complete data for this column is placed into an overflow area and the allocated length is not used.

There is a performance impact when accessing the data because additional I/O operations are generated to retrieve the data in the overflow area. Because SQE drives asynchronous I/O operations much more aggressively than CQE, you should see improved performance when handling VARCHAR data.

6.8 Encoded-vector index maintenance enhancements

Figure 6-9 shows the internal representation of an encoded-vector index (EVI). An EVI consists of a symbol table, which is a sorted list of distinct key values with a code assigned to each value. Since this list is sorted, a binary search can be used on it.

The EVI maintenance problem may arise when adding records that do not arrive in ascending order to the base file. Because there is no room between two entries in the symbol table and it must stay sorted, out-of-range values are placed into a non-sorted overflow area. Since this overflow area is not sorted, a linear search must be performed each time a specific key is searched for in the overflow area.

Considering that this may cause a performance impact, thresholds are placed on the number of values in the overflow area. They depend on the byte code size of the EVI:

- 1 Byte EVI → Threshold set to 100 entries in the overflow area
- 2 Byte EVI → Threshold set to 1,000 entries in the overflow area
- 4 Byte EVI → Threshold set to 10,000 entries in the overflow area

Each time the threshold is reached, the EVI becomes invalidated. It is refreshed, incorporating the overflow area into the sorted symbol table. This means that inserting new values into a table, with an EVI defined on it, generated considerable overhead when:

- The new value was out of order.
- The new value was out of order and the overflow area threshold was reached.

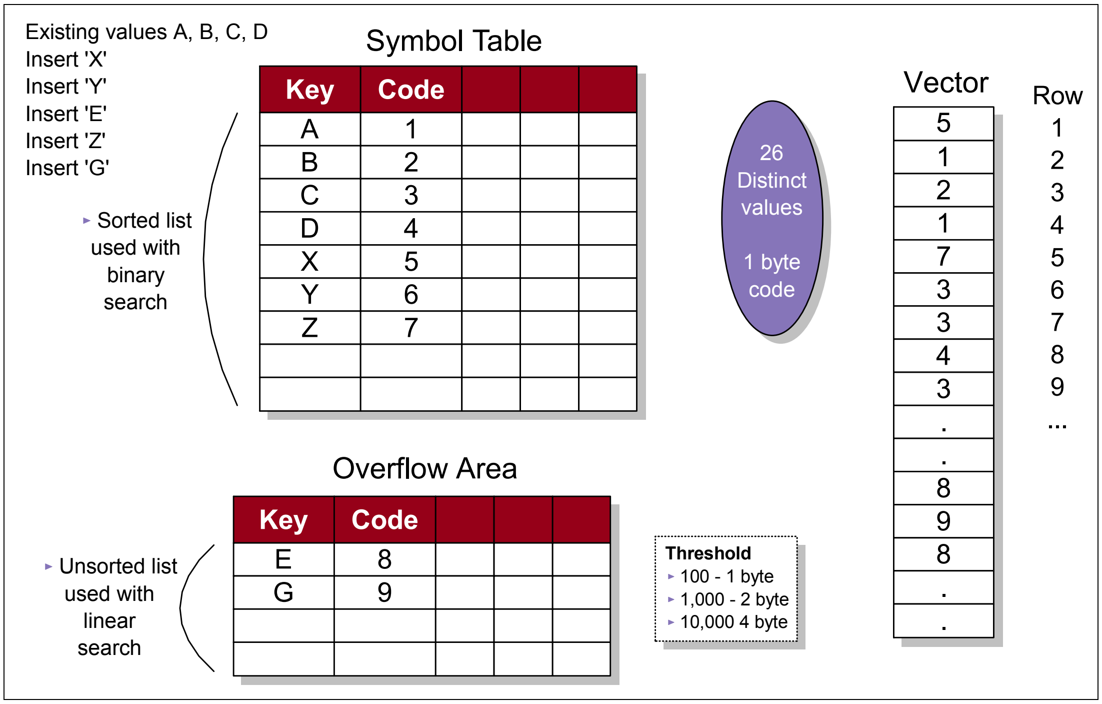


Figure 6-9 EVI maintenance

Because of this behavior, one of the main recommendations is to use EVIs only on read-only tables or on tables with a minimum of INSERT, UPDATE or DELETE activity. Another recommendation is to create EVIs with the maximum byte code size expected for the table. When loading data, the recommendation is to drop the EVIs, load the data and then create the EVIs again so that all data is held in the symbol table.

Starting with V5R2, the overflow area is indexed after 100 entries are present. With this, a binary search is now possible for the overflow area. The internal threshold was raised to 500,000 entries independent of the byte code size of the EVI.

Keep in mind that the index on the overflow area does not persist over an initial program load (IPL). Also, when saving the EVI, the radix index on the overflow area is not saved. It is recreated “on first touch”. This can be during query optimization time or by using the Display Library (DSPLIB) command.

> **Important:** When you save an EVI from a system with V5R2 installed, and restore this EVI to a system with an older version of OS/400, the new threshold of 500,000 entries in the overflow area is preserved on that system. In these cases, recreate the EVI. Otherwise, performance problems may occur when using the EVI.

With the new threshold of 500,000 entries for the overflow area and the index over the overflow area, it is possible to use EVIs on tables with a larger insert rate than before V5R2. We tested the worst case for an EVI by inserting rows with new key values into a table in reverse order. We achieved insert rates that were more than 1,000 times higher than when trying to do the same on a system with V5R1 installed.

Note: You can expect EVI maintenance performance gains on your production system to be lower. Our test looked at the worst case for using EVIs prior to V5R2.

A

Statistics Manager API code examples

This appendix provides provides sample code on how to use the APIs inside a program. To learn more about the types of APIs that are available with Statistics Manager in iSeries Navigator, see 4.4, “Statistics Manager APIs” on page 87.

Create Statistics Collection

Example A-1 shows how to use the Create Statistics Collection API to generate a CL command to actually create column statistics for a given table. Using this command, you can generate user requests for column statistics that are either executed immediately or in the background.

Example: A-1   Create Statistics Collection API

```
// DESCRIPTION:  Create database file statistics
//
// FUNCTION:     Program for sample CRTDBFSTC CL command
//               using the QdbstRequestStatistics API.
//
// PARAMETER LIST:
//  PARM   USE  TYPE         DESCRIPTION
//    1     I   CHAR(10)     Library ASP device name
//    2     I   CHAR(20)     Qualified database file name (file/lib)
//    3     I   CHAR(10)     Database file member
//    4     I   CHAR(12)     Collection mode
//    5.1   I   BIN(16)      Number of stats collections following
//    5.1.1 I   BIN(16)      Displacement to first stats collection
//    ...
//    5.1.n I   BIN(16)      Displacement to last stats collection
//    i-th stats collection at given displacment:
//    5.i.1 I   BIN(16)      Number of elements specified (1...4)
//    5.i.2 I   CHAR(10)     Column name
//    5.i.3 I   CHAR(30)     Stats name
//    5.i.4 I   CHAR(10)     Aging mode
//    5.i.5 I   CHAR(20)     Qualified translation table name (file/lib)
//


Have CRTDBFSTC.C400 source as DBSTCMD/QCPPSRC(CRTDBFSTC)


CRTCPPMOD MODULE(DBSTCMD/CRTDBFSTC)
          SRCFILE(DBSTCMD/QCPPSRC)

CRTPGM PGM(DBSTCMD/CRTDBFSTC)
       MODULE(DBSTCMD/CRTDBFSTC)
       ENTMOD(CRTDBFSTC)
       BNDSRVPGM(QSYS/QDBSTMGR QSYS/QP0ZCPA)
       ACTGRP(*CALLER)


//--- Have the following in DBSTCMD/QCMDSRC(CRTDBFSTC) ------------------------


CMD        PROMPT('Create DB-File Statistics')
             PARM       KWD(ASPDEV) TYPE(*SNAME) LEN(10) +
                          DFT(*) PROMPT('Library ASP device name') +
                          SPCVAL((*) (*SYSBAS))
             PARM       KWD(FILE) TYPE(FILEQUAL) PROMPT('Database file')
             PARM       KWD(MBR) TYPE(*SNAME) PROMPT('Member') +
                            DFT(*FIRST) SPCVAL((*FIRST) (*LAST))
             PARM       KWD(COLLMODE) TYPE(*CHAR) LEN(12) RSTD(*YES) +
                          DFT(*IMMEDIATE) SPCVAL((*IMMEDIATE) +
                          (*BACKGROUND)) PROMPT('Collection mode')
             PARM       KWD(STCCOLL) TYPE(STCCOLLFMT) MIN(0) +
                        MAX(300) LISTDSPL(*INT2)

/* QUALIFIED FILE NAME */
 FILEQUAL:   QUAL       TYPE(*SNAME) LEN(10)
             QUAL       TYPE(*SNAME) LEN(10) DFT(*LIBL) +
                          SPCVAL((*LIBL) (*CURLIB) (*USRLIBL)) +
                          PROMPT('Library')

/* QUALIFIED TRANSLATE TABLE NAME */
 TBLQUAL:    QUAL       TYPE(*SNAME) LEN(10) DFT(*NONE) +
                          SPCVAL((*NONE '          '))
             QUAL       TYPE(*SNAME) LEN(10) DFT(*NONE) +
                          SPCVAL((LIBL) (*CURLIB) +
                                (*USRLIBL) (*NONE '          ')) +
                          PROMPT('Library')

/* A single, single column stats collection definition */
STCCOLLFMT:  ELEM       TYPE(*SNAME) PROMPT('Column name')
             ELEM       TYPE(*NAME) LEN(30) DFT(*GEN) +
                          SPCVAL((*GEN)) PROMPT('Statistics name')
             ELEM       TYPE(*CHAR) LEN(10) RSTD(*YES) DFT(*SYS) +
                          SPCVAL((*SYS) (*USER)) PROMPT('Aging mode')
             ELEM       TYPE(TBLQUAL) PROMPT('Translate table')


// Use the following to create the actual command -----------------------------


CRTCMD CMD(DBSTCMD/CRTDBFSTC)
       PGM(DBSTCMD/CRTDBFSTC)
       SRCFILE(DBSTCMD/QCMDSRC)
       SRCMBR(CRTDBFSTC)


//--- Start of C++ Source Code ------------------------------------------------


// database file statistics APIs
#ifndef QDBSTMGR_H
#include <qdbstmgr.h>
#endif

// memcpy, ...
#ifndef memory_h
#include <memory.h>
#endif

// API error code structures
#ifndef QUSEC_h
#include <qusec.h>
#endif

// job log output
#ifndef __QPOZTRC_H
#include <qp0ztrc.h>
#endif

// convert hex to char
#ifndef __cvthc_h
#include <mih/cvthc.h>
```

```
#endif

// exception handling
#ifndef __except_h
#include <except.h>
#endif

// resend escape message
#ifndef QMHRSNEM_h
#include <qmhrsnem.h>
#endif

#include <assert.h>

//-----------------------------------------------------------------------------
// Calculate the stats name length (quoted '"', or ' ' or '\0' terminated)
static QDBST_uint32 QdbstCmd_calcStatsNameLen
(
 const char *StatsName,
 const size_t maxLen
);
//-----------------------------------------------------------------------------
// Resignal exceptions to one above the invocation pointer given as
// communications "area" passed on the #pragma exception_handler.
static void QdbstCmd_ResignalHdlr(_INTRPT_Hndlr_Parms_T *errmsg);
//-----------------------------------------------------------------------------
// CL inner list input for single stats collection definition
struct QdbstCmd_SingleStats_argv_t
{
 QDBST_uint16 nElements;
 char colName[10];
 char statsName[30];
 char agingMode[10];
 char transTable[20];
};
//-----------------------------------------------------------------------------
// Dynamic allocation of API input buffer and exception safe cleanup
struct QdbstCmd_AutoPtr
{
    QdbstCmd_AutoPtr(const size_t Size)
    : xPtr(new char[Size]),
      xSize(Size)
    {}

    ~QdbstCmd_AutoPtr()
    { delete[] xPtr; }

    char  *xPtr;
    size_t xSize;
};
//-----------------------------------------------------------------------------
int main(int argc, char *argv[])
{
 // setup exception resignal handler
    // get invocation pointer for resignal scoping
    volatile _INVPTR invPtr = _INVP(0);

    #pragma exception_handler (QdbstCmd_ResignalHdlr, invPtr, 0, _C2_MH_ESCAPE)

 // check number of parameters
 // ...

 // address (sub set of) input parameters
    const char* const ASPDeviceName   = argv[1];
    const char* const FileName        = argv[2];
    const char* const LibName         = argv[2] + 10;
    const char* const MbrName         = argv[3];
    const char* const CollMode        = argv[4];
    const char* const ListBuffer      = argv[5];

    const QDBST_uint16 nStats         = *(const QDBST_uint16 *) ListBuffer;
    assert(nStats);

    const QDBST_uint16 *pStatsDispl   = ((QDBST_uint16 *) ListBuffer) + 1;

 // allocate API input buffer
    QdbstCmd_AutoPtr pInputRequestKeeper(
        sizeof(Qdbst_STIR0100_Hdr_t) +
        nStats * (sizeof(Qdbst_STIR0100_StatsHdr_t) +
                    sizeof(Qdbst_STIR0100_StatsDtl_t)));

    struct InputRequest_t
    {
       Qdbst_STIR0100_Hdr_t hdr;
       struct SingleStats_t
       {
            Qdbst_STIR0100_StatsHdr_t statshdr;
            Qdbst_STIR0100_StatsDtl_t statsdtl;
       } Stats[1]; // gives addressiblity via and beyond first element
    } *pInputRequest = (InputRequest_t *) pInputRequestKeeper.xPtr;

 // init input buffer
    memset(pInputRequest, 0, pInputRequestKeeper.xSize);

 // set input header
    memcpy(pInputRequest->hdr.ASPDeviceName, ASPDeviceName, 10);
    memcpy(pInputRequest->hdr.FileName, FileName, 10);
    memcpy(pInputRequest->hdr.FileLibraryName, LibName, 10);
    memcpy(pInputRequest->hdr.FileMemberName, MbrName, 10);
    memcpy(pInputRequest->hdr.CollectionMode, CollMode, 12);
    pInputRequest->hdr.iStatsOffset = offsetof(InputRequest_t, Stats);
    pInputRequest->hdr.iStatsCount = nStats;

 // fill single stats collections in input buffer
    for(unsigned int iStats = 0; iStats < nStats; ++iStats)
    {
        // address current stats coll input
        QdbstCmd_SingleStats_argv_t *pStatsArgv =
            (QdbstCmd_SingleStats_argv_t *) (ListBuffer + pStatsDispl[iStats]);
        assert(pStatsArgv->nElements == 4);

        // set stats hdr
        pInputRequest->Stats[iStats].statshdr.iStatsLen =
            sizeof(Qdbst_STIR0100_StatsHdr_t) +
            sizeof(Qdbst_STIR0100_StatsDtl_t);
        pInputRequest->Stats[iStats].statshdr.iStatsNameLen =
            QdbstCmd_calcStatsNameLen(pStatsArgv->statsName,
                                        sizeof(pStatsArgv->statsName));
        memcpy(pInputRequest->Stats[iStats].statshdr.StatsName,
                pStatsArgv->statsName,
                    pInputRequest->Stats[iStats].statshdr.iStatsNameLen);
        memcpy(pInputRequest->Stats[iStats].statshdr.AgingMode,
                pStatsArgv->agingMode, 10);
        pInputRequest->Stats[iStats].statshdr.iColumnDisplacement =
                sizeof(Qdbst_STIR0100_StatsHdr_t);
        pInputRequest->Stats[iStats].statshdr.iColumnCount = 1;

        // set stats dtl
        pInputRequest->Stats[iStats].statsdtl.iColumnDefLen =
            sizeof(Qdbst_STIR0100_StatsDtl_t);
        memcpy(pInputRequest->Stats[iStats].statsdtl.ColumnName,
            pStatsArgv->colName, 10);
        memcpy(pInputRequest->Stats[iStats].statsdtl.TransTableName,
            pStatsArgv->transTable, 10);
        memcpy(pInputRequest->Stats[iStats].statsdtl.TransTableLibName,
            pStatsArgv->transTable+10, 10);

    } // end: fill single stats collections

 // setup feedback related data
 // Note: Not required, just as an example.

    // number of keys to be requested
    const unsigned int ciFeedbackKeys = 5;
    const unsigned int ciExpectedFeedbackKeys = 3 + 2 * nStats;

    unsigned int FeedbackKeys[ciFeedbackKeys] =
                    {Qdbst_KeyNo_ElapsedTime,
                     Qdbst_KeyNo_RequestID,
                     Qdbst_KeyNo_StatsCount,
                     Qdbst_KeyNo_StatsID,
                     Qdbst_KeyNo_StatsName};

    struct Fullfeedback_t
    {
     Qdbst_FeedbackArea_Hdr_t       Header;
     Qdbst_KeyValue_ElapsedTime_t   ElapsedTime;
     Qdbst_KeyValue_RequestID_t     RequestID;
     Qdbst_KeyValue_StatsCount_t    StatsCount;
     char                           Buffer[1];
    };

    QdbstCmd_AutoPtr FeedbackAreaBuffer(
        sizeof(Fullfeedback_t) - 1 + // fixed portion
        nStats * (sizeof(Qdbst_KeyValue_t) + 128) + // max stats names
        nStats * (sizeof(Qdbst_KeyValue_StatsID_t)));

 // call the Create Stats API

    // force exceptions to be signalled
        Qus_EC_t ec;
        ec.Bytes_Provided = 0;

    QDBST_uint32 iInputLen = pInputRequestKeeper.xSize;
    QDBST_uint32 iFeedbackKeys = ciFeedbackKeys;

    QdbstRequestStatistics(pInputRequest,
                          &iInputLen,
                          "STIR0100",
                          FeedbackAreaBuffer.xPtr,
                          &FeedbackAreaBuffer.xSize,
                          FeedbackKeys,
                          &iFeedbackKeys,
                          &ec);


 // log the feedback
 // Note: Not required, just as an example.
 {
    const Fullfeedback_t *const pFeedbackArea =
        (Fullfeedback_t *) FeedbackAreaBuffer.xPtr;
    char  szBuffer[256];

    assert(pFeedbackArea->Header.iKeysReturned == ciExpectedFeedbackKeys);

    QpOzLprintf("ELAPSED/ESTIMATED TIME: %i\n",
        pFeedbackArea->ElapsedTime.Data);

    if (!memcmp(CollMode, Qdbst_CollectionMode_BACKGRND,
            sizeof(Qdbst_CollectionMode_BACKGRND)-1))
    {
        cvthc(szBuffer, pFeedbackArea->RequestID.Data, 32);
        szBuffer[32] = '\0';
        QpOzLprintf("REQUESTID: %s\n", szBuffer);
    }
    else
    if (!memcmp(CollMode, Qdbst_CollectionMode_IMMEDIATE,
            sizeof(Qdbst_CollectionMode_IMMEDIATE)-1))
    {
        assert(pFeedbackArea->StatsCount.Data == nStats);

        const Qdbst_KeyValue_StatsID_t *pStatsID =
            (const Qdbst_KeyValue_StatsID_t *) pFeedbackArea->Buffer;
        for(unsigned int iStats = 0; iStats < nStats;
            ++iStats, pStatsID = (Qdbst_KeyValue_StatsID_t *)
                                     ((char *) pStatsID + pStatsID->iEntryLen))
        {
            cvthc(szBuffer, pStatsID->Data, 32);
            szBuffer[32] = '\0';
            QpOzLprintf("STATSID (%i)  : %s\n", iStats, szBuffer);
        }

        const Qdbst_KeyValue_t *pStatsName = (Qdbst_KeyValue_t *) pStatsID;
        for(unsigned int iStats = 0; iStats < nStats;
            ++iStats, pStatsName = (const Qdbst_KeyValue_t *)
                        ((char *) pStatsName + pStatsName->iEntryLen))
        {
            memcpy(szBuffer, (char *) pStatsName + sizeof(Qdbst_KeyValue_t),
                    pStatsName->iDataLen);
            szBuffer[pStatsName->iDataLen] = '\0';
            QpOzLprintf("STATSNAME (%i): %s\n", iStats, szBuffer);
        }

    } // end: if *IMMEDIATE

 } // end: log feedback

 // return success
    #pragma disable_handler
    return 0;
```

```
} // end: main
//-----------------------------------------------------------------------------
static QDBST_uint32 QdbstCmd_calcStatsNameLen
(
 const char *StatsName,
 const size_t maxLen
)
{
 QDBST_uint32 iLen = 0;

 if (StatsName[0] == '\"')
 {
    for(++iLen; (iLen < maxLen && StatsName[iLen] != '\"'); ++iLen);
    ++iLen;
 }
 else
 {
    for(;(iLen < maxLen && StatsName[iLen] != ' ' && StatsName[iLen]); ++iLen);
 }

 return iLen;
} // end: QdbstCmd_calcStatsNameLen
//-----------------------------------------------------------------------------
static void QdbstCmd_ResignalHdlr(_INTRPT_Hndlr_Parms_T *errmsg)
{
 // force errors with QMHRSNEM to be signalled, not stored
    Qus_EC_t ec;
    ec.Bytes_Provided = 0;

 // QMHRSNEM input - use format 0200 to specify "to call stack" via
 // invocation pointer
    Qmh_Rsnem_RSNM0200_t rsnm0200;

    // com area contains invocation pointer of a call stack level.
    // We will resignal to 1 (Call_Counter) above that level.
    // *PGMBDY allows us to skip ILE PEP etc.
        rsnm0200.Call_Stack_Entry = *((_INVPTR *) errmsg->Com_Area);
        rsnm0200.Call_Counter     = 1;
        memcpy(rsnm0200.Pointer_Qualifier, "*PGMBDY   ", 10);

 // resignal
    QMHRSNEM((char *)&errmsg->Msg_Ref_Key, // Message key
             &ec,                          // Error code
             &rsnm0200,                    // To call stack entry
             sizeof(rsnm0200),             // Length of to call stack
             "RSNM0200",                   // Format
             &(errmsg->Target),            // From call stack entry
             0);                           // From call stack counter

} // end: QdbstCmd_ResignalHdlr
//-----------------------------------------------------------------------------
```

Delete Statistics Collection

Example A-2 shows how to use the Delete Statistics Collection API to delete a specific or all column statistics that were collected for a table. First, you must create a C++ program. Then create a corresponding command that can then be used. To delete a specific column statistic for a table, you must know the statistics ID. You can obtain this value by using the List Statistics Collection API.

Example: A-2   Delete Statistics Collection API

```
//
// DESCRIPTION:  Program behind the DLTDBFSTC CL command
//
// FUNCTION:     Delete database file statistics using
//               the database file statistics API
//               QdbstDeleteStatistics.
//
// PARAMETER LIST:
//  PARM  USE  TYPE         DESCRIPTION
//    1    I   CHAR(10)     Library ASP device name
//    2    I   CHAR(20)     Qualified database file name (file/lib)
//    3    I   CHAR(10)     Database file member
//    4    I   HEX(16)      Statistics ID
//
// Program create and function registration
// Have source as DBSTCMD/QCPPSRC(DLTDBFSTC)


CHGCURLIB DBSTCMD

CRTCPPMOD
    MODULE(DBSTCMD/DLTDBFSTC)
    SRCFILE(DBSTCMD/QCPPSRC)

CRTSRVPGM
    SRVPGM(DBSTCMD/DLTDBFSTC)
    MODULE(DBSTCMD/DLTDBFSTC)
    BNDSRVPGM(QSYS/QDBSTMGR QSYS/QP0ZCPA)
    EXPORT(*ALL)
    STGMDL(*SNGLVL)
    ACTGRP(*CALLER)


// Have the following in DBSTCMD/QCMDSRC(DLTDBFSTC)


CMD        PROMPT('Delete DB-File Statistics')
             PARM       KWD(ASPDEV) TYPE(*SNAME) LEN(10) +
                          DFT(*) PROMPT('Library ASP device name') +
                          SPCVAL((*) (*SYSBAS))
             PARM       KWD(FILE) TYPE(FILEQUAL) PROMPT('Database file')
             PARM       KWD(MBR) TYPE(*SNAME) PROMPT('Member') +
                       DFT(*FIRST) SPCVAL((*FIRST) (*LAST))
             PARM       KWD(STCID) TYPE(*HEX) LEN(16) +
                          DFT(*ANY) PROMPT('Statistics ID') +
                        SPCVAL((*ANY 0000000000000000))

/* QUALIFIED FILE NAME */
 FILEQUAL:   QUAL       TYPE(*SNAME) LEN(10)
             QUAL       TYPE(*SNAME) LEN(10) DFT(*LIBL) +
                          SPCVAL((*LIBL) (*CURLIB) (*USRLIBL)) +
                          PROMPT('Library')


// Use the following to create the actual command


CRTCMD CMD(DBSTCMD/DLTDBFSTC)
       PGM(DBSTCMD/DLTDBFSTC)
       SRCFILE(DBSTCMD/QCMDSRC)


//----- START OF C++ SOURCE --------------------------------------------


#ifndef QDBSTMGR_H
#include <qdbstmgr.h>
#endif

#ifndef memory_h
#include <memory.h>
#endif

#ifndef QUSEC_h
#include <qusec.h>
#endif

#include <assert.h>

//--------------------------------------------------------------------------------
int main(int argc, char *argv[])
{
 // check number of parameters
 // NOTE: refine
    assert(argc == 5);

 // fill in API input
    Qdbst_STID0100_t InputDelete;

    memcpy(InputDelete.ASPDeviceName, argv[1], 10);
    memcpy(InputDelete.FileName, argv[2], 10);
    memcpy(InputDelete.FileLibraryName, argv[2]+10, 10);
    memcpy(InputDelete.FileMemberName, argv[3], 10);
    memcpy(InputDelete.StatisticsID, argv[4], 16);

 // call the API
    Qus_EC_t ec;
    ec.Bytes_Provided = 0;

    QDBST_uint32 iInputLen = sizeof(InputDelete);
    QDBST_uint32 iFeedback = 0;

    QdbstDeleteStatistics(&InputDelete,
                          &iInputLen,
                          "STID0100",
                          0, 0, 0, &iFeedback, // no feedback
                          &ec);
 // return
    return 0;

} // end: main
//------------------------------------------------------------------------------
```

Update Statistics Collection

Example A-3 shows how to update existing column statistics using the Update Statistics Collection API. Note that one of the input parameters is the *hexadecimal statistics ID*. You can obtain this value by using the List Statistics Collection API. Using this command, you can:

- Change the name of a statistics collection
- Refresh a statistics collection
- Change whether the statistics data is aged by the system or manually aged
- Block the system statistics collection for a file

Example: A-3 Update Statistics Collection API

```
// DESCRIPTION:  Update database file statistics
//
// FUNCTION:     Program for sample UPDDBFSTC CL command
//               using the QdbstUpdateStatistics API.
//
// PARAMETER LIST:
//  PARM USE  TYPE         DESCRIPTION
//    1   I   CHAR(10)     Library ASP device name
//    2   I   CHAR(20)     Qualified database file name (file/lib)
//    3   I   CHAR(10)     Database file member
//    4   I   CHAR(32)     Statistics ID
//    5   I   CHAR(12)     Data refresh (mode | *NO)
//    6   I   CHAR(10)     Aging mode (mode | *SAME)
//    7   I   CHAR(5)      Block system activity (*YES | *NO | *SAME)
//    8   I   CHAR(30)     Stats name (name | *SAME)
//
// Create commands:
//

Have UPDDBFSTC.C400 source as DBSTCMD/QCPPSRC(UPDDBFSTC)


CRTCPPMOD MODULE(DBSTCMD/UPDDBFSTC)
          SRCFILE(DBSTCMD/QCPPSRC)

CRTPGM PGM(DBSTCMD/UPDDBFSTC)
       MODULE(DBSTCMD/UPDDBFSTC)
       ENTMOD(UPDDBFSTC)
       BNDSRVPGM(QSYS/QDBSTMGR QSYS/QP0ZCPA)
       ACTGRP(*CALLER)


// Have the following in DBSTCMD/QCMDSRC(UPDDBFSTC)


CMD        PROMPT('Update DB-File Statistics')
             PARM      KWD(ASPDEV) TYPE(*SNAME) LEN(10) +
                         DFT(*) PROMPT('Library ASP device name') +
                         SPCVAL((*) (*SYSBAS))
             PARM       KWD(FILE) TYPE(FILEQUAL) PROMPT('Database file')
             PARM       KWD(MBR) TYPE(*SNAME) PROMPT('Member') +
                            DFT(*FIRST) SPCVAL((*FIRST) (*LAST))
             PARM       KWD(STCID) TYPE(*HEX) LEN(16) +
                          PROMPT('Statistics ID')
             PARM       KWD(UPDDTA) TYPE(*CHAR) LEN(12) RSTD(*YES) +
                          DFT(*IMMEDIATE) SPCVAL((*IMMEDIATE) +
                          (*BACKGROUND) (*NO)) PROMPT('Data refresh')
             PARM       KWD(AGEMODE) TYPE(*CHAR) LEN(10) RSTD(*YES) DFT(*SAME) +
                          SPCVAL((*SYS) (*USER) (*SAME)) PROMPT('Aging mode')
             PARM       KWD(BLKSYS) TYPE(*CHAR) LEN(5) RSTD(*YES) DFT(*SAME) +
                          SPCVAL((*YES) (*NO) (*SAME)) +
                          PROMPT('Block system activity')
             PARM       KWD(STCNAM) TYPE(*CHAR) LEN(30) DFT(*SAME) +
                          SPCVAL((*SAME)) PROMPT('Statistics name')


/* QUALIFIED FILE NAME */
 FILEQUAL:   QUAL       TYPE(*SNAME) LEN(10)
             QUAL       TYPE(*SNAME) LEN(10) DFT(*LIBL) +
                          SPCVAL((*LIBL) (*CURLIB) (*USRLIBL)) +
                          PROMPT('Library')


// Use the following to create the actual command


CRTCMD CMD(DBSTCMD/UPDDBFSTC)
       PGM(DBSTCMD/UPDDBFSTC)
       SRCFILE(DBSTCMD/QCMDSRC)


//--- Start of C++ Source ---------------------------------------------------

// database file statistics APIs
#ifndef QDBSTMGR_H
#include <qdbstmgr.h>
#endif

// memcpy, ...
#ifndef memory_h
#include <memory.h>
#endif

// API error code structures
#ifndef QUSEC_h
#include <qusec.h>
#endif

// job log output
#ifndef __QP0ZTRC_H
#include <qp0ztrc.h>
#endif

/*
// convert char to hex
#ifndef __cvtch_h
#include <mih/cvtch.h>
#endif
*/
```

```
// convert hex to char
#ifndef __cvthc_h
#include <mih/cvthc.h>
#endif

// exception handling
#ifndef __except_h
#include <except.h>
#endif

// resend escape message
#ifndef QMHRSNEM_h
#include <qmhrsnem.h>
#endif

#include <assert.h>

//-----------------------------------------------------------------------------
// Calculate the stats name length (quoted '"', or ' ' or '\0' terminated)
static QDBST_uint32 QdbstCmd_calcStatsNameLen
(
 const char *StatsName,
 const size_t maxLen
);
//-----------------------------------------------------------------------------
// Resignal exceptions to one above the invocation pointer given as
// communications "area" passed on the #pragma exception_handler.
static void QdbstCmd_ResignalHdlr(_INTRPT_Hndlr_Parms_T *errmsg);
//-----------------------------------------------------------------------------
int main(int argc, char *argv[])
{
 // setup exception resignal handler
    // get invocation pointer for resignal scoping
    volatile _INVPTR invPtr = _INVP(0);

    #pragma exception_handler (QdbstCmd_ResignalHdlr, invPtr, 0, _C2_MH_ESCAPE)

 // check number of parameters
 // ...

 // address (sub set of) input parameters
    const char* const ASPDeviceName   = argv[1];
    const char* const FileName        = argv[2];
    const char* const LibName         = argv[2] + 10;
    const char* const MbrName         = argv[3];
    const char* const StatsID         = argv[4];
    const char* const CollMode        = argv[5];
    const char* const AgingMode       = argv[6];
    const char* const BlockOption     = argv[7];
    const char* const StatsName       = argv[8];

 // API input
    struct InputUpdate_t
    {
       Qdbst_STIU0100_Hdr_t hdr;
       char                 Buffer[sizeof(Qdbst_KeyValue_StatsData_t) +
                                   sizeof(Qdbst_KeyValue_AgingMode_t) +
                                   sizeof(Qdbst_KeyValue_BlockOption_t) +
                                   sizeof(Qdbst_KeyValue_t) + 128];
```

```
   } InputUpdate;

// init input buffer
   memset(&InputUpdate, 0, sizeof(InputUpdate));

// set input header (fixed part)
   memcpy(InputUpdate.hdr.ASPDeviceName, ASPDeviceName, 10);
   memcpy(InputUpdate.hdr.FileName, FileName, 10);
   memcpy(InputUpdate.hdr.FileLibraryName, LibName, 10);
   memcpy(InputUpdate.hdr.FileMemberName, MbrName, 10);

   //cvtch(InputUpdate.hdr.StatisticsID, StatsID, 32);
   memcpy(InputUpdate.hdr.StatisticsID, StatsID, 16);

   InputUpdate.hdr.iFieldOffset = offsetof(InputUpdate_t, Buffer);

// determine and set dynamic input
int updData = 0;
int updAgingMode = 0;
int updStatsName = 0;
int updBlockOption = 0;

{
   unsigned int iOffset = 0;
   QDBST_uint32 iKeys = 0;

   // data refresh requested?
   if (memcmp(CollMode, "*NO", sizeof("*NO")-1))
   {
       Qdbst_KeyValue_StatsData_t *pStatsData =
           (Qdbst_KeyValue_StatsData_t *) (InputUpdate.Buffer + iOffset);

       pStatsData->iEntryLen = sizeof(Qdbst_KeyValue_StatsData_t);
       pStatsData->iDataLen = sizeof(Qdbst_StatsData_IMMEDIATE)-1;
       pStatsData->iKey = Qdbst_KeyNo_StatsData;
       memcpy(pStatsData->Data, CollMode, pStatsData->iDataLen);

       updData = 1;

       iOffset += pStatsData->iEntryLen;
       ++iKeys;
   }

   // aging mode update requested?
   if (memcmp(AgingMode, "*SAME", sizeof("*SAME")-1))
   {
       Qdbst_KeyValue_AgingMode_t *pAgingMode =
           (Qdbst_KeyValue_AgingMode_t *) (InputUpdate.Buffer + iOffset);

       pAgingMode->iEntryLen = sizeof(Qdbst_KeyValue_AgingMode_t);
       pAgingMode->iDataLen = sizeof(Qdbst_AgingMode_SYS)-1;
       pAgingMode->iKey = Qdbst_KeyNo_AgingMode;
       memcpy(pAgingMode->Data, AgingMode, pAgingMode->iDataLen);

       updAgingMode = 1;

       iOffset += pAgingMode->iEntryLen;
       ++iKeys;
   }
   // block system activity update requested?
   if (memcmp(BlockOption, "*SAME", sizeof("*SAME")-1))
   {
       Qdbst_KeyValue_BlockOption_t *pBlockOption =
           (Qdbst_KeyValue_BlockOption_t *) (InputUpdate.Buffer + iOffset);

       pBlockOption->iEntryLen = sizeof(Qdbst_KeyValue_BlockOption_t);
       pBlockOption->iDataLen = sizeof(Qdbst_BlockOption_NOBLOCK);
       pBlockOption->iKey = Qdbst_KeyNo_BlockOption;
       pBlockOption->Data[0] = (BlockOption[1] == 'Y'
                               ? Qdbst_BlockOption_BLOCK
                               : Qdbst_BlockOption_NOBLOCK);

       updBlockOption = 1;

       iOffset += pBlockOption->iEntryLen;
       ++iKeys;
   }

   // stats name update requested?
   if (memcmp(StatsName, "*SAME", sizeof("*SAME")-1))
   {
       Qdbst_KeyValue_t *pStatsNameHdr =
           (Qdbst_KeyValue_t *) (InputUpdate.Buffer + iOffset);

       pStatsNameHdr->iDataLen = QdbstCmd_calcStatsNameLen(StatsName, 30);
       pStatsNameHdr->iEntryLen = sizeof(Qdbst_KeyValue_t) +
                               ((pStatsNameHdr->iDataLen+3) & (~3U));
       pStatsNameHdr->iKey = Qdbst_KeyNo_StatsName;
       memcpy((char*) pStatsNameHdr + sizeof(Qdbst_KeyValue_t),
               StatsName, pStatsNameHdr->iDataLen);

       updStatsName = 1;

       iOffset += pStatsNameHdr->iEntryLen;
       ++iKeys;
   }

   InputUpdate.hdr.iFieldCount = iKeys;
} // end: dynamic update options

// setup feedback related data
// Note: Not required, just as an example.

   // number of keys to be requested
   const unsigned int ciFeedbackKeys = 4;

   unsigned int FeedbackKeys[ciFeedbackKeys] =
                   {Qdbst_KeyNo_RequestID,
                    Qdbst_KeyNo_AgingMode,
                    Qdbst_KeyNo_BlockOption,
                    Qdbst_KeyNo_StatsName};

   struct Fullfeedback_t
   {
    Qdbst_FeedbackArea_Hdr_t       Header;
    Qdbst_KeyValue_RequestID_t     RequestID;
    Qdbst_KeyValue_AgingMode_t     AgingMode;
    Qdbst_KeyValue_BlockOption_t   BlockOption;
    Qdbst_KeyValue_t               StatsName_Hdr;
    char                           StatsName_Data[128];
   } FeedbackArea;

// call the Update Stats API

   // force exceptions to be signalled
       Qus_EC_t ec;
       ec.Bytes_Provided = 0;

   QDBST_uint32 iInputLen = sizeof(InputUpdate);
   QDBST_uint32 iFeedbackKeys = ciFeedbackKeys;
   QDBST_uint32 iFeedbackSize = sizeof(FeedbackArea);

   QdbstUpdateStatistics(&InputUpdate,
                         &iInputLen,
                         "STIU0100",
                         &FeedbackArea,
                         &iFeedbackSize,
                         FeedbackKeys,
                         &iFeedbackKeys,
                         &ec);


// log the feedback
// Note: Not required, just as an example.
{
   assert(FeedbackArea.Header.iKeysReturned == ciFeedbackKeys);

   char szBuffer[256];

   if (updData)
   {
       cvthc(szBuffer, FeedbackArea.RequestID.Data, 32);
       szBuffer[32] = '\0';
       QpOzLprintf("REQUESTID: %s\n", szBuffer);
   }

   if (updAgingMode)
   {
       memcpy(szBuffer, FeedbackArea.AgingMode.Data,
               sizeof(Qdbst_AgingMode_SYS)-1);
       szBuffer[sizeof(Qdbst_AgingMode_SYS)-1] = '\0';
       QpOzLprintf("PREVIOUS AGING MODE: %s\n", szBuffer);
   }

   if (updBlockOption)
   {
       QpOzLprintf("PREVIOUS BLOCK OPTION: %c\n",
           FeedbackArea.BlockOption.Data[0]);
   }

   if (updStatsName)
   {
       memcpy(szBuffer, FeedbackArea.StatsName_Data,
           FeedbackArea.StatsName_Hdr.iDataLen);
       szBuffer[FeedbackArea.StatsName_Hdr.iDataLen] = '\0';
       QpOzLprintf("PREVIOUS STATSNAME: %s\n", szBuffer);
   }

} // end: log feedback
```

```
 // return success
    #pragma disable_handler
    return 0;

} // end: main
//-----------------------------------------------------------------------------
static QDBST_uint32 QdbstCmd_calcStatsNameLen
(
 const char *StatsName,
 const size_t maxLen
)
{
 QDBST_uint32 iLen = 0;

 if (StatsName[0] == '\"')
 {
    for(++iLen; (iLen < maxLen && StatsName[iLen] != '\"'); ++iLen);
    ++iLen;
 }
 else
 {
    for(;(iLen < maxLen && StatsName[iLen] != ' ' && StatsName[iLen]); ++iLen);
 }

 return iLen;
} // end: QdbstCmd_calcStatsNameLen
//-----------------------------------------------------------------------------
static void QdbstCmd_ResignalHdlr(_INTRPT_Hndlr_Parms_T *errmsg)
{
 // force errors with QMHRSNEM to be signalled, not stored
    Qus_EC_t ec;
    ec.Bytes_Provided = 0;

 // QMHRSNEM input - use format 0200 to specify "to call stack" via
 // invocation pointer
    Qmh_Rsnem_RSNM0200_t rsnm0200;

    // com area contains invocation pointer of a call stack level.
    // We will resignal to 1 (Call_Counter) above that level.
    // *PGMBDY allows us to skip ILE PEP etc.
        rsnm0200.Call_Stack_Entry = *((_INVPTR *) errmsg->Com_Area);
        rsnm0200.Call_Counter     = 1;
        memcpy(rsnm0200.Pointer_Qualifier, "*PGMBDY   ", 10);

 // resignal
    QMHRSNEM((char *)&errmsg->Msg_Ref_Key, // Message key
             &ec,                          // Error code
             &rsnm0200,                    // To call stack entry
             sizeof(rsnm0200),             // Length of to call stack
             "RSNM0200",                   // Format
             &(errmsg->Target),            // From call stack entry
             0);                           // From call stack counter

} // end: QdbstCmd_ResignalHdlr
//-----------------------------------------------------------------------------
```

List Statistics Collection

This example demonstrates how to use the List Statistics Collection API to create a user-defined table function (UDTF). Example A-4 shows the code that you need to generate the programs and the source code. When using the UDTF, make sure to pad all input parameters with blanks at the end so that they contain 10 characters each. Also be sure to use all uppercase for the names. Otherwise, the UDTF returns an error.

Possible input values are:

- *For ASP Device name*: The actual ASP name, special value *SYSBAS, or * (meaning current)
- *For library name*: The actual library name, special values *LIBL, *CURLIB, or *USRLIBL
- *For filename*: The actual file name or special value *ALL if an actual library name was provided
- *For member name*: The actual member name, special value *FIRST, *LAST, or *ALL

Example: A-4 List Statistics Collection API

```
// DESCRIPTION:  Sample UDTF LSTDBFSTC
//
// FUNCTION:     List database file statistics via a
//               user-defined table function (UDTF)
//               using the database file statistics
//               QdbstListStatistics API.
//
// Program create, function registration, and sample function call
// Have source as DBSTCMD/QCPPSRC(LSTDBFSTC)


CRTCPPMOD
    MODULE(DBSTCMD/LSTDBFSTC)
    SRCFILE(DBSTCMD/QCPPSRC)

CRTSRVPGM
    SRVPGM(DBSTCMD/LSTDBFSTC)
    MODULE(DBSTCMD/LSTDBFSTC)
    BNDSRVPGM(QSYS/QDBSTMGR QSYS/QP0ZCPA)
    EXPORT(*ALL)
    STGMDL(*SNGLVL)
    ACTGRP(*CALLER)


// SQL statements to create UDTF


DROP FUNCTION DBSTCMD/LSTDBFSTC

CREATE FUNCTION DBSTCMD/LSTDBFSTC
(
    ASPDEV VARCHAR(10),
    LIBNAM VARCHAR(10),
    FILNAM VARCHAR(10),
    MBRNAM VARCHAR(10)
)
RETURNS TABLE
(
    ASPDEV CHAR(10),
    LIBNAM CHAR(10),
    FILNAM CHAR(10),
    MBRNAM CHAR(10),
    COLNAM CHAR(10),
    CRTUSR CHAR(10),
    STALE  CHAR(1),
    AGEMOD CHAR(10),
    BLKOPT CHAR(1),
    CARDIN BIGINT,
    MFVCNT INTEGER,
    HSTCNT INTEGER,
    STCID  CHAR(32),
    STCSIZ BIGINT,
    STCNAM VARCHAR(128)
)
RETURNS NULL ON NULL INPUT
NO FINAL CALL
NO DBINFO
EXTERNAL NAME 'DBSTCMD/LSTDBFSTC(LSTDBFSTC)'
LANGUAGE C++
SCRATCHPAD
PARAMETER STYLE DB2SQL
DISALLOW PARALLEL
NOT FENCED
CARDINALITY 1000


// Sample SQL statements to use UDTF - getting stats info for all files in LIB1


SELECT *
FROM TABLE(LSTDBFSTC('*         ',
                     'LIB1      ',
                     '*ALL      ',
                     '*ALL    ')) AS A


// Sample SQL statements to use UDTF - getting info required for UPTDBFSTC for stale stats


SELECT FILNAM, MBRNAM, STCID
FROM TABLE(LSTDBFSTC('*         ',
                     'LIB1      ',
                     '*ALL      ',
                     '*ALL    ')) AS A
WHERE STALE = 'Y'


//-START OF C++ SOURCE CODE----------------------------------------------------
// Include files

// SQL UDF/UDTF support
#ifndef  SQL_H_SQLUDF
#include <sqludf.h>
#endif

// statistics APIs
#ifndef QDBSTMGR_H
#include <qdbstmgr.h>
#endif
```

```
// User space handling - generic header
#ifndef QUSGEN_h
#include <qusgen.h>
#endif

// User space handling - resolve user space pointer
#ifndef QUSPTRUS_h
#include <qusptrus.h>
#endif

// User space handling - delete  user space
#ifndef QUSDLTUS_h
#include <qusdltus.h>
#endif

// User space handling - create user space
#ifndef QUSCRTUS_h
#include <quscrtus.h>
#endif

#ifndef memory_h
#include <memory.h>
#endif

#ifndef __string_h
#include <string.h>
#endif

#ifndef __stddef_h
#include <stddef.h>
#endif

#ifndef __stdlib_h
#include <stdlib.h>
#endif

#ifndef QUSEC_h
#include <qusec.h>
#endif

#ifndef __except_h
#include <except.h>
#endif

// convert hex to char
#ifndef __cvthc_h
#include <mih/cvthc.h>
#endif

// for debug
#ifndef __QP0ZTRC_H
#include <qp0ztrc.h>
#endif

//------------------------------------------------------------------------------
// classes/structs

struct LSTDBFSTC_ListEntry_t
{
    enum { KeyIndex_MAX = 15 };
    static const QDBST_uint32 KeyNumbers[KeyIndex_MAX];

    Qdbst_STOL0100_Hdr_t            ListEntryHeader;
    Qdbst_KeyValue_ASPDeviceUsed_t  ASPName;
    Qdbst_KeyValue_FileLibUsed_t    LibName;
    Qdbst_KeyValue_FileUsed_t       FilName;
    Qdbst_KeyValue_FileMbrUsed_t    MbrName;
    Qdbst_KeyValue_OneColumnName_t  ColName;
    Qdbst_KeyValue_CreatingUser_t   CrtUser;
    Qdbst_KeyValue_AgingStatus_t    Stale;
    Qdbst_KeyValue_AgingMode_t      AgingMode;
    Qdbst_KeyValue_BlockOption_t    BlockOption;
    Qdbst_KeyValue_CurStatsSize_t   StatsSize;
    Qdbst_KeyValue_Cardinality_t    Cardinality;
    Qdbst_KeyValue_MFVCount_t       MFVCount;
    Qdbst_KeyValue_HistRCount_t     HistRCount;
    Qdbst_KeyValue_StatsID_t        StatsID;
    Qdbst_KeyValue_t                StatsName_Hdr;
    char                            StatsName_Data[128];
};

const QDBST_uint32
    LSTDBFSTC_ListEntry_t::KeyNumbers[LSTDBFSTC_ListEntry_t::KeyIndex_MAX] =
{
    Qdbst_KeyNo_ASPDeviceUsed,
    Qdbst_KeyNo_FileLibUsed,
    Qdbst_KeyNo_FileUsed,
    Qdbst_KeyNo_FileMbrUsed,
    Qdbst_KeyNo_ColumnNames,
    Qdbst_KeyNo_CreatingUser,
    Qdbst_KeyNo_AgingStatus,
    Qdbst_KeyNo_AgingMode,
    Qdbst_KeyNo_BlockOption,
    Qdbst_KeyNo_CurStatsSize,
    Qdbst_KeyNo_Cardinality,
    Qdbst_KeyNo_MFVCount,
    Qdbst_KeyNo_HistRCount,
    Qdbst_KeyNo_StatsID,
    Qdbst_KeyNo_StatsName
};

struct LSTDBFSTC_ScratchPad_t
{
    // of scratch pad, as set by SQL environment
    // keep in here, to have pointers below aligned OK
    unsigned long length;

    // keep track of how many list entries already processed
    QDBST_uint32 iCurrentListEntry;

    // Pointer to user space used for QdbstListStatistics.
    // Will be initialized in OPEN call and possibly gets updated
    // in FETCH calls, when we reach a continuation point.
    Qus_Generic_Header_0300_t *pUserSpaceStart;

    // Pointer to next list entry to process,
    // if iCurrentListEntry is less than total number of list entries
    // available in current userspace
    // (compare pUserSpaceStart->Number_List_Entries)
    LSTDBFSTC_ListEntry_t *pCurrentListEntry;

    // qualified userspace name
    char UserSpaceName[20];

}; // end: LSTDBFSTC_ScratchPad_t

//-------------------------------------------------------------------------------
// Local functions

// Handle UDTF call type OPEN
static int LSTDBFSTC_Open
(
    const char*const ASPName_in,
    const char*const LibName_in,
    const char*const FilName_in,
    const char*const MbrName_in,
    LSTDBFSTC_ScratchPad_t *pScratchPad,
    char *const SQLState_out,
    char *const SQLMsgTxt_out
);

// Handle UDTF call type CLOSE
static int LSTDBFSTC_Close
(
    LSTDBFSTC_ScratchPad_t *pScratchPad,
    char *const SQLState_out,
    char *const SQLMsgTxt_out
);

//-------------------------------------------------------------------------------
// UDTF main entry point
extern "C" void LSTDBFSTC
(
    const char          ASPName_in[10],
    const char          LibName_in[10],
    const char          FilName_in[10],
    const char          MbrName_in[10],

    char                ASPName_out[10],
    char                LibName_out[10],
    char                FilName_out[10],
    char                MbrName_out[10],
    char                ColName_out[10],
    char                CrtUser_out[10],
    char *const         pStale_out,
    char                AgingMode_out[10],
    char *const         pBlockOption_out,
    long long *const    pCardinality_out,
    int *const          pMFVCount_out,
    int *const          pHistRCount_out,
    char *const         pStatsID_out,
    long long *const    pStatsSize_out,
    char                StatsName_out[128+1],

    const short *const  pASPName_inInd,
    const short *const  pLibName_inInd,
    const short *const  pFilName_inInd,
    const short *const  pMbrName_inInd,
    short *const        pASPName_outInd,
    short *const        pLibName_outInd,
    short *const        pFilName_outInd,
    short *const        pMbrName_outInd,
    short *const        pColName_outInd,
    short *const        pCrtUser_outInd,
    short *const        pStale_outInd,
    short *const        pAgingMode_outInd,
    short *const        pBlockOption_outInd,
    short *const        pCardinality_outInd,
    short *const        pMFVCount_outInd,
    short *const        pHistRCount_outInd,
    short *const        pStatsID_outInd,
    short *const        pStatsSize_outInd,
    short *const        pStatsName_outInd,

    char  *const                    SQLState_out,
    const char *const,              // sqludf_fname
    const char *const,              // sqludf_fspecname
    char  *const                    SQLMsgTxt_out,
    LSTDBFSTC_ScratchPad_t *const   pScratchPad,
    const SQLUDF_CALL_TYPE *const   pCallType
)
{
 #pragma exception_handler(LSTDBFSTC_Error, 0, _C1_ALL, _C2_ALL, _CTLA_HANDLE )

 // init basic output
    strcpy(SQLState_out, "01H00");
    *SQLMsgTxt_out = '\0';

 // act based on call type
    switch(*pCallType)
    {
        case SQLUDF_ENUM_CALLTYPE_MINUS1:   // *OPEN
            LSTDBFSTC_Open( ASPName_in,
                            LibName_in,
                            FilName_in,
                            MbrName_in,
                            pScratchPad,
                            SQLState_out,
                            SQLMsgTxt_out);
        return;

        case SQLUDF_ENUM_CALLTYPE_PLUS1:    // *CLOSE
            LSTDBFSTC_Close(pScratchPad, SQLState_out, SQLMsgTxt_out);
        return;

        case SQLUDF_ENUM_CALLTYPE_ZERO:     // *FETCH   - fall through
        break;

        default:                            // call type not handled here
        return;

    } // end: switch


 // check for EOF
    if (pScratchPad->iCurrentListEntry >=
            pScratchPad->pUserSpaceStart->Number_List_Entries)
    {
        // NOTE: add list continuation handling code

        // return EOF
            strcpy(SQLState_out, "02000");
            return;
    }


 // fill in output from current entry
    memcpy(ASPName_out, pScratchPad->pCurrentListEntry->ASPName.Data, 10);
    memcpy(LibName_out, pScratchPad->pCurrentListEntry->LibName.Data, 10);
    memcpy(FilName_out, pScratchPad->pCurrentListEntry->FilName.Data, 10);
    memcpy(MbrName_out, pScratchPad->pCurrentListEntry->MbrName.Data, 10);
    memcpy(ColName_out, pScratchPad->pCurrentListEntry->ColName.Data, 10);
    memcpy(CrtUser_out, pScratchPad->pCurrentListEntry->CrtUser.Data, 10);
    *pStale_out =
        (pScratchPad->pCurrentListEntry->Stale.Data[0]-'0' ? 'Y' : 'N');
    memcpy(AgingMode_out, pScratchPad->pCurrentListEntry->AgingMode.Data, 10);
    *pBlockOption_out =
        (pScratchPad->pCurrentListEntry->BlockOption.Data[0]-'0' ? 'Y' : 'N');
    *pStatsSize_out   = pScratchPad->pCurrentListEntry->StatsSize.Data;
    *pCardinality_out = pScratchPad->pCurrentListEntry->Cardinality.Data;
    *pMFVCount_out    = pScratchPad->pCurrentListEntry->MFVCount.Data;
    *pHistRCount_out  = pScratchPad->pCurrentListEntry->HistRCount.Data;

    cvthc(pStatsID_out, pScratchPad->pCurrentListEntry->StatsID.Data, 32);

    memcpy(StatsName_out, pScratchPad->pCurrentListEntry->StatsName_Data,
                      pScratchPad->pCurrentListEntry->StatsName_Hdr.iDataLen);
    StatsName_out[pScratchPad->pCurrentListEntry->StatsName_Hdr.iDataLen]='\0';

    *pASPName_outInd    = 0;
    *pLibName_outInd    = 0;
    *pFilName_outInd    = 0;
    *pMbrName_outInd    = 0;
    *pColName_outInd    = 0;
    *pStatsName_outInd  = 0;
    *pCrtUser_outInd    = 0;
    *pStale_outInd      = 0;
    *pAgingMode_outInd  = 0;
    *pBlockOption_outInd= 0;
    *pStatsSize_outInd  = 0;
    *pCardinality_outInd= 0;
    *pMFVCount_outInd   = 0;
    *pHistRCount_outInd = 0;
    *pStatsID_outInd    = 0;

 // do some tracing
 /*
    QpOzLprintf("ASP/LIB/FIL/MBR (%i): %s, %s, %s, %s\n",
                pScratchPad->iCurrentListEntry,
                ASPName_out,
                LibName_out,
                FilName_out,
                MbrName_out);
 */

 // advance to next entry
    if (++(pScratchPad->iCurrentListEntry) <
            pScratchPad->pUserSpaceStart->Number_List_Entries)
```

```
        pScratchPad->pCurrentListEntry = (LSTDBFSTC_ListEntry_t *)
            ((char *) pScratchPad->pCurrentListEntry  +
                pScratchPad->pCurrentListEntry->ListEntryHeader.iEntryLen);

 // return
    strcpy(SQLState_out, "00000");
    return;

 #pragma disable_handler
 // exception handler
 LSTDBFSTC_Error:
    // indicate error state
    strcpy(SQLState_out, "58004");

    // dump some context data
    #pragma exception_handler(LSTDBFSTC_Error2, 0, _C1_ALL, _C2_ALL, _CTLA_HANDLE )

    QpOzLprintf("ASP name = %s\n", ASPName_in);
    QpOzLprintf("Lib name = %s\n", LibName_in);
    QpOzLprintf("Fil name = %s\n", FilName_in);
    QpOzLprintf("Mbr name = %s\n", MbrName_in);
    QpOzLprintf("iCurrentListEntry = %i\n", pScratchPad->iCurrentListEntry);

    system("DLTUSRSPC QTEMP/LSTDBFSTC");

    #pragma disable_handler

    LSTDBFSTC_Error2: ;

} // end: LSTDBFSTC
//-------------------------------------------------------------------------------
static int LSTDBFSTC_Open
(
    const char*const ASPName_in,
    const char*const LibName_in,
    const char*const FilName_in,
    const char*const MbrName_in,
    LSTDBFSTC_ScratchPad_t *pScratchPad,
    char *const SQLState_out,
    char *const SQLMsgTxt_out
)
{
 Qus_EC_t ec;

 // create a temporary user space
    memcpy(pScratchPad->UserSpaceName, "LSTDBFSTC QTEMP     ", 20);

    // force (unexpected) exceptions to be signalled
        ec.Bytes_Provided = 0;

    // call the Create User Space API
        QUSCRTUS((void *) pScratchPad->UserSpaceName,    // Name
                  "UDTF      ",      // extended attribute
                  0x10000,           // initial size (64K)
                  "\0",              // initial value
                  "*EXCLUDE  ",      // public authority
                  "LSTDBFSTC_OPEN",   // desc
                  "*NO       ",      // no replace.
                  &ec);
```

```
// call the list API
   // input structure with our number of keys in variable length part

   struct LSTDBFSTC_STIL0100_t
   {
       Qdbst_STIL0100_Hdr hdr;
       QDBST_uint32       keys[LSTDBFSTC_ListEntry_t::KeyIndex_MAX];
   } STIL0100;

   memset(&STIL0100, 0, sizeof(STIL0100));

   // set header
       memcpy(STIL0100.hdr.ASPDeviceName, ASPName_in, 10);
       memcpy(STIL0100.hdr.FileLibraryName, LibName_in, 10);
       memcpy(STIL0100.hdr.FileName, FilName_in, 10);
       memcpy(STIL0100.hdr.FileMemberName, MbrName_in, 10);
       STIL0100.hdr.ColumnOption = Qdbst_ColumnOption_STATSONLY;

       memcpy(STIL0100.hdr.ContinuationHandle,
              Qdbst_ContinuationHandle_FIRST,
              sizeof(STIL0100.hdr.ContinuationHandle));

       STIL0100.hdr.iFieldCount  = LSTDBFSTC_ListEntry_t::KeyIndex_MAX;
       STIL0100.hdr.iFieldOffset = offsetof(LSTDBFSTC_STIL0100_t, keys);

   // set keys
       memcpy(STIL0100.keys, LSTDBFSTC_ListEntry_t::KeyNumbers,
                               sizeof(LSTDBFSTC_ListEntry_t::KeyNumbers));

// call API
   // force (unexpected) exceptions to be signalled
       ec.Bytes_Provided = 0;

   // copy constant, so that we can pass pointer to API
       unsigned int iInputLen = sizeof(LSTDBFSTC_STIL0100_t);

   QdbstListStatistics(pScratchPad->UserSpaceName,
                       "STOL0100",
                       &STIL0100,
                       &iInputLen,
                       "STIL0100",
                       &ec);

// address user space
   // force (unexpected) exceptions to be signalled
       ec.Bytes_Provided = 0;

   QUSPTRUS(pScratchPad->UserSpaceName,
            &pScratchPad->pUserSpaceStart,
            &ec);

// update remaining scratchpad fields
   pScratchPad->pCurrentListEntry = (LSTDBFSTC_ListEntry_t *)
        (((char*) pScratchPad->pUserSpaceStart) +
           pScratchPad->pUserSpaceStart->Offset_List_Data);
   pScratchPad->iCurrentListEntry = 0;

// update status
   if (pScratchPad->pUserSpaceStart->Information_Status == 'I')
   {
         strcpy(SQLState_out, "58004");
         return -1;
    }

    strcpy(SQLState_out, "00000");
    return 0;

} // end: LSTDBFSTC_Open
//-----------------------------------------------------------------------------
static int LSTDBFSTC_Close
(
    LSTDBFSTC_ScratchPad_t *pScratchPad,
    char *const SQLState_out,
    char *const SQLMsgTxt_out
)
{
 // delete the temporary user space
 if (pScratchPad->pUserSpaceStart)
 {
    // force (unexpected) exceptions to be signalled
        Qus_EC_t ec;
        ec.Bytes_Provided = 0;

    // call the Delete User Space API
        QUSDLTUS((void *) pScratchPad->UserSpaceName,
                  &ec);
 }

 strcpy(SQLState_out, "00000");
 return 0;

} // end: LSTDBFSTC_Close
//-----------------------------------------------------------------------------
```

The result of using the UDTF is a table that shows the following values for each column of statistics for one or more tables:

- Name of the ASP
- Library name
- File name
- Member name
- Column statistics are gathered on
- User that requested collection of statistics
- Status of statistic date (stale or not stale)
- Aging of statistics is done by the system or the user
- Whether statistics collection for this table was blocked
- Size of all statistics for the table
- Cardinality
- Number of the most frequent value
- Number of groups in the histogram
- ID of the statistics
- Name of the statistics

You can download the compiled versions of these programs and commands from the Web at:

ftp://testcase.boulder.ibm.com/as400/fromibm/dbsttools.savf

B

Sample database and SQL queries

This appendix details the sample database that was used for some of the tests described in this redbook. It also lists the Structured Query Language (SQL) query statements used for some of our testing of the SQL Query Engine (SQE). Some of the queries were run against a schema with five tables as shown in Figure B-1.

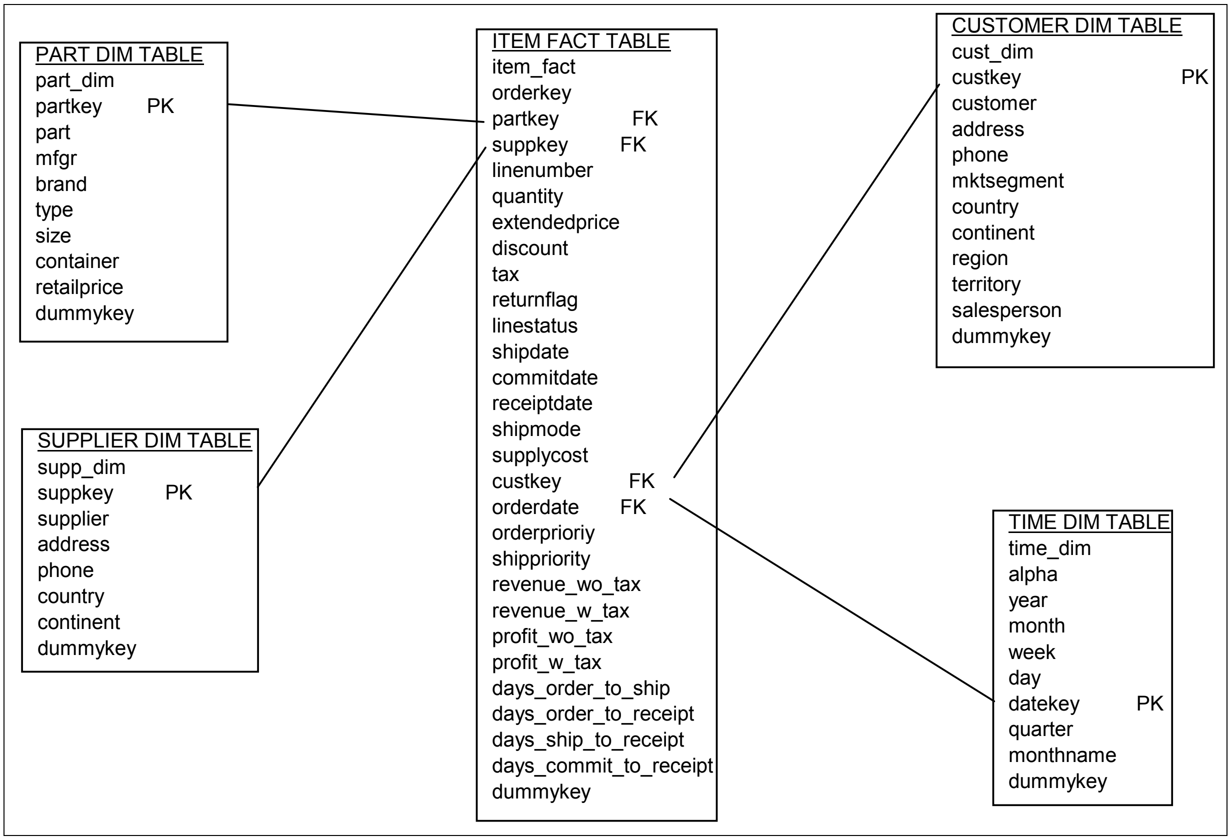


Figure B-1 Schema of STARLIB

Sample database

The database table used for the tests described in this redbook was ITEM_FACT. Its layout is shown in Table B-1. The table contained 6,001,215 rows and occupied approximately 1.6 GB. Indexes were created and dropped as needed to perform specific tests. A modified copy of the table, ITEM_FACT_SHORT, was created with some VARCHAR columns. This was used specifically to test performance of queries referencing variable length character columns.

The table sizes are:

- Table PART_DIM has 200,000 records.
- Table SUPPLIER_DIM has 10,000 records.
- Table CUST_DIM has 150,000 records.
- Table TIME_DIM has 1,450 records.
- Table ITEM_FACT has 6,001,215 records.

Table B-1 ITEM_FACT: Database table layout

Column name	Type	Length
ORDERKEY	DECIMAL	16,0
PARTKEY	INTEGER	
SUPPKEY	INTEGER	
LINENUMBER	INTEGER	
QUANTITY	DECIMAL	15,2
EXTENDED PRICE	DECIMAL	15,2
DISCOUNT	DECIMAL	15,2
TAX	DECIMAL	15,2
RETURNFLAG	CHARACTER	1
LINESTATUS	CHARACTER	1
SHIPDATE	DATE	
COMMITDATE	DATE	
RECEIPTDATE	DATE	
SHIPMODE	CHARACTER	10
SUPPLYCOST	DECIMAL	15,2
CUSTKEY	INTEGER	
ORDERDATE	DATE	
ORDERPRIORITY	CHARACTER	15
SHIPPRIORITY	INTEGER	
REVENUE_WO_TAX	DECIMAL	15,2
REVENUE_W_TAX	DECIMAL	15,2
PROFIT_WO_TAX	DECIMAL	15,2
PROFIT_W-TAX	DECIMAL	15,2

Column name	Type	Length
DAYS_ORDER_TO_SHIP	INTEGER	
DAYS_ORDER_TO_RECEIPT	INTEGER	
DAYS_SHIP_TO_RECEIPT	INTEGER	
DAYS_COMMIT_TO_RECEIPT	INTEGER	
YEAR	SMALLINT	
MONTH	SMALLINT	
QUARTER	SMALLINT	
DUMMYKEY	CHARACTER	1
EXPANDER	CHARACTER	100

SQL statements

The SQL query statements used for some of our SQE testing are listed here:

```
/* Query 1 - Simple select */
SELECT *
   FROM VETEAM01.ITEM_FACT;

/* Query 2 - Simple select with “equal” where predicate */
SELECT *
   FROM  VETEAM01.ITEM_FACT
   WHERE RETURNFLAG = 'R';

/* Query 3 - Simple select with two “equal” ANDed where predicates */
SELECT *
   FROM VETEAM01.ITEM_FACT
   WHERE RETURNFLAG = 'R' AND SHIPMODE = 'AIR';

/* Query 4 - Simple select with “equal” & “not equal” ANDed where predicates*/
SELECT *
   FROM VETEAM01.ITEM_FACT
   WHERE  RETURNFLAG = 'R' AND  RECEIPTDATE < '1997-11-7';

/* Query 5 - Select with derived source column and derived where predicate*/
SELECT ORDERKEY,  ((SUPPLYCOST*10 ) - (PROFIT_WO_TAX)) AS VARIANCE
   FROM VETEAM01.ITEM_FACT
   WHERE  (PROFIT_WO_TAX ) < (SUPPLYCOST*10);
* +++++++++++++++++++++ Queries with COUNT function ++++++++++++++++++++ */
/* Query 6 */
SELECT COUNT (*) FROM VETEAM01.ITEM_FACT;
/* Query 7 */
SELECT COUNT(*) FROM  VETEAM01.ITEM_FACT
   WHERE RETURNFLAG = 'R';
/* Query 8 */
SELECT COUNT(*) FROM VETEAM01.ITEM_FACT
   WHERE RETURNFLAG = 'R' AND SHIPMODE = 'AIR';
/* Query 9 */
SELECT COUNT(*) FROM VETEAM01.ITEM_FACT
   WHERE  RETURNFLAG = 'R' AND  RECEIPTDATE < '1997-11-7';

* ++++++++++ Queries with COUNT DISTINCT function +++++++ */
```

```
/* Query 10 */
SELECT COUNT (DISTINCT SUPPKEY) FROM VETEAM01.ITEM_FACT;
/* Query 11 */
SELECT COUNT(DISTINCT SUPPKEY) FROM  VETEAM01.ITEM_FACT
   WHERE RETURNFLAG = 'R';
/* Query 12 */
SELECT COUNT(DISTINCT SUPPKEY) FROM VETEAM01.ITEM_FACT
   WHERE RETURNFLAG = 'R' AND SHIPMODE = 'AIR       ';
/* Query 13 */
SELECT COUNT(DISTINCT SUPPKEY) FROM VETEAM01.ITEM_FACT
   WHERE  RETURNFLAG = 'R' AND  RECEIPTDATE < '1997-11-7';
/* ++++++++ Queries with DISTINCT function ++++++++++++ */
/* Query 14 */
SELECT DISTINCT SUPPKEY FROM VETEAM01.ITEM_FACT;
/* Query 15 */
SELECT DISTINCT SUPPKEY FROM  VETEAM01.ITEM_FACT
   WHERE RETURNFLAG = 'R';
/* Query 16 */
SELECT  DISTINCT SUPPKEY FROM VETEAM01.ITEM_FACT
   WHERE RETURNFLAG = 'R' AND SHIPMODE = 'AIR';
/* Query 17 */
SELECT DISTINCT SUPPKEY FROM VETEAM01.ITEM_FACT
   WHERE  RETURNFLAG = 'R' AND  RECEIPTDATE < '1997-11-7';

* ++++++++++++++ Queries with GROUP BY ++++++++++++++++ */
/* Query 18 */
SELECT SUPPKEY, SUM(REVENUE_W_TAX) FROM VETEAM01.ITEM_FACT
   GROUP BY SUPPKEY;
/* Query 19 */
SELECT SUPPKEY, PARTKEY, SUM(REVENUE_W_TAX) FROM VETEAM01.ITEM_FACT
   GROUP BY SUPPKEY, PARTKEY;
/* Query 20 */
SELECT SUPPKEY, PARTKEY, SUM(REVENUE_W_TAX) FROM VETEAM01.ITEM_FACT
   WHERE  RETURNFLAG = 'R' AND  RECEIPTDATE < '1997-11-7' AND
         SHIPMODE = 'AIR'
   GROUP BY SUPPKEY, PARTKEY;

/* +++++++++++++++ Queries with ORDER BY ++++++++++++++++ */
/* Query 21 */
SELECT * FROM VETEAM01.ITEM_FACT
   ORDER BY YEAR, MONTH;
/* Query 22 - Order by Descending */
SELECT * FROM  VETEAM01.ITEM_FACT
       ORDER BY ORDERPRIORITY DESC;
/* Query 23 */
SELECT ORDERKEY, SUPPKEY, SUPPLYCOST
   FROM VETEAM01.ITEM_FACT
   WHERE  RETURNFLAG = 'R' AND  RECEIPTDATE < '1997-11-7'
   ORDER BY SHIPDATE;
/* Query 24 */
SELECT ORDERKEY,  ((SUPPLYCOST*10 ) - (PROFIT_WO_TAX)) AS VARIANCE
   FROM VETEAM01.ITEM_FACT
   WHERE  (PROFIT_WO_TAX ) < (SUPPLYCOST*10)
   ORDER BY VARIANCE DESC;
/* Query 25 - Order by with Group by */
SELECT SUPPKEY, PARTKEY, SUM(REVENUE_W_TAX) TOTAL_REVENUE
   FROM VETEAM01.ITEM_FACT
   WHERE  RETURNFLAG = 'R' AND  RECEIPTDATE < '1997-11-7' AND
         SHIPMODE = 'AIR'
   GROUP BY SUPPKEY, PARTKEY
```

```
ORDER BY PARTKEY;

/* ++++++++++++++++ Queries with HAVING +++++++++++++++ */
/* Query 26 */
SELECT SUPPKEY, SUM(REVENUE_W_TAX) FROM VETEAM01.ITEM_FACT
   GROUP BY SUPPKEY
   HAVING SUPPKEY = 7029;
/* Query 27 */
SELECT SUPPKEY, PARTKEY, SUM(REVENUE_W_TAX) FROM VETEAM01.ITEM_FACT
   GROUP BY SUPPKEY, PARTKEY
   HAVING PARTKEY> 106170;
/* Query 28 */
SELECT SUPPKEY, PARTKEY, SUM(REVENUE_W_TAX) FROM VETEAM01.ITEM_FACT
   WHERE  RETURNFLAG = 'R' AND  RECEIPTDATE < '1997-11-7' AND
         SHIPMODE = 'AIR'
   GROUP BY SUPPKEY, PARTKEY
   HAVING COUNT (*) > 5;

/* ++++++++++++ Queries with VARCHAR Columns +++++++++++ */
/* Query 29 */
SELECT * FROM VETEAM01.ITEM_FACT_SHORT
   WHERE  QUANTITY < 30 AND SHIPMODE = 'REG AIR';
/* Query 30 */
SELECT * FROM VETEAM01.ITEM_FACT_SHORT
   WHERE  SHIPMODE != 'REG AIR' AND ORDERPRIORITY = '4-NOT SPECIFIED'
   ORDER BY SHIPMODE;

/* +++++++++++ Query with Compound Key Selection ++++++++++++++++++ */
/* Query 31 */
SELECT ORDERKEY, PARTKEY, SUPPKEY, SUPPLYCOST
   FROM VETEAM01.ITEM_FACT
   WHERE  SHIPMODE = 'TRUCK'  AND
        (PROFIT_W_TAX) <  (REVENUE_W_TAX*0.4)
   ORDER BY SHIPDATE;
/* +++++++++++++++++++++ Illogical Query ++++++++++++++++++++++++ */
/* Query 32 */
SELECT ORDERKEY,  ((SUPPLYCOST*10 ) - (PROFIT_WO_TAX)) AS VARIANCE
   FROM VETEAM01.ITEM_FACT
   WHERE  99 =100
   ORDER BY VARIANCE DESC;
```

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 190.

- *Advanced Functions and Administration on DB2 Universal Database for iSeries*, SG24-4249
- *SQL Performance Diagnosis on IBM DB2 Universal Database for iSeries*, SG24-6654
- *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503
- *Using AS/400 Database Monitor and Visual Explain To Identify and Tune SQL Queries*, REDP-0502

Other resources

These publications are also relevant as further information sources:

- *DB2 Universal Database for iSeries SQL Reference*

  http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/db2/rbafzmst.htm
- *DB2 Universal Database for iSeries - Database Performance and Query Optimization* manual

  http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp

  When you reach this site, click **Database** → **Performance and Query Optimization**.
- *Indexing and Statistics Strategies for DB2 UDB for iSeries* white paper

  http://www-03.ibm.com/servers/enable/site/bi/strategy/strategy.pdf

Referenced Web sites

These Web sites are also relevant as further information sources:

- IBM eServer iSeries Information Center

  http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp
- iDoctor for iSeries

  https://www-912.ibm.com/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

IBM Redbooks

Preparing for and Tuning the SQL Query Engine on DB2 for i5/OS

(0.2”spine)
0.17”<->0.473”
90<->249 pages

Preparing for and Tuning the SQL Query Engine on DB2 for i5/OS

Redbooks

Understand the new architecture for query processing

Learn about the data access methods and Statistics Manager

See how to monitor and tune the SQL engine

The IBM System i family, which encompasses the IBM AS/400, eServer iSeries, eServer i5, and System i5, has a successful 24-year history of satisfying hundreds of customers' requirements, thanks to its developers. DB2 for i5/OS, also known as DB2 Universal Database for iSeries, is deeply rooted in the architecture and heritage of the AS/400 and its predecessor System/38.

The database has undergone significant changes over the years to maintain its competitive advantage including fundamental changes to the structure of the database to compete successfully in the industry. In doing so, IBM Rochester launched a project to re-engineer and redesign important components of the database. The redesigned components were architected using object-oriented design concepts and then implemented using object-oriented implementation techniques.

The *query optimizer* was one of the key components that was redesigned. This IBM Redbook gives a broad understanding of the architectural changes of the database concerning the query optimizer. It explains the following concepts:

- The architecture of the query optimizer
- The data access methods used by the query optimizer
- The Statistics Manager included in V5R2
- The feedback message changes of the query optimizer
- Some performance measurements
- Changes and enhancements made in V5R4

The objective of this redbook is to help you minimize any SQL or database performance issues when upgrading to OS/400 V5R2 or V5R3 or to IBM i5/OS V5R4. Prior to reading this book, you should have some knowledge of database performance and query optimization.

SG24-6598-01 ISBN 0738494992