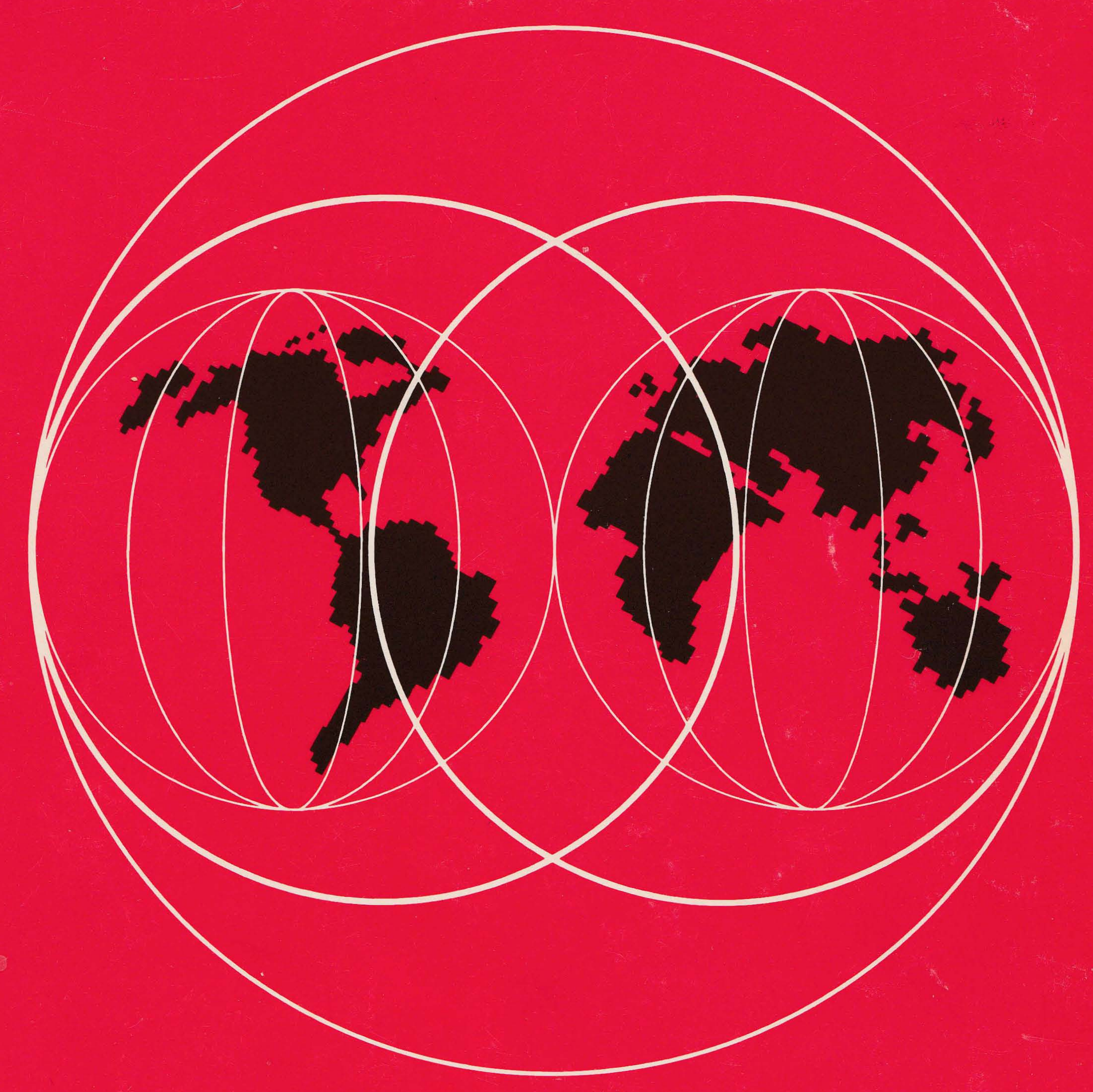


IBM

International Technical Support Centers

STRUCTURED QUERY LANGUAGE/400: A GUIDE FOR IMPLEMENTATION OS/400 RELEASE 3.0

GG24-3321-01

Structured Query Language/400: A Guide For Implementation OS/400 Release 3.0

Document Number GG24-3321-01

August 20, 1990

International Technical Support Center
Rochester,
Department 977, Building 663
Highway 52 and 37th Street NW
Rochester, Minnesota, USA

Take Note

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page iii,

Second Edition (September 1990)

This edition applies to Release 3 Modification Level 0 of the IBM Operating System/400 Licensed Program (Program 5728-SS1) and IBM Structured Query Language/400 (SQL/400) Licensed Program (Program 5728-ST1).

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below. Requests for IBM publications should be made to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Center
Dept. 977, Building 663
Highway 52 and 37th Street NW
Rochester, Minnesota, USA 55904 USA

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1990. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Special Notices

This publication is intended to compare and contrast the alternative query interfaces available on the AS/400 system, and give the reader an understanding of SQL/400, providing guidelines and examples for implementation in an application programming environment. It covers areas associated with application development with SQL/400. The information in this publication is not intended as the specification of the programming interfaces that are provided by Structured Query Language/400 for use by customers in writing programs that request or receive its services. See the PUBLICATIONS section of the IBM PROGRAMMING ANNOUNCEMENT for Structured Query Language/400 (5728-ST1).

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain 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.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific

© Copyright IBM Corp. 1990

information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

Application System/400
AS/400
C/400
COBOL/400
DB2
FORTRAN/400
IBM
MVS/XA
Operating System/2
Operating System/400
OS/2
OS/400
Personal System/2
PS/2
RPG/400
SAA
SQL/400
VM/XA

The following terms, which are denoted by a double asterisk (**) in this publication, are trademarks of other companies.

Lotus is a trademark of the Lotus Development Corporation.
Microsoft is a trademark of Microsoft Corporation.

Abstract

This document is intended for experienced AS/400 programmers, analysts, and implementers who are responsible for the creation and maintenance of application programs on the AS/400 system. It assumes the reader has an appreciation of programming in RPG/400, COBOL/400, C/400, FORTRAN/400 or PL/I as available on the AS/400 system at Release 3 Modification Level 0, and understands the AS/400 system database, the design and creation of physical and logical files, and how to design, create and maintain high-level language programs.

This document has two main purposes. First, it compares and contrasts the alternative query interfaces available on the AS/400 system. Secondly, it intends to give the reader an understanding of SQL/400, and provide guidelines and examples for implementation in an application programming environment. It covers areas associated with application development with SQL/400.

RSYS (270 pages)

© Copyright IBM Corp. 1990

Acknowledgments

The advisor for this project was:

Lamont Baker
International Technical Support Center, Rochester

The authors of this document are:

Craig Tamlin
IBM Australia

Alex Metzler
IBM Switzerland

Klaus Subtil
IBM Germany

Jim Jackson
IBM United Kingdom

This publication is the result of a residency conducted at the International Technical Support Center, Rochester.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Randy Egan
Application Business Systems Laboratory, Rochester

Mark Anderson
Application Business Systems Laboratory, Rochester

© Copyright IBM Corp. 1990

Preface

This document discusses implementation topics for users already familiar with SQL and the AS/400 system. The guide covers SQL/400 and its relationship to other query tools available on the AS/400 system, as well as how it relates to the other AS/400 data definition facilities. It describes how to integrate SQL into applications, the detailed aspects of SQL security and recovery, and SQL performance considerations. Finally, SQL/400 with regard to SAA is discussed, along with SQL/400 portability.

The information in this document is an update to the original publication, published in May 1989, and is the result of a residency at the International Technical Support Center, Rochester, Minnesota.

Purpose of This Document

The purpose of this document is to provide AS/400 implementers, system analysts and programmers responsible for the creation and maintenance of AS/400 application systems with guidelines on using SQL within their environment.

How the Document is Organized

The first section of the document compares and contrasts the alternative query interfaces available on the AS/400 system, and also the various data definition options. The second section of the document develops the reader's understanding of SQL/400, and provides guidelines and examples for application development using SQL/400 and its implementation in the programming environment.

Related Documents

For more information about using SQL statements, statement syntax and parameters, see the following manuals:

- *Programming: Structured Query Language Programmer's Guide - SC21-9609*
- *Programming: Structured Query Language Reference - SC21-9608*

The following manuals are also related to the topics covered in this document.

- *Systems Application Architecture Common Programming Interface Database Reference - SC26-4348*
- *Programming: Control Language Programmer's Guide - SC21-8077*
- *Programming: Backup and Recovery - SC21-8079*
- *DDM User's Guide - SC21-9600*
- *PL/I User's Guide and Reference - SC09-1156*
- *ANS Data Base Language - ANSI X3.135-1986*
- *ISO Standard Data Base Language SQL - ISO 9075-1987*

© Copyright IBM Corp. 1990

- *Draft Federal Information Standards SQL (1986)*
- *Programming: Data Base Guide - SC21-9659*
- *Programming: DDS Reference - SC21-9620*
- *Common Programming Interface Query Reference - SC26-4349*

Audience and Skills Level

The document assumes that the reader understands AS/400 system database concepts, along with the design and creation of physical and logical files. It also assumes a knowledge of how to design, create and maintain high-level language programs and a familiarity with either RPG/400, COBOL/400 '85, C/400, FORTRAN/400, or PL/I as available on the AS/400 system at Release 3.0.

Conventions Used in the Manual

SQL/400 offers two naming conventions:

- SQL naming convention: This uses periods (.) between the collection name and table name, as well as between the table name and column name.

  ```
  collection.table.column
  ```

- System naming convention: This uses a slash (/) between the collection name and table name and a period (.) between the table name and column name.

  ```
  collection/table.column
  ```

We have chosen to use the system naming convention in our examples. This is the default naming convention in Interactive SQL, and is likely to be chosen by most users for consistency with other AS/400 commands. It is also likely to be the preferred convention for those familiar with the System/38, since it allows the use of library lists. (See "Collection/Library Concept" on page 3-3 for further information on the use of library lists.) For this reason, we have also omitted library/collection names in most of our examples, assuming that the object would be located through the use of the library list.

Several SQL objects are equivalent to AS/400 system objects; the SQL collection is an AS/400 library (including a data dictionary, journal and journal receiver), the table is a physical file, and views and indexes are a logical files. These terms are used interchangeably throughout the text.

The example SQL statements shown in this document are based on sample tables in "Sample Tables Used in Examples" on page F-1. When used in reference to HLL programs, SQL statements have not been delimited by "EXEC SQL" and "END-EXEC" statements, as is required for COBOL/400 , RPG/400 and "EXEC SQL" only required with ";" as the statement end delimiter, for C/400, FORTRAN/400, and AS/400 PL/I.

Compile commands for COBOL, RPG, C, FORTRAN and PL/I programs using SQL are as follows:

- CRTSQLCBL
- CRTSQLRPG

- CRTSQLC
- CRTSQLFTN
- CRTSQLPLI

However, in the text these commands are referred to generically as the CRTSQLxxx commands.

Programming Examples

Most HLL references in this document are to RPG/400 and COBOL/400 and usually have equivalents in C, FORTRAN and PL/I.

Changes From Previous Release

This publication is an update to the original publication published in May 1989. The changes in this release of the manual include:

- OS/400 Query Management in Chapter 1
- Section on subquery in Chapter 3
- OS/400 Release 2.0 changes including:
  - Database to Collection
  - Changes to Interactive SQL
- Totally new performance chapter.

Contents

1. AS/400 Query Tools . . . 1-1
SQL/400 (5728-ST1) . . . 1-1
Operation . . . 1-1
Where to Use SQL/400 . . . 1-1
AS/400 Query (5728-QU1) . . . 1-2
Operation . . . 1-2
Where to Use AS/400 Query . . . 1-3
Query/38 (5728-DB1) . . . 1-3
Where to Use Query/38 . . . 1-4
Open Query File (OPNQRYF) Command . . . 1-4
Where to Use OPNQRYF . . . 1-5
AS/400 PC Support (5728-PC1) File Transfer . . . 1-5
Menu-Based Natural Language Query (MBNLQ) . . . 1-6
Where to Use AS/400 PC Support . . . 1-6
OS/400 Query Management . . . 1-6
Operation . . . 1-6
Where to Use OS/400 Query Management . . . 1-7
Summary of Functions . . . 1-8

2. AS/400 Data Definition Facilities . . . 2-1
Data Description Specifications - DDS . . . 2-1
Physical and Logical Files . . . 2-1
Physical Files . . . 2-2
Logical Files . . . 2-2
Summary . . . 2-3
Interactive Data Definition Utility - IDDU . . . 2-4
File Definitions . . . 2-4
Record Format Definitions . . . 2-5
Field Definitions . . . 2-5
SQL Data Definition Language (DDL) . . . 2-6
Summary of DDL Functions . . . 2-8
Mixing DDLs When Using SQL . . . 2-8
DDL Recommendations . . . 2-9

3. Application Programming With SQL . . . 3-1
SQL Program Design Considerations . . . 3-1
Naming Conventions . . . 3-1
SQL Naming Convention . . . 3-1
System Naming Convention . . . 3-2
Naming Conventions When Qualifying a Join . . . 3-2
Naming Convention Recommendations . . . 3-2
Collection/Library Concept . . . 3-3
SQL Catalog . . . 3-3
Catalog as an Aid to the Optimizer . . . 3-4
Indexes . . . 3-5
Views . . . 3-7
Inserting Data into Views using SQL or DFU . . . 3-7
Join, Subquery and Union . . . 3-8
Join . . . 3-8
Subquery . . . 3-13
Union . . . 3-15

© Copyright IBM Corp. 1990

Union All . . . 3-16
Compiling Applications . . . 3-17
Precompiler Process . . . 3-17
Precompiler Options . . . 3-18
Binding an Application . . . 3-20
Error Processing . . . 3-20
WHENEVER . . . 3-21
Using the SQLCA . . . 3-27
Coding a Routine for Unexpected Errors . . . 3-28
Indicator Variables . . . 3-28
Program Design Guidelines . . . 3-29
Wordiness . . . 3-30
SQL Implementation Techniques . . . 3-32
Defining Tables and Files . . . 3-33
New Table Creation . . . 3-33
Moving Existing Files/Tables into an SQL Collection . . . 3-33
IDDU File Considerations . . . 3-33
Database Creation Summary . . . 3-34
Copying and Moving Files/Tables: Journaling Considerations . . . 3-34
Indexes . . . 3-35
Join Files . . . 3-35
Changing File/Table Structure . . . 3-36
SQL Objects and Override Data Base File (OVRDBF) . . . 3-36
Using Tables and Files in High-Level Language Programs . . . 3-37
Using SQL to Access DDS- and IDDU-Defined Files and SQL Tables 3-37
File and Table Field Names in COBOL/400 . . . 3-38
Use of ALIAS in DDS Statements . . . 3-38
SQL Field Names . . . 3-39
RPG Host Variable Definition Considerations . . . 3-39
SQL Table/File Access versus HLL Table/File Access . . . 3-40
SQL INCLUDE . . . 3-41
HLL (Non-SQL) Access of SQL Tables . . . 3-42
Workstation Files . . . 3-43
Declarative SQL Statements . . . 3-43
When to Use SQL . . . 3-43
SQL and SAA . . . 3-43
Knowledge of SQL . . . 3-44
Multiple AS/400's . . . 3-44
DDM Files . . . 3-44
SQL Commitment Control . . . 3-44
Error Handling . . . 3-44
Performance of SQL versus High-Level Language Accesses . . . 3-45
Which DDL? . . . 3-45
Considerations for Data Conversion of System/36 Files . . . 3-45
System/36 File Library . . . 3-46
System/36 File Types . . . 3-46
Conversion Methods . . . 3-46
Data Conversion Steps . . . 3-47
Conclusion . . . 3-48

4. Static and Dynamic SQL . . . 4-1
Static SQL . . . 4-1
Processing Without Cursors . . . 4-1
Retrieving . . . 4-1
Updating . . . 4-2

Deleting 4-3
Inserting 4-3
Processing With Cursors 4-4
Retrieval 4-4
Updating 4-4
Deleting 4-7
Dynamic SQL 4-7
Dynamic SELECT Statements 4-7
Fixed-List SELECT statements 4-7
Varying-List SELECT statements 4-10
Dynamic Non-SELECT Statements 4-10
Statements Containing No Parameter Markers 4-10
Statements Which Contain Parameter Markers 4-11
Dynamic SQL Performance 4-13

5. SQL Performance 5-1
The Nature of Database I/O 5-2
Creating the Access Plan 5-2
Data Retrieval 5-5
Design Guidelines 5-8
Introduction 5-8
General Considerations 5-8
Database Design 5-9
Normalization 5-9
Table Size 5-11
Indexes 5-11
Matching Attributes of Join Fields 5-13
Database File Management 5-14
Journal Management 5-14
Application Design 5-15
Reusable ODP's Across Invocations 5-15
Program Design 5-17
Optimizing CPU Usage - Avoid Dynamic SQL 5-17
Optimizing Index Usage 5-18
Minimize the Number of SQL Statements 5-18
Updating via Cursor Operation 5-19
Partial Update Capable Join with Subqueries 5-20
Include Selection Columns in ORDER BY and GROUP BY 5-21
OR and IN Predicates 5-22
Index Usage with the LIKE Predicate 5-23
Specify a BETWEEN clause on keys whenever possible 5-23
Join Optimization 5-23
Avoid Numeric Conversion 5-26
Avoid String Truncation 5-26
Avoid Arithmetic Expressions 5-27
Index usage with UPDATE 5-27
Optimizing Concurrency 5-29
Use COMMIT HOLD 5-29
Optimizing I/O with Blocking 5-29
Reduce the Number of Rows Processed 5-30
Data Management Methods 5-31
Access Path 5-31
Row Selection Options 5-32
Reusability of ODP's 5-38
The Optimizer 5-44

Precompile Optimization . . . 5-44
Cost Estimation . . . 5-45
Access Plan and Validation . . . 5-47
Optimizer Decision-Making Rules . . . 5-48
Join Optimization . . . 5-48
Subquery Optimization . . . 5-50
Analyzing Performance Problems . . . 5-52
Methodology . . . 5-52
Determining Indexes . . . 5-53
Identifying Problem Code . . . 5-54
Job Log and Debug Mode . . . 5-54
Interactive SQL . . . 5-54
Identifying a Temporary Index . . . 5-56
Inefficient Indexes . . . 5-58
Work With Jobs Displays . . . 5-59
Job Trace . . . 5-59
The Nature Of Trace . . . 5-60
Running the Trace . . . 5-60
Module Names to Look for in Trace . . . 5-60
Performance Tools . . . 5-65
Positioning: SQL vs Native File Management . . . 5-66
Environment . . . 5-66
Disclaimer . . . 5-67
Native File Management vs SQL . . . 5-67
SQL in General . . . 5-68
SQL vs Keyed Files . . . 5-68
SQL vs Non-keyed Files . . . 5-68
SQL vs OPNQRYF . . . 5-69
Static vs Dynamic SQL . . . 5-69
Table of Comparisons . . . 5-70
Batch Processing . . . 5-70
The Importance of Indexes in SQL . . . 5-70
SQL Performance Enhancements in Release 3.0 . . . 5-71

6. Distributed Data Management (DDM) Considerations . . . 6-1
Interactive Access to Remote Tables . . . 6-1
CL Commands on a Remote SQL Table via DDM . . . 6-3
Programming Considerations . . . 6-3
Read-Only Remote File/Table Access . . . 6-3
Remote File/Table Updating Required . . . 6-3
Limitations and Recommendations . . . 6-4

7. SQL Commitment Control . . . 7-1
Default Values . . . 7-1
Interactive Implications under Commitment Control . . . 7-2
Batch Implications under Commitment Control . . . 7-2
Cursor Stability and *CHG . . . 7-2
Repeatable Read and *ALL . . . 7-2
Row Locking Under the Various Commitment Control Options . . . 7-3
COMMIT and ROLLBACK with HOLD Option . . . 7-6
Rollback Considerations . . . 7-10
HLL Use of Commitment Control . . . 7-10
Non-SQL Table Processing within an HLL Program . . . 7-10
SQL Table Processing within an HLL Program . . . 7-11
HLL or SQL Table Processing Summary . . . 7-11

SQL and Non-SQL Table Access in the Same HLL Program 7-12
Commitment Control on Tables/Journals in Different Collections 7-12
Ending Journaling 7-13
SQL Journaling in Different Collections 7-13
"Read Only" Access of Tables in Different Collections 7-14
Commitment Control Considerations for New Tables 7-14

8. SQL Security 8-1
Default Security Levels 8-1
Changing Authorities 8-2
Authority to the Collection 8-2
SELECT, INSERT, UPDATE and DELETE 8-2
Views 8-3
Granting/Revoking ALL Authority 8-4
Authority for Selecting on System Tables 8-5
Authority for Creating Objects in a Collection 8-6
Authority for Creating an Index 8-6
Security Recommendations 8-6
Using SQL GRANT and REVOKE 8-7
Using System Security for All Authorities 8-7
Interactive SQL and Security 8-7
Program Adoption of Authority 8-8
Commitment Control 8-8

9. Interactive SQL 9-1
Starting Interactive SQL 9-1
Interactive SQL Session 9-1
HELP Support 9-2
Session Services 9-2
Prompting and List Functions 9-3
Prompting Within SEU 9-3
Where to Use Interactive SQL and SQL Prompting 9-3

10. SQL Standards 10-1
ISO 9075-1989 and ANS X3.135-1-1989 10-1
ANS X3.135-1-1989 Integrity Enhancement 10-3
ANS X3.168-1989 Embedded SQL 10-4
FIPS 127.1 Compliance 10-4
SAA Common Programming Interface Database Reference 10-5

11. SQL/400 Portability 11-1
Data Definition Language 11-1
SAA Size Limits 11-1
SAA Functions Not Implemented in SQL/400 11-1
System Catalogs 11-2
SQL/400 Data Definition Extensions 11-2
Authorization Control Differences 11-2
Other Considerations 11-3
Data Migration 11-3
Data Conversion Considerations 11-3
Summary 11-3
Data Manipulation Language 11-4
Locking Rules 11-4
Isolation Levels 11-4
COBOL/400 and SQL Portability 11-4

Data Type Equivalence 11-4
Using COPY-DDS or SQL INCLUDE 11-5
GOBACK Statement 11-5
SQL Continuation Characters 11-5

12. SQL/400 and Relational Theory 12-1
Codd's Relational Rules 12-1
Referential Integrity 12-3
AS/400 and Referential Integrity 12-5

Appendix A. Code Example For Use of SQL WHENEVER in RPG A-1

Appendix B. Code Example For Use of SQL WHENEVER in COBOL B-1

Appendix C. Code Example For Use of SQL WHENEVER in PL/I C-1

Appendix D. Code Example For Use of SQL in C/400 . . . . . D-1

Appendix E. Code Example For Use of SQL in FORTRAN/400 . . . . . E-1

Appendix F. Sample Tables Used in Examples F-1
Inventory Table F-1
Supplier Table F-2
Quotations Table F-3
Employee Table F-4
Employee Project Account Table F-6
Department Table F-7
Project Table F-8

Index X-1

Figures

1-1. Query/38 Table 1-4
3-1. WHENEVER SQLERROR 3-26
3-2. WHENEVER NOT FOUND 3-26
3-3. SQL File/Table Access in an RPG Program 3-38
3-4. SQL File/Table Access in a COBOL Program 3-38
3-5. Use of ALIAS Parameter in a COBOL Program 3-39
3-6. HLL and SQL Access of Files/Tables 3-41
4-1. Example Screen For a Dynamic SQL Program 4-8
4-2. Program to Illustrate the Use of Dynamic Fixed-List SELECT 4-9
4-3. Program to Illustrate the Use of EXECUTE IMMEDIATE 4-10
4-4. Program to Illustrate the Use of PREPARE and EXECUTE 4-12
5-1. Methods of Accessing AS/400 Data 5-7
5-2. Non-reusable Open Data Path Mode 5-39
5-3. Reusable Open Data Path using ISVs 5-41
5-4. Reusable Open Data Path Mode 5-42
5-5. Building the Query Definition Template and Creating the Access Plan 5-45
5-6. Statement Entered by the User 5-50
5-7. Statement as Converted by the Optimizer 5-51
5-8. Statement Entered by the User 5-51
5-9. OS/400 Query Component Module Flow 5-63
12-1. Example of a Foreign Key 12-4

© Copyright IBM Corp. 1990

1. AS/400 Query Tools

With the introduction of SQL on the AS/400* system, the programmer and end user find themselves faced with a vast array of data manipulation tools which have, to some extent, overlapping functions. For querying data, a user can now use:

- SQL/400*
- AS/400 Query
- Query/38
- Open Query File command (with a high-level language program)
- AS/400 PC Support File Transfer
- OS/400* Query Management.

This chapter provides a short description of each of these tools, as well as an indication as to where each tool is best suited and the type of user it is aimed at. A summary table is also provided, cross-referencing each product and the functions that it offers.

Note that all of these Query tools use the same underlying database query support, known as the OS/400 Query component. Performance of each query tool is similar, however not identical. Performance may vary depending on the type of query being processed. For more information refer to "SQL Performance" on page 5-1.

SQL/400 (5728-ST1)

The SQL/400 support complies with the SQL defined in the Systems Application Architecture* (SAA)*. SQL may be used to define tables (physical files) and views (logical files) as well as perform a query on a table or view. Most SQL functions may be performed either interactively or in application programs written in one of the following high-level programming languages: RPG/400*, COBOL/400*, C/400*, FORTRAN/400* and AS/400 PL/I.

Operation

To execute the interactive functions of SQL, you must have the SQL/400 licensed program. However, you can create a high-level language program which uses SQL, compile it on an AS/400 system that has the SQL/400 licensed program and execute the object code from the compilation on any AS/400 with the IBM* Operating System/400* program installed.

Where to Use SQL/400

SQL/400 is a programmer facility which will allow the user to query and manipulate data in a database. The query component may be used in a high-level language program to read one or more records to be processed by other HLL statements. SQL is often the easiest form of coding database I/O requests. It is most useful for development of applications which are or may be ported to other SAA platforms and required relational database access.

© Copyright IBM Corp. 1990

Interactive SQL can also be used by programmers to test SQL statements which will later be embedded in a program. Interactive SQL is very useful for examining data stored in files. It has the same display and print facilities as AS/400 Query, however, it does not have the all of the formatting capabilities. It has an advantage over the DSPPFM (Display Physical File Member) command in that it can display packed data on the screen, whereas DSPPFM cannot. It is a very flexible debugging tool, allowing you to patch records, query data, and construct SQL statements.

It is important to understand that Interactive SQL is not intended for end user use. Prompting facilities make it easy for users unfamiliar with SQL syntax. However, there is no way of preventing users from deleting or updating data in tables, if authority is set up for use only in a HLL program. In addition, there is no easy way for retrieving and individually storing queries.

AS/400 Query (5728-QU1)

AS/400 Query is a decision support utility to obtain information and format reports from any database files that have been defined on the system.

The major features of AS/400 Query are:

- Accepts files described with DDS, IDDU or SQL
- Combines up to 32 files (physical or logical) in one query
- Selects records and fields
- Defines result fields
- Sorts using various collating sequences
- Formats columns
- Allows report break definitions and summary functions
- Outputs to screen, printer or file
- Allows detail or summary processing
- Has the ability to individually name and save queries for future use.

Operation

There are three commands for using AS/400 Query:

- The Start Query (STRQRY) command displays the Query menu. From this menu, the user may select either the "Work With Queries" option or the "Run an Existing Query" option.
- The Work with Queries (WRKQRY) command allows the user to create, change, run, copy, delete, display and print queries in interactive mode, supported by various productivity functions such as extensive help text, file and query lists, default query, and the "show" option.
- The Run Query (RUNQRY) command may be called from the command screen or from a CL program. This enables the programmer to imbed query applications into batch jobs, and is particularly useful where query output to a file needs further processing.

Where to Use AS/400 Query

AS/400 Query is a tool for end users, to allow for easy generation of reports on data which their environment allows them to access. It is most useful for "ad-hoc" queries and provides flexibility to the end user in defining and accomplishing his reporting requirements.

AS/400 Query may also be used by programmers to provide a reporting tool to the end user in a controlled environment. Programmers can insert AS/400 Query RUNQRY commands into job streams without the need to write HLL programs.

Since it is very similar to Query/36, the System/36 user will have little or no difficulty using AS/400 Query. The very high level of "ease-of-use" within AS/400 Query will particularly attract first-time users.

AS/400 Query provides one function which is not available in any other AS/400 query product: the data/text merge option. This option allows merging of data into Office/400 documents in several ways and is very likely to be used where office integration on an end-user level is required.

Query/38 (5728-DB1)

Query/38, the System/38 Query tool, is provided for compatibility for System/38 users wishing to migrate existing System/38 queries, and is therefore most likely to be used by those already using the System/38. System/38 queries can be run on the AS/400 system without this product, but cannot be modified. In order to be able to create and modify such queries using Query/38, the AS/400 System/38 Utilities program product (5728-DB1) must be installed.

Query/38 is an easy-to-use tool to produce reports rapidly on data stored in files. These files can be DDS, SQL or IDDU-created files. Query/38 provides basic headings and simple logic features. It allows selection of certain records, or certain fields from a file, as well as field creation (such as result fields), and automatic summing and averaging.

Query/38 provides three functions not available in SQL/400 or AS/400 Query. First, Query/38 will produce reports on DDM-files (that is, remote data). Second, Query/38 accepts multiple format logical files (for example, a logical file which views header and detail records from two different files). Finally, Query/38 provides the ability to create "tables" which summarize data rather than list it.[1] See Figure 1-1 on page 1-4 for an example of a table created by Query/38. Summary tables can indicate totals, averages, and cumulative values for "group by" fields. They can also indicate rankings (for example, from smallest average to largest) in separate columns, and "count" and "cumulative count" records in each group. A similar result can be achieved using SQL "COUNT", "SUM" and "AVERAGE" functions, with the exception of the "ranking" function, which is unique to Query/38. In addition, tables can be created with a major grouping field at the side, *and* a minor grouping field at the top. This function is not available through SQL.

[1] These "tables", which are reports produced by Query/38, have nothing to do with SQL tables, which are physical files.

STATE	COUNT OF RECORDS	RANK BY COUNT OF RECORDS	AVG ORDER AMOUNT	RANK BY AVG OF ORDER AMOUNT
'MI'	10	2	$165.07	2
'MN'	5	3	$175.80	1
'NY'	15	1	$ 80.54	3

Figure 1-1. Query/38 Table

Where to Use Query/38

Query/38 is a tool for end users, allowing for easy generation of reports on data which they are allowed to access. It is particularly useful for "ad-hoc" queries, and "one-off" reports, since it generates the report-writing program itself.

Query/38 may also be the easiest tool to use when summary "tables" are required, especially when major and minor classes are required. Also, it provides the fastest method of producing a report on multiple-format logical files, and on DDM files.

SQL, OPNQRYF and AS/400 Query are more powerful and flexible *programmer* tools than System/38 Query, and therefore System/38 Query is less likely to be used by programmers in the future.

Tests prove that the performance of Query/38 is notably *slower* than AS/400 Query, when performing the same function. So for AS/400 users not requiring the unique functions of Query/38, it is recommended to move to AS/400 Query purely for performance reasons.

Open Query File (OPNQRYF) Command

The Open Query File (OPNQRYF) command enables a subset of records from a file to be selected for use during a particular execution of a program. It acts as a filter between the program and the database, so that the program only retrieves records that meet the criteria specified in the OPNQRYF command.

The command must be used in conjunction with an HLL program. This processing program may be run several times on different data sets, by changing the OPNQRYF "select" criteria.

Record selection, field creation, calculations, grouping and sorting are functions of OPNQRYF. OPNQRYF is the only facility on the AS/400 which can perform advanced mathematical functions including sine, cosine, tangent, hyperbolic tangent, variance, standard deviation, and so on. (See the *Control Language Programmer's Guide* for a complete list of functions). Physical files and logical files can both be joined, and the key selected from the primary or secondary file (two features not available in DDS). Also, OPNQRYF will create an index if none exists, or use an existing one if it can.

Where to Use OPNQRYF

OPNQRYF is a programmer tool that can be used to improve efficiency of programs. It can be used where a subset of records from a file is to be used for processing, and a logical file has not been created. It sometimes provides a more efficient processing method than reading a whole file if only some records are required. For example, instead of the program reading and locking every record in a file, OPNQRYF can pass to the program only those records requested. It is also a tool to be used when the same program is to be run several times, each time using different sets of data from the same, or different, file. By changing the select conditions of the OPNQRYF, a whole new set of data can be processed.

OPNQRYF has more advanced select and calculation functions than SQL/400 and AS/400 Query. It provides the ability to update selected fields, through the associated HLL program, which AS/400 Query does not. It cannot be used in conjunction with Interactive SQL nor Query/38, since each requires its own open data path (ODP), and OPNQRYF requires all subsequent file accesses to share the same path.

AS/400 PC Support (5728-PC1) File Transfer

AS/400 PC Support provides a utility to transfer data from an AS/400 system file to a PC file using an SQL-style query interface. Complete files, portions of files, data from two or more files joined at transfer time, and grouped records from files can all be transferred from the system to a PC file, to a PC display, or to a PC printer or virtual printer. Conversely, a PC file can be transferred to an existing AS/400 file and member, or a new file or member can be created at request time. The transfer from the PC to the AS/400 system does not use an SQL-style query. Only transfers of the entire file are possible.

The transfer request prompting follows the standard SQL requests, that is:

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

When transferring data to a personal computer (RTOPC command) the result can be simply displayed on the screen, providing an Interactive SQL-type function. Since PC Support also accesses DDM files and SQL/400 does not, this function could be used in place of Interactive SQL for queries on remote files.

When transferring data to the system (RFROMPC command), a physical database file will be created at execution time, referencing fields from a PC field reference file if required, if it does not already exist. Again, the target file could be a DDM file on a remote system. The transfer facility of PC Support/400 includes a high degree of "user-friendliness" featuring contextual help text, list functions and the "show" option.

Menu-Based Natural Language Query (MBNLQ)

The PRPQ Menu-Based Natural Language Query (MBNLQ), 5799-PGP, enhances the potential use of PC Support/400 by occasional end users. MBNLQ is a front-end to the transfer facility of PC Support and allows the user to easily and automatically generate English interfaces to System/36, System/38, and AS/400 databases. The English interfaces may be customized to reflect personal preferences and can be used to interactively construct and execute English queries that retrieve information from the AS/400 databases. The query is created by selecting the query phrases from a screen using the cursor, or a mouse. An example of such a query, built by selecting from menus, follows:

```
Find part number and price and description of parts with price
greater than 5 and with color the same as 'grey' or 'orange'.
```

MBNLQ can output data in .WKS file format, useful for report formatting, tabulation, and charting by popular spreadsheet packages, such as LOTUS** 1-2-3, or Microsoft** EXCEL.

Where to Use AS/400 PC Support

AS/400 PC Support can be used by both programmers and end users. End users may find it useful for extracting information from AS/400 files to be used with PC-based applications such as spreadsheets. Programmers may create PC-based applications using the interface offered by the transfer facility to retrieve data from the host. This data can be processed further within the PC application leading to greater distributed processing.

OS/400 Query Management

OS/400 Query Management is a combination of functions, allowing access to and manipulation of data in a relational database, and the arrangement of the information into a readable (and usable) form. Services are provided in two areas, querying and report writing in order to achieve this. OS/400 Query Management provides the SAA Query/CPI Interface Support for the AS/400, as described in the manual *Common Programming Interface Query Reference SC26-4349.*

Operation

Query Management is included within OS/400. Application programs use it through a program-to-program call interface, or through Query Management objects that contain the queries, procedures and forms. OS/400 Query Management uses SQL statements for its database access. Therefore, any file on the system, defined by DDS, IDDU or SQL can be accessed from a Query Management object. Objects can be created directly, or they can be created by converting existing AS/400 Query definitions.

The source statements can be exported to other SAA platforms, in order to create appropriate Query Management objects on that system.

Where to Use OS/400 Query Management

OS/400 Query Management is a very powerful tool in a DP environment. It is not directly aimed at being an end user tool. It does not provide the end user interface necessary to input the query criteria, and then review the results. It is powerful in the ability it has to be used by the DP department as a provider of database services to the end user, and in its portability across the SAA platforms. Because OS/400 Query Management uses SQL statements, the performance implications for the access to the data, are the same as for using SQL. However, because OS/400 Query Management does more than just access the data, allowances must be made for the extra work being done.

Overhead occurs when the output is written to the printer or to a screen. As these will be the most commonly used functions of OS/400 Query Management then, the performance implications must be taken into account but generally, OS/400 Query Management will be slower. If data selected is output to a database file, performance is similar to other query tools performing this function.

However, if summary reports are required, OS/400 Query Management may be quicker than AS/400 Query for large files.

There are several instances where the use of OS/400 Query Management is useful.

- If queries or reports change on a regular basis, then because the queries are stored external to the programs, they can be changed without having to change or recompile the programs - only the Query Management objects have to be modified.
- OS/400 Query Management objects can be used on other SAA platforms. Therefore in a multi-platform environment, objects can be created on one machine type, transferred to another and processed on this other machine.
- Variable substitution is allowed in the query to allow runtime specification of the selection criteria of data. This provides greater flexibility to the user.
- The application is provided with a reduced set of error codes. This simplifies error checking.
- The database calls are handled by the Query Management objects. Therefore, application programming is simplified.

Summary of Functions

OS/400 Query Management and SQL/400 are combined in the following table, as Query Management relies upon SQL/400 runtime support for its data access function, hence the abilities of Query Management are bound by SQL/400.

Table 1-1 (Page 1 of 2). Comparison of Query Tools on AS/400

	SQL/400 and QM	AS/400 Query	Query/38	OPNQRYF	AS/400 PC Support
SQL collections, DDS files, IDDU files	Y	Y	Y	Y	Y
DDM files	N	N	Y (even non-S/38 DDM files)	Y (on same target system; with GROUP processing, both systems must be of same type - two AS/400s or two S/38s)	Y
Program described files with IDDU-linked definitions	Y(single column for record)	Y	N	N	Y
Can select order	Y	Y	Y	Y	Y
Select fields (PROJECT)	Y	Y	Y	Y (create a new format)	Y
Select records (SELECT)	Y	Y	Y	Y	Y
Create new fields (such as results)	Y	Y	Y	Y	Y
Able to join files	Y (up to 32 PFs in total)	Y (up to 32 PFs in total)	Y (only 2, second must be keyed)	Y (up to 32 PFs in total)	Y (up to 32 PFs in total)
Able to join logical files	Y (except if view uses Group by)	Y (except if query uses subtotals)	Y (not join-logical files)	Y (not with Group by)	Y (not with Group by)
Able to join file to itself	Y	Y	Y	Y	Y
When joining, key can be from any file (not allowed in DDS-LF)	Y	Y	Y	Y	Y
Equal/unequal join	Both	Both	Equal -"no match" message	Both	Both
Group processing	Y	limited	limited	Y	Y
Selection after grouping	Y	N	N	Y	Y
Total only processing	Y	Y	sub-total with final only	Y	Y
Built-in functions (like AVG, COUNT, MIN, MAX, SUM)	Y	Y	Limited	Y	Y

Table 1-1 (Page 2 of 2). Comparison of Query Tools on AS/400

	SQL/400 and QM	AS/400 Query	Query/38	OPNQRYF	AS/400 PC Support
UNION (that is, append results of 2-32 queries)	Y	N	N	N	N
DISTINCT function	Y	N	N	Y	N
Creates temporary or permanent file	Both	Creates report to file if desired	Creates report to file if desired	N	Both
Produces report	Y (QM & Interactive SQL only). ISQL limited function	Y	Y	N (unless in conjunction with HLL program)	Y
Updating of selected records	Y	N	N	Y	N
Random processing (like RPG CHAIN)	Y	N	N	Y (if not using group or unique-key functions)	N
Insert result into another file	Y	Y	N	Y (in conjunction with HLL program)	Y
Can create "tables" with ranking, major/minor classes	User programmed - Limited (no minor classes or ranking)	N	Y	N	N
HLL restrictions	COBOL'85 RPG/400 C/400 (SQL only: also FORTRAN and PL/I)	N/A	N/A	not Basic	HLL must use register handling techniques
DBCS support	Y	Y	Y	Y	N
Performance optimizing	Y	Y	Y	Y	Y
Environment	AS/400	AS/400, S/36	S/38	AS/400, S/38	PC
Programmer or End user tool	Programmer	End user	End user	Programmer	Both

2. AS/400 Data Definition Facilities

This chapter introduces the various methods for definition of externally described files on the system. With the addition of SQL, there are three main alternatives, and the decision about which to use will depend on what functions are required, and be mainly based on the nature of the applications themselves. These are:

- Data Description Specifications (DDS)
- Interactive Data Definition Utility (IDDU)
- SQL Data Definition Language Statements (SQL DDL)

This topic does not intend to cover in full detail every aspect of all three data definition facilities; rather it provides an outline of their function with relation to creation of files, and the relationship between these and the query interfaces available on the AS/400 system.

Data Description Specifications - DDS

DDS is a powerful facility for externally describing a file that is to be used by a program or system utility (licensed program product). DDS can be used for the definition of physical files (PFs), logical files (LFs), display files, printer files, and ICF files. Our discussion of DDS here refers mainly to the definition of physical and logical files.

DDS is designed for use by programmers for the complete definition of these file types. As with RPG, the programmer uses the Source Entry Utility (SEU) to create specifications which are based on positional entries in records in a source file member.

When the required file is created, the programmer issues a command that includes the name of the new file, as well as the name of the file where the DDS source statements are stored along with some additional file attributes. The creation process will take these source statements, validate them, then create the file if no errors are found.

The main advantage of an externally described file as obtained from DDS is that the programmer need not define a field (or format or file) more than once. A new program, screen, or any other file simply uses the definition already available. This reduces the chance of error in the definition of the fields, formats and files.

Physical and Logical Files

A physical file is a description of how data is seen in a program and how data is actually stored in the database. A physical file contains one record format with one or more members. This can be contrasted to a logical file, which is just a description of how data is seen in a program. The logical file contains no data, but it defines different formats over one or more physical files.

It follows that a logical file can only be created once the physical file or files, upon which the logical file is based, is created.

© Copyright IBM Corp. 1990

DDS has a variety of functions that are available in the definition of a field, record format, and file. To help illustrate the power of a DDS-defined file, it is useful to discuss some of these functions.

Physical Files

There are two basic areas in the data description specifications for the definition of the physical file attributes. The first area is for the definition of the file, format, and field names. Field characteristics (such as data type, length, and decimal places) can also be specified. In addition, one or more of the fields may be chosen as the key upon which the file is sequenced.

The second area provides for more free-format keywords for each field, format or file. These provide extra definition. Some of the main functions available are:

1. ALIAS - Allows the program to use a different name for a field
2. ALTSEQ - Specifies different collating sequence when sequencing data
3. COLHDG - Column headings as displayed when using Data File Utility (DFU), AS/400 Query, Screen Design Aid (SDA), Interactive SQL, etc.
4. COMP - Specifies data validity checking for a later defined display file
5. DESCEND - Sequence for key field, descending
6. DFT - Sets the default value for a field when a physical file field name is not referenced in an associated logical file, and an output operation occurs to the logical file
7. EDTCDE and EDTWRD - Used in the formatting of a field when outputting it to a display or printer file (for example, providing floating $ signs, date edit, other numeric edit)
8. FLTPCN - Used for the definition of the precision of a floating point field
9. RANGE - Specifies data validity checking for a later defined display file
10. REF - Used in specifying a field reference file from which to obtain field definitions
11. REFFLD - To change the field name when obtaining the definition from the field reference file specified in the PFILE parameter, or to reference a field from a file other than the one named in the REF parameter
12. REFSHIFT - Specifies data type validity checking for a later defined display file
13. TEXT - Descriptive text associated with the field or record
14. UNIQUE - Prevents the physical file from containing records with duplicate key values
15. VALUES - Specifies data validity checking for a later defined display.

Logical Files

A logical file is somewhat different from a physical file. It can provide a subset or view of information that is contained in the primary physical file or a combination of physical files. This is application-dependent, and as such the need for creation of logical files will depend on the structure of the application itself.

In relational theory, there are five main operators that are available for definition of a view of physical data:

- Sequence - ordering the records based on the value in a field or fields
- Project - choosing a subset of the fields contained in the physical file
- Select - choosing only records which match specified criteria
- Union - appending two physical files with the same record format to produce one large new file
- Join - producing a new record format based on two or more physical files with different formats and a common (join) field or fields.

Using DDS, a logical file can be built to handle all of these operators. Sequence, project, select and join form the mainstay of relational function that you see in many applications.

Like physical files, logical files also have some additional function keywords for extra definition.

The following functions are extra for logical files:

1. CONCAT - concatenates two or more physical file fields into one
2. SST - identifies a substring of a physical file field as a logical file field
3. JOIN, JREF, JFLD, JFILE, JDFTVAL, JDUPSEQ - support the join relational operator
4. RENAME - changes the name of a physical file field in your program when using the logical file.
5. ALL, DYNSLT, COMP - Select/omit keywords
6. REFACCPTH - Reference Access Path Definition
7. TRNTBL - Translation Table.

Summary

In summary, you have seen that DDS provides the following:

- Programmer's utility - entry of specifications via SEU
- Allows for the definition of physical files
- Allows for the definition of logical files and provides the five relational operators: sequence, select, project, union, and join
- Has facilities for providing extended definition to the field characteristics
- Interfaces with SDA for panel definition
- Can be used in a field reference file environment where a particular field is only required to be defined once.

For more information on Data Description Specifications, see the *Programming: DDS Reference (SC21-9620)*, and for further detail on Field Reference Files, see *Programming: Database Guide (SC21-9659-1)*.

Interactive Data Definition Utility - IDDU

The Interactive Data Definition Utility (IDDU) is an interactive tool which prompts a user to define the characteristics and content of data files on a system. It is primarily designed for compatibility with IDDU/36 so that users migrating from a System/36 to the AS/400 System/36 Environment have the same facility as they may have previously used.

IDDU is designed to be used by a knowledgeable end user or a programmer. As the prompts for input are presented in an English-like manner, and the entry screens have defined help text, an end user who has a requirement to build a file can quite easily create the needed definition, usually without having to refer to a manual.

IDDU is designed to create the external definition of fields, record formats, and files. Its full list of facilities include some extra functions:

- Create and describe data dictionaries, file, record and field definitions
- Change existing dictionaries or definitions
- Copy definitions
- Delete definitions
- Print definitions
- Rename definitions
- Show the relationship between definitions and files.

In addition, IDDU allows for some extra function to assist the user in setting up the initial content of the file:

- Create a database file
- Enter, update and delete data in this file.

The definitions created by IDDU are stored in data dictionary files. These objects are identified by QIDCTxnn, where the xnn will vary for each object (x is a character, and nn two digits). There are 17 of these. In addition, there is an object with the same name as the library, with an object type of *DTADCT (data dictionary). This is a special object containing all the definitions in this library. Entries in this object only get created once files are defined to the dictionary. All SQL collections contain these same data dictionary objects as the basis for its catalog.

File Definitions

The function for the creation of files is relatively straightforward. File definition requires the selection of one or more record formats, and optional text and long comment. IDDU follows the traditional data dictionary approach which states that there is adequate facility for the full definition of every element in detail. The long comment facility allows a description of up to 1360 characters. This is also available for the record formats, fields, and data dictionary. The text facility is for the traditional AS/400 system object text which is generally 50 characters long. Again, this is also available for the format, field and data dictionary definitions.

Record Format Definitions

Creating a record format is quite similar to that for a file, with the exception that generally only field definitions must be selected. All other facilities are available. In this environment a user is likely to select a number of fields to complete the record format.

Field Definitions

As expected, fields require more information for the correct definition of the data. The following are some of the main facilities available for field definition:

- General characteristics - name, type, size, decimals
- Column headings - for AS/400 Query output
- Long comment
- Keyboard shift - for controlling input values in SDA panels and DFU
- Alias - alternative name when definition is included into a high-level language program.

For numeric fields:

- Data type: zoned, packed, binary, floating point
- Numeric editing choices:

  Decimal point character
  Thousand separator
  Negative sign
  Print zero value
  Replace leading zeros
  Show single leading zero
  Date/time edit separator
  Edit code or edit word.

IDDU is only concerned with the creation of physical files. It has no facility for the creation of logical files with a different key sequence, or projection of fields, or to join multiple files. IDDU was originally conceived on the System/36 for use with Query/36, which was the tool that an end user has to produce logical-file-like operations on the created file. These functions have now been extended to the AS/400 system with the AS/400 Query product.

When working with an IDDU-defined file, it is important to understand that the process of changing the structure of the file is more difficult than for a DDS- or SQL DDL-externally defined file. Changing IDDU definitions for externally described files can be a lengthy procedure, although IDDU does support the creation of new versions. If a change is required, the original file must be renamed, the definition changed (to add a new field, for example), a new file must be created, then the original data copied back in. Any AS/400 Query queries defined for that files must be also updated for the new file definition. This is much more complex than for DDS or SQL-defined files.

SQL Data Definition Language (DDL)

The format of SQL data definition language for the description of data is somewhat different from that of DDS and IDDU. It uses statements which contain clauses that provide the definition information. It is generally easier to read and code SQL DDL specifications than DDS.

There are eight basic statements which can be used in the definition of data. These can be entered using Interactive SQL, or can be embedded in a program:

- CREATE COLLECTION - creates a library which can contain tables, views, and indexes. This also sets up the catalog and data dictionary objects along with a journal and journal receiver.
- CREATE TABLE - creates a table in the collection with column definitions provided.
- CREATE VIEW - creates a view on one or more tables.
- CREATE INDEX - creates an index on a table.
- DROP - deletes a collection, table, view or index and updates the catalog.
- COMMENT ON - SQL catalog comment field for the description of tables, views and columns.
- LABEL ON - SQL catalog text field for the description of tables, views and column headings.
- GRANT/REVOKE - changes table and view security rights for users.

It is worthwhile discussing these statements in more detail.

CREATE COLLECTION: This statement will create all of the same database objects that IDDU creates when building a data dictionary. These are physical and logical files which contain the field, format and file definitions. SQL catalog tables and views are created, the contents of which can be displayed by the user to examine the structure of the data. A journal and journal receiver are also created for data integrity in a commitment control environment.

CREATE TABLE: This requires the user to specify the name of the table (physical file), along with each column (field) name. The user must specify other attributes such as field data type, length, and whether the column allows a system default value. CREATE TABLE creates a non-keyed physical file and updates the catalog.

CREATE VIEW: This creates a logical file over one or more tables or views and updates the catalog. This will support the relational operators: select, project, and join. Union and sequence are not supported in a view (or a table). If required, these must be specified in the final SELECT statement for the data.

CREATE INDEX: This will create an index that can be used with a table or a view and updates the catalog. An index can be created over one or more columns. The database optimizer will decide whether the index is used based on the column name (or names) in the index. An index will generally improve the performance of a SELECT statement which requires data ordered by the field on which the index was built, or a SELECT statement based on data in that field. The index is a keyed logical file, however it cannot be queried by SQL.

DROP: This statement allows the user to delete an index, view, table, or collection. When an index, view or table is dropped, the related entry in the SQL catalog is also removed.

COMMENT ON: Allows for a 254-character description to be added to the catalog entry for a table, view, or column.

LABEL ON: Allows for a shorter descriptive text to be added to the catalog entry for a table, view, or column. The label for a table or view can be 30 characters and is the object text. The label for a column can be 20 characters and is used as the column heading when using AS/400 Query and the field heading when working with DFU or SDA on the table. For a table, this text is displayed when performing a DSPLIB command or viewing the catalog.

GRANT/REVOKE: Changes the security for tables and views. See "SQL Security" on page 8-1 for more information.

Summary of DDL Functions

Function	DDS	IDDU	SQL DDL
Target User	Programmer	Programmer or End User in S/36 Environment	Programmer
Entry Environment	Source Member	IDDU Programs & Menus	Interactive SQL or embedded in programs
User Assistance for creation of definitions	SEU Syntax Check	Interactive Msgs, Online Help	Precompiler msgs, SEU or ISQL syntax check
Documentation of created data structure	CRT command listing, DSPFD/DSPFFD	IDDU Print utility, DSPFD/DSPFFD	Select on the SQL catalog, DSPFD/DSPFFD
Creation of Physical Files	Yes	Yes	Yes
Creation of Logical Files	Yes	No[1]	Yes
Multiple Record Format Support	Yes - LF	Yes - program described PF	No
Data File Utility capable	Yes	Yes	Yes
Relational Operators Supported	All	None	All except sequence and union. Must be done with SQL DML
External Definition Available to Program	Yes	Yes (if externally described and in S/36E)	Yes
Utility Functions	None - use System Utilities	Copy, Delete, Print, Rename	Drop
Definition of field output attributes (for DSPF, DFU, Printer File - eg: Numeric Edit)	Yes (full)	Yes	No (column headings only)
Communication File Definition	Yes	No	No

Table 2-1. Comparison of Data Definition Functions on AS/400

Mixing DDLs When Using SQL

Often when using SQL the need arises for some DDL functions available only under the other DDL methods (for example, the numeric edit facilities). DDS can be used to create a file, with an extended definition for the fields. These extensions are functions that SQL cannot provide, such as formatting a date. The file can be created directly into the SQL collection library allowing the user to perform SQL table functions, but with the addition of DDS keywords to add extra edit facilities. Also an existing file can be copied into an SQL collection using the CPYF CL command with the CRTFILE(*YES) parameter. All editing specified in the DDS will be copied with the new Physical File. MOVOBJ or CRTDUPOBJ commands could also be used. In addition, the Physcial File can

[1] With IDDU, many of the logical file functions which are provided by DDS and SQL but not available directly from IDDU, are supported in a user environment with the AS/400 Query product. IDDU has the ability to create keyed physical files.

be created directly into an SQL collection, by specifying the collection name as the library on the Create Physical File (CRTPF) command.

For example, if an edit code is used to format a date or a numeric, then that edit control specification is copied to the SQL table. So, if rows are inserted into the table, and then displayed with Interactive SQL, the data will be formatted according to the edit codes specified. This gives you the ability to define functions not available directly through SQL DDL statements.

Note: Not all of the DDS keywords are compatible when moving a file into an SQL collection/library. If Default Value (DFT) and Check Validity Checking (CHECK) keywords are included on the file, when it is copied or moved an error will result (CPF-2F74).

The file that is created with this method has all the characteristics of an SQL table, once it is in an SQL collection, except for journaling. Journaling will need to be changed. See "Defining Tables and Files" on page 3-33 for more information on what changes are required.

DDL Recommendations

At this stage, it is clear that not all data description facilities provide all of the functions that are required. DDS does not create data dictionary entries for files outside an SQL collection like IDDU and SQL DDL. IDDU can be cumbersome in the way changes are made to the file structure, and does not support the logical file concept. SQL DDL does not allow for the extended column (field) definition to provide items like edit codes, multi-line AS/400 Query column headings, Screen Design Aid data validity checking, and so on.

The reader might be concerned that there is no clear-cut method that should be used. The following list is an outline of likely decisions one might make for data definition in an application environment:

1. Migrating from System/36

   If a user is migrating from a System/36 to the System/36 Environment, then the choice of data definition language will depend on whether the extra function that is available outside the System/36 Environment is to be used. The System/36 uses program-described files. If this is to continue under the System/36 Environment, then there is no concern, as no additional data definition is required. If the user wishes to make use of AS/400 Query (as Query/36 may have been used on the System/36), then IDDU is the obvious choice for further definition of data. If the user intends to convert from the System/36 Environment, and use external definitions for files, then the user may still choose IDDU (if he is already familiar with it) or DDS if other facilities like logical files are required. If SQL statements are to be used in a converted application, then either IDDU or SQL DDL can be used.

2. Applications which use only high-level language input/output (I/O) statements

   For applications which use only high-level language I/O statements, there is no need for definition in any facility other than DDS. Many customers migrating from System/38 will already be DDS users. All of the facilities that were available on the System/38 are provided, and more. AS/400 Query will work on DDS-defined files. If SQL is *not* being used, it is better that the external definition be created using DDS. The only reason a developer may

have for using SQL DDL statements in this environment is for a quicker (and more straightforward) method of creating a simple data definition for a physical file.

3. Applications which use both SQL data manipulation language and high-level language I/O statements

   In an environment where both forms of I/O statements are required, then there are other choices to make. First, some form of SQL collection is required for a full SQL application (that is if all the functions in SQL are being used including commitment control for instance, the data must be defined in an SQL collection).[2] However, as discussed before, if the definition is created with SQL DDL, it does not provide the extended definition available under the other forms.

   If the table (file) were to be used in Screen Design Aid, AS/400 Query or DFU, then SQL would not always provide the column (field) definition as required. In this instance, it is recommended to follow the directions in the topic "Mixing DDLs When Using SQL" on page 2-8. This outlines how the table (file) can be defined using *both* SQL DDL and DDS to provide both forms of definition required in each environment.

   When considering programs that use high-level language I/O statements as well as SQL DML, then it is important to realize that DDS is not required for the successful operation of the high-level language I/O component. High-level language I/O statements will operate equally as well on SQL-defined files. Not all programs in an application will necessarily use SQL DML statements. These other programs which only contain high-level language I/O statements can retrieve the SQL external definition as easily as retrieving the DDS definition.

4. Applications which use only SQL Data Manipulation Language I/O statements

   If applications are relatively simple, then often only SQL DDL is required. The programmer can create the external definition using SQL. Every reference to the file will retrieve this external definition. The programmer is then responsible for providing the extra definition of the columns (fields) if required, for instance in Screen Design Aid, printer files, and so on.

Recommendation

For most application development environments, it is recommended to create Physical Files with DDS. The created files can optionally be moved into SQL collections for full SQL usage with commitment control.

[2] But, if a user creates his own journal, then SQL and commitment control can be used outside a collection.

3. Application Programming With SQL

The use of SQL in an application environment can substantially change the nature and efficiency of the programmer's tasks on the AS/400 system. Programs with embedded SQL will look and operate differently from those which use high-level language (HLL) input/output (I/O) statements. It should be noted that to optimize performance, complex applications will probably have a mixture of HLL and SQL I/O statements. For more detailed information on performance, see "SQL Performance" on page 5-1.

This and the following chapters assume a basic knowledge of the SQL language. This can be obtained from reading *Programming: Structured Query Language Reference (SC21-9608)* and *Programming: Structured Query Language Programmer's Guide (SC21-9609)*. The following chapters build upon many of the SQL concepts in these manuals.

SQL Program Design Considerations

This section covers the topics related to the creation of applications which use SQL as the primary I/O mechanism. There are many points worth considering before embarking on a major application development task using embedded SQL. Most of these points are global to SQL in all applications, and as such require a decision to be made on the standard to be adopted throughout the application system.

Naming Conventions

The AS/400 system implementation of SQL has two types of naming conventions for SQL objects. Examples of each naming convention are detailed below. The AS/400 system naming convention provides compatibility with all other AS/400 commands. This is the default for Interactive SQL and CRTSQLxxx commands. The other is the SQL naming convention, which provides SAA compatibility with other versions of SQL (SQL/DS, DB2* and OS/2* Extended Edition).

The choice of naming convention is made with a parameter that can be used when starting Interactive SQL or when precompiling. In Interactive SQL, the naming convention cannot be changed while the session is active. Within an application program, the naming convention is standard across the program. If a program at one level calls a program at another level, then the second level program may use a different naming convention.

SQL Naming Convention

The SQL naming convention appears as:

```
COLLECTION.TABLE
```

As a collection corresponds with a single library on the AS/400 system then the COLLECTION part of the qualified name can be considered to be a library name. If no collection name is provided, then for interactive and dynamic SQL the user profile name (also called the authorization ID) of the job is used. In static (embedded) SQL, the user profile of the program owner is used. With the SQL naming convention, the library list is *never* searched to find the required table.

© Copyright IBM Corp. 1990

If you do not specify the collection name when referencing a table, and the table is not in a collection with the same name as your user profile, then the SQL statement will get an error giving SQL code SQL0204.

System Naming Convention

The system naming convention appears as:

```
LIBRARY/FILE
```

If no library name is supplied, then the library list is searched for the required table. When using the system naming convention, the normal OS/400 rules for object location apply. The library list will be searched both at precompile time and at run time.

Naming Conventions When Qualifying a Join

In a SELECT statement which requires the qualification of a column (usually in a join operation) the choice of naming convention can make the qualification nomenclature confusing. In the following example, you will join two tables, where each has two columns, and one of these columns (COLB) is common to both tables. This example uses the system (*SYS) naming convention:

```
SELECT COLA, TABLE1.COLB, COLC
FROM COLLECTION/TABLE1, COLLECTION/TABLE2
WHERE TABLE1.COLB = TABLE2.COLB
```

Notice how you require a mixture of object separators - the slash (/) and the period (.).

If you were to use the SQL (*SQL) naming convention the same example would look like:

```
SELECT COLA, COLLECTION.TABLE1.COLB, COLC
FROM COLLECTION.TABLE1, COLLECTION.TABLE2
WHERE COLLECTION.TABLE1.COLB = COLLECTION.TABLE2.COLB
```

This time the object separators are all the same, but the SQL naming convention requires the collection name to be included with the qualification of the columns. As you can see, this gets complicated.

Naming Convention Recommendations

It is worthwhile to consistently use one naming convention or the other. It is not wise to mix these across applications. The decision about which convention to use will most likely be based upon the need for compatibility with other SQL products (SQL/DS, DB2 or OS/2 Extended Edition).

It is likely that most users will choose the system naming convention. This is because there is no effort (it is the default option) in using it. It offers the same facilities as for other system commands in terms of library search list when locating objects. It is most suited to users who are familiar with the AS/400 system object structure.

Collection/Library Concept

The AS/400 system library concept provides a structure quite different from other implementations of SQL. An SQL collection equates to a library on the AS/400 system. The library or collection gives a user a place to store all objects related to a particular application. This may include programs, source files, commands, and SQL tables, views and indexes.

SQL will work well in an environment where all of the objects for an application are stored together in the same library. Using the library list concept, an application can have access to SQL objects which have been identified in a program using the system naming convention (that is without a library qualification). This provides some advantages for testing, as an application can be executed against a test table just by changing the order of the libraries in the library list (or using the OVRDBF command). However, the library list concept can only be implemented when the system naming convention is used in an application.

Therefore, it is recommended that all of the SQL objects for a single production application be kept together in one collection/library. In this case, there will be one collection/library for each application. An application that uses a table from a different collection/library has three choices for the way the program can be coded to locate it:

- Explicitly name the collection/library as a qualification to the table name
- Use the library list to locate the table, and do not qualify the table in the application
- Use the Override Database File (OVRDBF) command.

Any of these forms is acceptable depending on the application requirements.

There are some important considerations with regard to commitment control in a multi-library environment that should be noted before this is implemented. These are detailed in "Commitment Control on Tables/Journals in Different Collections" on page 7-12.

Collection vs Database: In SQL/400 release 2.0, the term *Collection* was introduced to describe what was originally called a database, meaning an SQL library. This new term is used in every instance where DATABASE was previously used. The reason for this change is to reflect a change to the SAA definition for database access. Both DATABASE and COLLECTION can be used in SQL/400 today, but COLLECTION is the recommended term for new applications.

SQL Catalog

The SQL catalog is a series of tables and views which contain information about the content and structure of an SQL collection. Each collection has its own catalog and there are also two database files in QSYS containing cross-reference information on the relationships between files and dictionaries.[1]

[1] These are called QADBFDEP and QADBXREF. They are maintained by the AS/400 system database, and as such, users need not be concerned with their operation.

The main purpose of the SQL catalog (library component) is to provide information to the users about the structure of the SQL objects in the collection.

There are six views which users may query to get information about the collection. These are:

- SYSCOLUMNS
- SYSINDEXES
- SYSKEYS
- SYSTABLES
- SYSVIEWDEP
- SYSVIEWS.

A user can access these views by normal SQL SELECT statements. A discussion and the layout of each view is contained in the *Programming: Structured Query Language Programmer's Guide (SC21-9609)*.

The information contained in the SQL catalogs is updated whenever a change is made in the content or nature of the collection itself. This update is made implicitly when a user executes another SQL statement or CL command. For instance, a CREATE TABLE will be automatically reflected in the SYSTABLES and SYSCOLUMNS catalog views.

As these catalog views are dynamically updated, they are good candidates for a simple user application to produce a map of the collection. Such an application would simply list in a structured format the definition of the tables, views, indexes, columns, and keys so that a clear understanding of the whole collection may be obtained quickly.

Updating the catalog every time a change is made in the structure of the collection could cause some performance impact. There is an overhead associated with each SQL DDL statement. However, the effect is negligible, and the maintenance of the SQL catalog should never be a bottleneck. One very good reason for this is that updates to the catalog are largely made with changes to the structure of the collection itself. Even in very complex applications, this is not something which happens with a high frequency.

Catalog as an Aid to the Optimizer

Unlike other implementations of SQL, the SQL/400 catalog is not used for assisting the optimizer in obtaining the fastest result. Due to the inbuilt nature of the AS/400 database, all of the information required by the optimizer is already held in the physical file objects themselves and other OS/400 internal objects. The SQL/400 catalog is present for compatibility with other implementations of SQL. It provides quick access to on-line documentation of the collection and may serve as the base of printed documentation of the application being supported by the collection.

Indexes

Indexes may be created to improve performance in retrieving data from SQL tables. The optimizer chooses whether a user-created index is used or not. Users only have the facility to create indexes over data in a table. Indexes, referred to by the term access path, created by DDS logical files can also be used. Indexes cannot be created on views.

The system can access an index for table data access regardless of library location or object authority for the index.

There are some techniques for deciding whether an index can be used when performing a select. If the data is requested in a certain sequence, then it is likely that SQL could use an index for retrieval. If a WHERE or GROUP BY clause is included on a SELECT statement, the optimizer will usually use an index to retrieve the required rows. When two or more tables are joined, the to-file (second table) will have an index created over the join column. Users then are likely to create indexes to improve the performance of these selections. However, it is very important that users do not create too many indexes over a table. The performance of future insert, update, or delete operations on rows in the table could be poor due to the high overhead with each operation associated with the maintenance of all the indexes.

In designing an application, it is worthwhile evaluating in detail the requirement for an index. The following points should be considered:

- Size of table, with indexes

  For selections upon a relatively small table, it is likely that the system effort involved in maintaining the user-created index through changes in the data over a period of time is greater than the system effort to scan the table to retrieve the required data.

  On the other hand, if the file is large, then the effort of the system in building an index at select time can be a major task, and therefore should be avoided. It is better in this instance to have a permanent index created by the user that is kept up to date when the data in the table changes.

  The definition of a small or large file is not exact. This will also vary according to record length. You will have to determine what is appropriate for your application.

  The *Programming: Structured Query Language Programmer's Guide (SC21-9609)* states that when the size of a table reaches 10,000 rows or more, the user should start considering the performance implications of SQL statements. At 100,000 rows or more this should be considered more actively. If an SQL statement requires the ordering of 1000 rows or more, then consider improving the performance of the ORDER BY clause. When accessing more than one table in a SELECT statement (for example, a join), then you should consider trying to improve the performance of the SELECT statement. All these comments relate to the creation of indexes.

  A user-created index can have a key length of up to 120 characters in length. This can be made up from multiple columns in any sequence. The optimizer will handle sequence requests (the WHERE clause, ORDER BY, or join to-file) by building a temporary index for a key of up to 120 bytes. Generally the optimizer will choose to scan the table, rather than build a temporary index. Temporary indexes are built only for joins and selects which specify ORDER

BY and FOR UPDATE OF. Once the user operation requires a key of greater than 120 bytes, the sequence is handled with a sort facility. The major implication of this to users is that when the sort is used, the result is read-only and will not be affected by changes. If an ORDER BY is specified for a union request or the specified union operator returns only distinct records (not UNION ALL), then a sort will be used.

For union all, a sort is not performed but a temporary file is used.

The distinct operator processing will either use a sort, or an index to help exclude duplicate records.

If a unique index exists over the specified table in the FROM clause and this request is not a join and all of the keys are being returned, then minimal processing is done since the records returned will already be distinct.

If an index has been selected by the optimizer and at least the first key of the index is returned, then a sort will not be necessary and the index will be used to perform the distinct processing.

If the optmizer can not use an index for the distinct processing, then a sort of the returned records will be performed.

- Percentage of times the number of the selection based on a sequence is required

  If an unusual sequence is desired on an infrequent basis, it is often better to display a message to the user of the application with advice to wait for the selection, than to keep an index which is mostly never used, but frequently updated by other applications.

- Time application takes to perform select

  The previous point can be contrasted with the time it actually takes to build the index. If a selection is made over a very large file, the time to build the index may be unreasonable for an interactive application and may degrade performance for the entire system while the index is being built. In this instance, a CREATE INDEX may be done before the SELECT statement (with a DROP INDEX after the selection is complete). This means that the index doesn't exist when the table is being used by other applications. It can be created at a point in the application cycle where the delay is less noticeable (like running in a background task while the user is doing something else), thus minimizing the actual selection time.

- Messages displayed on screen message line while SELECT statement is running

  When executing a SELECT statement in Interactive SQL the select process may display a message on the message line of the screen indicating that an index is being created. `"Query Running. Building Access Path for file (file-name) in (library)."` If this is the case, then it is clear that the optimizer requires an index to process the select, and the user will experience a delay while the index is being built.

  If the access path message is displayed for an extended period of time, then, depending on the other points listed here, this is a very obvious clue that a permanent index should be created. This message does not appear when a SELECT statement is embedded in an application program. The SELECT statement must be run in Interactive SQL to determine if the optimizer requires an index.

- Using the LIKE comparison operator

  Wherever possible, the use of certain matching patterns with LIKE should be limited. It can perform a full scan of the column to be compared. The optimizer will assist the LIKE function by adding an extra clause to the SELECT statement when the leading comparison characters are not wildcard (percent sign or underscore). For example, it is better to code:

```
    SELECT PARTNO, DESCR, ONHAND
    FROM INVENTRY
    WHERE DESCR LIKE ':DESCR%'
than
    SELECT PARTNO, DESCR, ONHAND
    FROM INVENTRY
    WHERE DESCR LIKE '%:DESCR%'
```

  It is a good programming guideline to code alpha-search routines that are right-end generic rather than left-and-right generic.

For more information on the use of indexes from a performance perspective, refer to "SQL Performance" on page 5-1.

Views

An SQL view provides a subset of the data in a particular table, or an amalgamation of data from more than one table. It permits a user to see data in the way he wants to, regardless of how and where it is stored. The SQL view is the equivalent of a DDS logical file without a key field, and is of type LF.

To create a view, the SQL CREATE VIEW statement is used. In creating the view, column redefinition is not required. That is, it is not necessary to give the characteristics of each column in the view (like column-name CHAR(24) NOT NULL WITH DEFAULT), but just to name the column and table from which it is taken. The characteristics of the columns are retrieved from the table when the view is created.

Inserting Data into Views using SQL or DFU

The SQL INSERT statement or a DFU program can both be used to insert rows into a view in the same way that they can be used to insert rows into a table. However, data can only be inserted into views that are based on a single table. It is not possible to insert data into views that join more than one table, as it is not possible to insert data into join-logical files either.

Views do not contain any data themselves, but look at the data in their underlying tables. You may want to insert data into a view, which will in reality be inserted into the table on which the view is based. However, since views may not contain all the columns of a table (for example the view may contain only CUSTNO and AMOUNT from a table containing CUSTNO, CUSTNAME, ADDRESS, PHONE ... AMOUNT, and so on) you may only be inserting data into *some* of the columns in the table. In such cases, it depends on how the table was created as to how the system handles the insert request.

Nulls: When creating a table, columns can be defined with the NOT NULL WITH DEFAULT clause. NOT NULL specifies that the column cannot contain null values. WITH DEFAULT specifies that the column assumes default values when a row is inserted, where no actual value is given for that column. Therefore,

when inserting data into the columns of a view, the system will insert zeros and blanks into the corresponding numeric and character columns that are defined in the table but not in the view. However, if you create the table using the NOT NULL clause, but omit WITH DEFAULT, then SQL and DFU do not know what values to insert into the non-view columns, and therefore INSERT requests are not allowed. The view in this situation is considered as read only (as it is not insert-capable, however update and delete operations may be performed).

DDS-created physical files do not have a NOT NULL WITH DEFAULT clause when creating them but they should be treated as if they have been defined in this way. They always have the possibility to use SQL or DFU to insert data into them. The system will automatically insert default values (zeros and blanks) into the fields which are not part of the logical file. The DFT (default values) keyword in DDS allows default values other than zeros and blanks (for example, the STATUS field may default to 'OK').

For more information on treating NULL values in an HLL program with embedded SQL, see "Indicator Variables" on page 3-28.

Join, Subquery and Union

Union and join are two relational operators which both deal with the combining of two or more tables or views into one logical processing set. Subquery is a function that allows a result from a inner level select to influence the outer level select results by being used as comparative data in a where clause. They all are very powerful and provide functions which in a high-level language program would be very difficult to write. The following section briefly looks at each facility, and provides some examples of their uses.

Join

The use of the join operator is the key to unlocking a correctly normalized relational database. Normalization theory has a variety of rules including: data should be atomic, without repeating groups, and have column values which are functionally dependent on a primary key value.

A normalized relation is the most robust way of storing relational data. However, if there were no techniques for easily combining the normalized data, then there would be no point in normalizing in the first place. Join provides this facility plus a lot more.

Equi-Join: The normal join used in most applications is the simple equi-join. This is where two tables are joined based on a condition that relates a column that is usually common to both. The relationship is "equal", that is the value in a column in one table is equal to the value in a column in the other table.

For instance, a customer monthly accounts table (MONTACCT) may have as columns a customer number, payment details, outstanding balances, but no "company personal" details. These might be held in another table, the customer name and address table (NAMEADDR), as such:

```
Monthly Accounts Table          Customer Name and Address Table
  Customer Number                       Customer Number
  Amount Owing This Week                Customer Name
  Amount Overdue                        Street Address
  Purchases This Week                   Shipping Address
  Purchases Month to Date               Contact Person
  Previous Order Number                 Contact Phone Number
  Month Number                          Discount Group
      - etc -                           Customer Category
                                             - etc -
```

A join on these two tables may be needed to produce a financial summary for the month for the accounting staff. To do this, a join operation would be included in the SELECT statement like the following:

```
SELECT MONTACCT.CUSTNO, CUSTNAME, AMTOWE, AMTOVR, PURCHMTD
FROM MONTACCT, NAMEADDR
WHERE AMTOVR > 10000
AND MONTACCT.CUSTNO = NAMEADDR.CUSTNO
```

The result is a set which contains five columns cross-relating the data in the two tables. Note the join actually happens by specifying columns from the two tables, and providing an association between the two tables with the expression (MONTACCT.CUSTNO = NAMEADDR.CUSTNO). We have qualified the columns here as they have the same name in both tables. However, in a join the join columns do not have to have the same name.

In understanding how a join actually works it is helpful to think of the following conceptual example. Consider that when joining the two tables above, the result is first determined based on a join between all rows in both tables. For instance, assume that the tables only contain information for three customers. The data might look like this:

- MONTACCT Table -					- NAMEADDR Table -		
CUSTNO	AMTOWE	AMTOVR	PURCHMTD	...	CUSTNO	CUSTNAME	...
23	123000	20000	154000		23	Bloggs and Co	
55	99000	12000	120000		55	Smith and Jones Co	
88	44000	16000	55000		88	Brown and Family	

If all rows in one table were joined to all rows in the other table the result would be nine rows.

```
        - MONTACCT Table -                - NAMEADDR Table -
CUSTNO  AMTOWE  AMTOVR  PURCHMTD ...  CUSTNO   CUSTNAME   ...
  23    123000  20000    154000         23   Bloggs and Co
  23    123000  20000    154000         55   Smith and Jones Co
  23    123000  20000    154000         88   Brown and Family
  55     99000  12000    120000         23   Bloggs and Co
  55     99000  12000    120000         55   Smith and Jones Co
  55     99000  12000    120000         88   Brown and Family
  88     44000  16000     55000         23   Bloggs and Co
  88     44000  16000     55000         55   Smith and Jones Co
  88     44000  16000     55000         88   Brown and Family
```

The result at this stage is called the *Cartesian Product*. Now the selection criteria is used to restrict which rows are returned.

```
WHERE AMTOVR > 10000
AND MONTACCT.CUSTNO = NAMEADDR.CUSTNO
```

In this example, all AMTOVR values exceed 10000. The resulting rows therefore are those which have matching CUSTNO values. As you would expect, there are only three rows which satisfy this criteria. Although this is a simplistic example, it can be used in determining the result from a more complex join.

Greater-than, Less-than and Non-equi Joins: Not all join operations need to be equi-joins, although this is the most common use. The greater-than, less-than and non-equi joins also have a place in data processing, although their use is considered to be somewhat statistical, like for the production of a variance analysis report. These are based upon a join comparison which relates two columns that are *not equal* or where one is *greater-than* or *less-than* the other.

For instance, consider an application which stores information in a manufacturing environment. The information is broken down by product. The (simplified) data definition may look like this:

```
   Table MFGTAB in COLLECTN
Month Number               MONTH
Product Number             PROD
Quantity On Hand           QTYONH
Quantity Manufactured      QTYMFG
Quantity Sold              QTYSOLD
```

Normally, if you add the quantity on hand at the beginning of the month to the amount manufactured and subtract the quantity sold, then the result should be the quantity on hand for the following month. But due to breakages and returns this does not always happen. The following SELECT statement will demonstrate how a non-equi join can be used to highlight the products which have an inconsistency in their figures from one month to the next. This is an example of joining a table to itself.

```
SELECT X.MONTH, X.PROD, (X.QTYONH + X.QTYMFG - X.QTYSOLD),
       Y.QTYONH
FROM MFGTAB X, MFGTAB Y
WHERE (X.QTYONH + X.QTYMFG - X.QTYSOLD) ¬= Y.QTYONH
AND X.MONTH = Y.MONTH - 1
AND X.PROD = Y.PROD
ORDER BY 2, 1
```

A detailed description of the steps involved follows:

1. The columns displayed are:
   - Month number
   - Product number
   - The calculated quantity on hand for this month
   - The actual quantity on hand for next month.
2. We join the table to itself, but give each a synonym (X and Y) so that you can identify which version you are referencing (plus it is shorter). In this instance, X and Y are known as *correlation names*.
3. We include some conditions so that the correct columns will be compared:
   - We compare the calculated quantity on hand for this month with the actual quantity on hand for next month. This is the condition you are actually wanting to test. As you can see this is a non-equal condition.
   - We make sure that X represents this month, and Y represents next month. At first, this may be difficult to follow. Essentially it assigns values to X and Y so that X = Y - 1.
   - We ensure that we are only comparing the same products from month to month.
4. We sequence the output by month within a product.

In Interactive SQL, the output would look something like this:

```
....:...10....:...20....:...30....:...40
MONTH  PROD  Numeric Expression  QTYONH
   1    10            1,900       3,900
   2    10            8,400       2,900
   1    20            3,900         900
   2    20            2,100-      3,900
********  End of data  ********
```

You can see that using a non-equi join can be complicated, especially when joining a table over itself. Even though difficult to understand, sometimes a complex SQL SELECT statement is the most efficient method of coding such a function. (Consider the code required to do this in a high-level language.)

Inner and Outer Join: What you have considered in the examples above is a join known as an *inner* join. An inner join is a join that occurs where rows from say two tables are joined *only* when there is a match between the related columns. For every row in the first table, a row is added to the result set when a match (or the stated relationship if not equal) exists between the compared columns in each table.

Consider the customer monthly accounts and name-and-address master example from before. It produces a joined result so that the financial summary can be generated. It matches one row from the accounts table and one row from the name-and-address table. If you wanted a similar report, but for all customers (hence removing the AMTOVR > 10000 condition), the SELECT statement may look like this:

```
SELECT MONTACCT.CUSTNO, CUSTNAME, AMTOWE, AMTOVR, PURCHMTD
FROM MONTACCT, NAMEADDR
WHERE MONTACCT.CUSTNO = NAMEADDR.CUSTNO
```

This would work as expected, with one joined row for each customer.

However, if you had a new customer who has no previous history or has not made any purchases, then there may be no information for that customer stored on the monthly accounts table, but the name and address details will be in the name-and-address master. An inner join will produce a result which excludes that customer from the final report. What you would really like to do is generate a message saying that the customer has no transactions on record.

We might wish to accept null or default values for the monthly accounts table columns - which could be intercepted by the HLL program, and display an appropriate message. Thus what you want is a row returned *even if* there is not a corresponding row in the other table. This is called an *outer* join.

Outer join is not supported on AS/400 SQL.[2] With subquery support, there is a technique for providing the result from an outer join, using a union operator with the SELECT. An example of this is:

```
SELECT MONTACCT.CUSTNO, CUSTNAME, AMTOWE, AMTOVR, PURCHMTD
FROM MONTACCT, NAMEADDR
WHERE MONTACCT.CUSTNO = NAMEADDR.CUSTNO
UNION ALL
SELECT NAMEADDR.CUSTNO, CUSTNAME, -1,0,0
FROM MONTACCT, NAMEADDR
WHERE NAMEADDR.CUSTNO NOT IN (SELECT CUSTNO
                              FROM MONTACCT)
```

The subquery portion of this statement is the secondary SELECT statement following the IN predicate. Basically a subquery implies that a query is run within a query. This example gives the reader an idea of the nature of complexity of performing the outer join. The idea of IN in the example above is

[2] OPNQRYF will support a partial outer join. Query/38 will also provide the function as given by the example, but this is *not* an SQL-like query.

to select all the customer numbers from the MONTACCT table and then only include the numbers of the customers who do not appear in the MONTACCT table.

The statement could be read as follows:

1. Do the initial join selection for all customers who have a row in each table.
2. Do a union with the following SELECT statement.
3. Select all customers from NAMEADDR who do not have a matching row in the MONTACCT table.
4. In order for the union to work, the columns must match; therefore you return "dummy" values for the MONTACCT columns. One of these dummy values is a negative (as this could never be a real value). We do this so you can differentiate this row (without a match) from another row (that has a match).

In Interactive SQL, the final result might be:

```
 ...:...10....:...20....:...30....:...40....:...50....:...60..
CUSTNO  CUSTNAME                 AMTOWE       AMTOVR     PURCHMTD
  23    Bloggs and Co.          123,000       20,000      154,000
  55    Smith and Jones Co       99,000       12,000      120,000
  66    Gary's Used Cars              1-           0            0
  88    Brown and Family         44,000       16,000       55,000
********  End of data  ********
```

It would be relatively easy now for a program to identify which rows in the joined table do not have a match in the MONTACCT table, and print a message in place of the other fields.

Note: You may want to avoid the union because it will require temporary results which impacts open/close performance. Running the queries separately will give the same results and possibly better performance.

See the *Programming: Structured Query Language Programmer's Guide (SC21-9609)* for more information on join.

Subquery

Subqueries are discussed in the *Programming: Structured Query Language Programmer's Guide (SC21-9609)* in detail. That is the reason why this book contains only general information on this topic. The above mentioned manual illustrates the use of subqueries with different variations and examples.

Subquery - or *'nested* SELECT' expands the complexity of queries which can be defined with a single SQL statement. The purpose is to supply information needed to qualify a row or a group of rows. A subquery will typically be part of the WHERE clause, but may also appear in the HAVING clause of SELECT, UPDATE and DELETE statements. It is of the form *operand operator (subquery)*. The *outer-level* SELECT may be part of a DECLARE CURSOR, CREATE VIEW or INSERT statement. Let's look at an example to illustrate the terminology of inner- and outer select, where we are selecting all the female employees, who's salary is higher than the average salary of all employees (both sexes):

```
DECLARE C1 CURSOR FOR
SELECT lastname, firstnme, salary
FROM   templ
WHERE  salary >
       (SELECT avg(salary)
        FROM   templ)
```

This form of the SELECT statement is processed in two steps.

The inner-level SELECT will be evaluated and executed in the first step and results in an aggregate value, in this example the average salary of all employees. The aggregate value is the result table of the inner-level SELECT statement.

In the second step, the outer-level SELECT is processed against the result table of the inner-level SELECT.

In the above example, the subquery returned ONE value. Except subqueries using the EXISTS operand, all other syntactically correct subqueries produce a one-column result table. This means that all inner-level SELECT lists refer to a single column or to an expression or function with a single result. The result table can contain zero, one or more rows.

When the result table can contain more than one row, you must use one of the keywords ALL, SOME, ANY or IN in the operator of your WHERE clause. The following examples show the syntax of these keywords:

```
WHERE   operand   < ALL   (subquery)
```

- The operand must be lower than all values in the result table of the inner-level SELECT.

```
WHERE   operand   < ANY (or SOME)   (subquery)
```

- The operand must be lower than at least one value in the result table of the inner-level SELECT.

```
WHERE   operand   IN  (subquery)
```

- The operand must be among the values returned by the inner-level SELECT.

As mentioned before, the EXISTS keyword can be used in a subquery to test for the existence of a row or rows satisfying the search condition of the inner-level SELECT. The inner-level SELECT, linked to the outer-level with the EXISTS keyword, will not return a value. The search condition resolves to true, if the subquery result table contains rows, the condition becomes false if the result table is empty. The following example illustrates the effect of EXISTS and the use of a *correlated* subquery. The query returns all employees who are not manager (that is, the employee number should not be found in the department table).

```
DECLARE C1 CURSOR FOR
SELECT empno, lastname, workdept
FROM templ xxx
WHERE NOT EXISTS ( SELECT *
                   FROM tdept inner
                   WHERE mgrno = xxx.empno )
```

Other than the previously discussed subqueries, where the subqueries have been executed once and the resulting value has been substituted into the WHERE clause, in some queries it is necessary to evaluate the subquery for each row in the outer-level SELECT. As mentioned before, this type of query is called a *correlated* subquery. A correlation name appears next to the table name in the FROM clause of the outer-level SELECT, and is used to qualify a column name in the subselect with the outer-level table name. The correlation names are the words *inner* and *xxx* that appear beside the tables names in the FROM clause in the example above. The following example shows the use of a correlated subquery with the EXISTS keyword in a DELETE statement.

```
DELETE
FROM tdept x
WHERE mgrno = ' '
  AND NOT EXISTS ( SELECT *
                   FROM templ
                   WHERE deptno = x.deptno )
```

This example deletes all departments from the department table with no manager and employees.

SQL/400 allows nesting down to 32 levels, but keep in mind that every level down has an performance impact. A maximum of 32 tables can be referred to in an SQL statement. When designing your subquery, you should examine each level for performance improvement by providing permanent indexes.

See the *Programming: Structured Query Language Programmer's Guide (SC21-9609)* and the *Programming: Structured Query Language Reference (SC21-9608)* for more details.

Union

Union combines SELECT statements on two or more tables or views into a processing set with the same layout. The original tables or views do not have to have the same layout, and do not have to have all the same fields. The main restrictions with a union are that the results from each SELECT statement must have the same number of columns, and each of the corresponding columns must be of compatible [3] data type. The following example shows a union which does *not* have columns of compatible data type:

```
SELECT NAME, SALARY FROM TABLE1
UNION
SELECT SALARY, NAME FROM TABLE2
```

[3] All character fields are compatible and all numeric fields are compatible, but character is not compatible with numeric.

Assuming the salary column is a numeric, and the name column is character, then this union will not work.

However, a union may be executed over two similar type fields that have a different length, either numeric or character:

```
SELECT BIGNAME FROM TABLE1
UNION
SELECT SMALLNAME FROM TABLE2
```

The resulting set would be a single column as large as the BIGNAME column. Unions with numeric columns that have a size mismatch will also produce a result column which is the greater of the two columns. There are some extra considerations that should be noted with regard to numeric columns if they are of a different numeric type (for example small integer versus large integer). These are covered in the *Programming: Structured Query Language Programmer's Guide (SC21-9609).*

Union All

UNION ALL is a variation of the UNION operator. UNION will produce a set which has no duplicate rows. UNION ALL will not eliminate these duplicate rows. Depending on the requirements for a union, this may or may not be important.

One instance where UNION ALL is often used is in merging two separate tables that are date-related; for instance, merging this month's production results with the production results of this time last year to produce a report on the average result by product. Last year's results are no longer kept in the current production table.

In this instance, a primary column would be the date (or month) of production, and the other columns would contain information related to the type of production. For example:

```
SELECT MONTH, PROD, QTYONH, QTYMFG, QTYSOLD,
       (QTYONH+QTYMFG-QTYSOLD)
FROM MFGTAB
WHERE MONTH = :MONTHNUM
UNION ALL
SELECT MONTH, PROD, QTYONH, QTYMFG, QTYSOLD,
       (QTYONH+QTYMFG-QTYSOLD)
FROM MFGHIST
WHERE MONTH = :MONTHNUM

    An HLL program would then average the QTY
    values based on the PROD field.
```

The UNION ALL is used instead of UNION here in case the values are the same (from one year to the next).

The union operator works by doing the two (or more) selections separately, then combining the results. A temporary interim table is built to contain the results of the first select, and the results of successive selections are added to that.

The SELECT statement specified when performing a union cannot contain an ORDER BY clause with specific column names specified. This is because the column names may not be the same for both SELECT statements. When requiring the result set for a union to be ordered, the ORDER BY clause must give the number of the column in the SELECT statement list. In the above example, adding:

```
ORDER BY 6
```

at the end of the statement would produce the result set sequenced by the QTYONH+QTYMFG-QTYSOLD column. The ORDER BY clause must be placed after the second SELECT. It applies to the entire result set, not just the interim result of the second SELECT.

See the *Programming: Structured Query Language Programmer's Guide (SC21-9609)* for more information on union.

Compiling Applications

This part discusses the tasks related to compiling HLL programs with embedded SQL. The first section discusses what happens when a program is precompiled. The second section discusses some of the more important precompiler options. These are important to any programmer wanting to correctly understand the nature of SQL and how it fits into a high-level language (HLL) environment.

Precompiler Process

To compile a program which has embedded SQL statements in it, the SQL precompiler must be run, rather than the HLL compiler. The HLL compiler will not recognize the embedded SQL statements that the programmer codes. The precompilation step will take the embedded SQL statements and replace them with HLL calls to QSQROUTE, an SQL run time module. The content of the SQL statement as embedded by the programmer will determine the nature and number of parameters passed to QSQROUTE.

To get a better appreciation of this process, compare a compiled listing of a program with embedded SQL to the original source code.

The sequence of events that take place when creating an SQL program are as follows:

1. Programmer enters source program with embedded SQL (using SEU).
2. Programmer calls the precompiler with the command CRTSQLxxx.[4]
3. Precompiler takes the source code, and comments out all of the embedded SQL statements. These are validated, and if there are no errors, they are replaced by calls to QSQROUTE. The precompiler places this in a temporary source file in library QTEMP. If there are errors, an error report is generated, and the compile process ends.

[4] There are five precompilers available for the AS/400 system. xxx represents CBL for a COBOL compile, RPG for an RPG/400 compile, C for a C/400 compile, FTN for a FORTRAN/400 compile and PLI for a PL/I compile.

4. The HLL compiler is then called, and is run against the source code in the temporary source file.[5]

5. The program object is created if there are no HLL errors.

Precompiler Options

Most of the precompiler options are relatively straightforward. But there are a few that should be discussed in more detail.

1. COMMIT Parameter

   The default compiler option for the COMMIT parameter is *CHG. This implies that all rows in a table that are updated, deleted, or inserted are locked until the transaction is committed or rolled back. Also, a program that is compiled under *CHG (or *ALL) cannot contain SQL DDL statements (for example CREATE, DROP, GRANT). These can only be executed under a commitment control level of *NONE.

   This is a major source of frustration for programmers who do not require commitment control. *NONE must be explicitly specified each time the program is precompiled.

   You will need to decide upon a commitment control strategy for the application or program, and ensure that the correct parameter is included when precompiling. This applies to all precompilers: RPG, COBOL, C, FORTRAN and PL/I. See "Rollback Considerations" on page 7-10 and "SQL Commitment Control" on page 7-1 for more information on commitment control.

2. SQL Precompile Listing

   Be aware that if the precompile process is successful, no SQL precompile listing will be generated by default. If this listing is required, then the *SRC option must be specified in the OPTION parameter.

   If errors do occur in the precompilation phase, then only the lines in error will be listed on the error report. If a full program listing is required to assist in debugging, then this option should be used.

3. SQL Precompile Only

   If the language compile is not required and if only the precompile process is to be executed, the OPTION(*NOGEN) parameter can be specified. This could be useful when performing an early compile of skeleton code to ensure that all fields are correctly defined, but no other logic should be checked. It is also useful if anything other than the compiler defaults are desired. For example, if you want to use OPTION(*SRC) on the compiler, you could first pre-compile using OPTION(*NOGEN) and then, if all was correct, use OPTION(*SRC) and the file QSQLTEMP in the QTEMP library on the CRTxxxPGM commands.

   If you are using SQL in embedded in C/400 the CRTSQLC command does not allow the input of any optional parameters such as *DEBUG or *SRC. To use the debug function in C/400-SQL programs, for instance, you can create your own CRTCPGM command with *DEBUG as the default value and put it in a library ahead of QCC in your library list. For interactive compilations you can specify *NOGEN in the CRTSQLC command and then compile the SQL

[5] Unless the precompile option *NOGEN is specified. See "Precompiler Options" on page 3-18 for more details.

preprocessor output in source file QSQLTEMP in library QTEMP with the CRTCPGM command. The member name remains the same. Once your program runs successfully compile it with *NODEBUG option.

4. Naming Convention

   The default naming convention for the processing of SQL statements is *SYS - system naming convention. If your application uses the SQL naming convention then the OPTION parameter must contain *SQL.

   If you use an SQL INCLUDE to get some SQL source statements from a member, then by default these should be placed in the same source file as the program being compiled. However, often this is not the case, as one of the main features of the INCLUDE is to copy standard SQL code held in one central location. To change the place where the precompiler looks for the INCLUDE statements, specify the source file name on the INCFILE parameter. As this can be a qualified name, there is a facility for providing the name of the library.

5. Use of Double Quote or Apostrophe in COBOL and C

   There is one point of confusion that may appear for COBOL programmers. In Interactive SQL and SQL embedded in RPG, FORTRAN or PL/I programs, the default literal character delimiter is the apostrophe. For instance:

   ```
   SELECT * FROM table-name
   WHERE SURNAME = 'SMITH'
   ```

   In COBOL, normal program literals may be coded with either the apostrophe or double quote.[6] The embedded SQL statements can only use one or the other. The default is the double quote - which as you have seen, is different from that of Interactive SQL. A programmer may use interactive SQL to test the SQL statements to be included in the COBOL application, and get confused with the different literal delimiter between this and the COBOL program.

   To change this to an apostrophe, the precompiler parameter OPTION (*APOSTSQL *APOST) must be specified. *APOSTSQL specifies the delimiter for SQL literals as entered by the programmer. *APOST specifies the delimiter for COBOL literals. To avoid confusion, it is better that the same literal delimiter be used for SQL literals and COBOL literals.

   It is important that in a COBOL environment, a standard be decided upon. It is recommended that the default (double quotes) be used.

   The cleaner solution is to adopt the default of double quotes, and be aware that this is different when testing an SQL statement in Interactive SQL.

   In C/400, the double quote **must** be used. Therefore, the programmer must change any saved Interactive SQL sessions so that the single quote is replace by a double quote.

The precompiler is discussed in more detail in "Creating the Access Plan" on page 5-2.

[6] By default, if the apostrophe is coded a warning message is given, but the specification is accepted. The double quote is the COBOL standard.

Binding an Application

The binding of a program and any referenced tables and views contained within an access plan can be modified automatically at program execution time.

The creation or deletion of an index is an example of a change which will affect the program. Adding or deleting a column to a table is another example.

A column may be added or deleted without the binding process having to create a new access plan.[7] When an SQL statement is bound (a SELECT statement for example) the columns specified on the SELECT statement list are locked by name. If a SELECT * is specified, the columns that are in the table at bind time are locked also by name.[8]

An application program will only fail if a table, view, or referenced column in the table does not exist at runtime.

Your program will not have to be recompiled after any of the following:

- Renaming a column, if not referenced
- Changing column attributes
  - Provided the data type is not changed (alpha to numeric)
  - Decimal places may be changed
  - Column length may be changed, but an SQL warning code may be given and an indicator variable may be set

The fact that the column names are locked at bind time may cause an anomaly to occur. If a program that contains a SELECT * is compiled, the access plan is created referencing all the columns in the table. If the table is then changed (say adding a column) the program can still be run, as columns are retrieved by name. The new column is not noticed by the program at this stage. Now consider if the program is recompiled without being changed. A new access plan is created, which contains all of the columns. The program will not compile, as there are not enough host variables specified on the SELECT INTO statement to store each column value.

For more information on binding, including late binding, see "Creating the Access Plan" on page 5-2.

Error Processing

The use of the SQL communication area (SQLCA) is similar to a MONMSG in a CL program, or checking the return code or indicator from a program I/O statement. Good programming practice includes a check of such return codes after the execution of every I/O statement. For reasons of brevity, the examples in this document have not included checks of the SQLCA after each I/O statement.

---

7 For a column to be added or deleted, the table would have had to be renamed, and a new table created with the extra/deleted field, then the data copied back in.

8 SELECT * should be avoided wherever possible. It has some performance implications when the number of columns in the table is high. It is better to specify only the columns that are required.

In this section, you cover some of the key areas of the SQLCA, and cover coding methods for the inclusion of these techniques in a program. This is intended as a guideline of the different ways of handling both expected and unexpected return codes.

There are a variety of ways to handle the processing of errors from SQL operations. One major facility that is available for error processing is the WHENEVER statement.

WHENEVER

The WHENEVER statement specifies the host language label to which execution will be transferred when an exception or warning condition occurs. There are three types of the WHENEVER statement:

```
WHENEVER NOT FOUND
WHENEVER SQLWARNING
WHENEVER SQLERROR.
```

These can be considered as three levels of severity of return code.

The NOT FOUND type is used to identify an SQL return code of +100. This is likely to occur when executing a FETCH statement and the end of the SELECT group is reached, or when selecting, and no records are found.

The SQLWARNING type identifies either a zero return code with a warning condition, or a positive return code that is not +100. This could occur when a field value is truncated, for example.

The SQLERROR type identifies a negative SQL return code. This is likely to be a severe error situation. In static SQL, many errors often that have a negative return code will not allow the program to recompile. That is, the error is related to a major change in the structure of the database which the program is using, thus causing the error. In dynamic SQL, this may occur due to an error in the syntax of the SQL statement that is being dynamically executed.

To correctly understand how the WHENEVER works, it is worth explaining what happens when the precompiler generates the HLL calls to QSQROUTE. Included in "Code Example For Use of SQL WHENEVER in RPG" on page A-1 is a source listing of an RPG program and in "Code Example For Use of SQL WHENEVER in COBOL" on page B-1 there is an equivalent COBOL program. These include some SQL WHENEVER statements.

RPG Program Example: Examine the first group of C-specification statements in the following example, from the first SQL WHENEVER to the SQL COMMIT (just before the second SQL WHENEVER), all on the first page of the output.

The precompiler will take this code and comment out the programmer's SQL statements, and replace them with calls. Unlike most other SQL statements, the WHENEVER is not actually replaced by a call. The precompiler will include an HLL statement which will check for errors of the specified severity[9] on every SQL statement after this point in the program, until the end of the program or another WHENEVER is encountered for this severity.

[9] The three severities being NOT FOUND, SQLERROR, and SQLWARNING.

For instance, in the code described, the compiled output looks like this:

```
2500  C*
2600  C                         Z-ADD.06       PERCNT
2700  C                         MOVEL'MA%'     PROJID
2800  C*
2900  C* Update the selected projects by the new percentage. If
3000  C* errors occur during the update, ROLLBACK the changes.
3100  C*
3200  C*EXEC SQL WHENEVER SQLERROR GOTO UPDERR
3300  C*END-EXEC
3400  C*
3500  C*EXEC SQL
3600  C* UPDATE USER1/TEMPRACT
3700  C*    SET EMPTIME = EMPTIME * (1+:PERCNT)
3800  C*    WHERE PROJNO LIKE :PROJID
3900  C*END-EXEC
3500  C                         Z-ADD00002     SQLER6
3500  C                         CALL 'QSQROUTE'
3500  C                         PARM           SQLCA
3500  C                         PARM           PERCNT
3500  C                         PARM           PROJID
3500  C               SQLCOD    CABLT0         UPDERR  1
4000  C*
4100  C* Commit changes.
4200  C*
4300  C*EXEC SQL COMMIT
4400  C*END-EXEC
4300  C                         Z-ADD00003     SQLER6
4300  C                         CALL 'QSQROUTE'
4300  C                         PARM           SQLCA
4300  C               SQLCOD    CABLT0         UPDERR  2
4500  C*
```

Notice the last statements on lines 3500 **1** and 4300 **2**. The function of the WHENEVER is simply to place a compare-and-branch after every SQL statement until the next WHENEVER of the same severity or the end of the program.

This highlights an important aspect of the usage of the WHENEVER statement. The replacement is done on a simple code sequence basis, and is not based upon program logic flow.

Assume that, between the UPDATE and COMMIT statements in the code example above, there was a call to a subroutine. The error processing for SQL calls in that subroutine would depend upon the location of the subroutine code in the program and any WHENEVER statements that are in effect based on its location. In this example, an error in an SQL statement in the subroutine may not necessarily cause the program logic to continue at the label UPDERR.

The inclusion of more than one WHENEVER at one part of the program can be a normal practice, provided they are of different levels of severity. Each WHENEVER will place a different test after the SQL calls that follow it. For example, later on in the program listing (on the second page), a FETCH

statement is performed to retrieve into host variable RPT1. Two WHENEVER statements are in effect, one for SQLERROR **3** and one for NOT FOUND **4**.

```
7000  C*EXEC SQL
7100  C*   FETCH C1 INTO :RPT1
7200  C*END-EXEC
7000  C                    Z-ADD00008     SQLER6
7000  C                    CALL 'QSQROUTE'
7000  C                    PARM           SQLCA
7000  C                    PARM           EMPNO
7000  C                    PARM           PROJNO
7000  C                    PARM           ACTNO
7000  C                    PARM           STARDT
7000  C                    PARM           ENDDT
7000  C                    PARM           EMPTIM
7000  C          SQLCOD    CABLT0         RPTERR  3
7000  C          SQLCOD    CABEQ100       DONE1   4
7300  C                    EXCPTRECB
7400  C                    END
7500  C          DONE1     TAG
7600  C*EXEC SQL
7700  C*  CLOSE C1
7800  C*END-EXEC
7600  C                    Z-ADD00009     SQLER6
7600  C                    CALL 'QSQROUTE'
7600  C                    PARM           SQLCA
7600  C          SQLCOD    CABLT0         RPTERR  3
7600  C          SQLCOD    CABEQ100       DONE1   4
7900  C*
```

As you can see, the result is relatively straightforward. If a NOT FOUND SQL code is returned, the logic is transferred to the DONE1 tag. If an ERROR code is returned, the logic is transferred to the RPTERR tag; otherwise the logic continues on following the CAB statements.

To cancel the WHENEVER processing for a severity, the CONTINUE clause is used. For instance,

```
WHENEVER NOT FOUND CONTINUE
```

will just stop the precompiler inserting the CABEQ100 statement after every SQL call from that point on, until another WHENEVER is coded for "NOT FOUND".

COBOL Program Example: Like the RPG example above, similar processing applies in COBOL. In the COBOL program listing in Appendix B, examine the A000-MAIN paragraph, from the first SQL WHENEVER to the SQL COMMIT, just before the second SQL WHENEVER statement.

The precompiler takes this code and comments out the programmer's statements, replacing them with CALL statements. As we know for RPG, the WHENEVER statement is not replaced by a CALL. Instead, a COBOL statement will be placed after every SQL statement following this to check for errors of the specified severity (NOT FOUND, SQLERROR and SQLWARNING). This will be

done until another WHENEVER statement of the same severity is encountered, or the program ends.

The compiled output for the code described is:

```
      011800*****EXEC SQL
      011900*****     WHENEVER SQLERROR GO TO E010-UPDATE-ERROR
      012000*****END-EXEC.
      012100*****EXEC SQL
      012200*****     UPDATE USER1/TEMPRACT
                        SET EMPTIME = EMPTIME * (1+:PERCENTAGE)
      012400*****       WHERE PROJNO LIKE :PROJID
      012500*****END-EXEC.
  111 012500     COMPUTE SQLERRD(6) = 00002
  112 012500     CALL "QSQROUTE" USING SQLCA  PERCENTAGE  PROJID
  113 012500     IF SQLCODE < 0 GO TO E010-UPDATE-ERROR ELSE 5
  115 012500     MOVE SQLCAID TO SQLCAID.
```

As you can see from statement 113 **5** the function of the WHENEVER is to place an IF - THEN statement after each SQL statement until the next WHENEVER of the same severity is found, or the program ends.[10]

This highlights an important aspect of the WHENEVER statement. It is not based on program logic, simply on a replacement of code basis. This means that if a paragraph was performed from inside the piece of code above, and also performed from various other parts of the program, and an SQL error occurred during execution of that paragraph, the action taken by SQL would depend upon the most recent WHENEVER statement found of the same severity, prior to the location of the paragraph in the program.

The inclusion of more than one WHENEVER statement in a part of a program can be normal practice provided that they are of different severity levels. Each WHENEVER places a different IF statement after SQL statements. For example, when execution of B000-GENERATE-REPORT1 begins, one WHENEVER statement is already in effect (from immediately after the COMMIT statement in A000-MAIN). A second WHENEVER condition is brought into effect at the start of B000-GENERATE-REPORT1. The compiled code for B000-GENERATE-REPORT1 is as follows:

[10] The "ELSE MOVE SQLCAID TO SQLCAID" statement is generated purely to handle the nesting of IF statements that may be in effect when the SQL statement is executed.

```
     011800 B000-GENERATE-REPORT1.
     022900*****EXEC SQL
     023000*****     WHENEVER NOT FOUND GO TO A100-DONE1
     023100*****END-EXEC.
     023200*****EXEC SQL
     023300*****     FETCH C1 INTO :TEMPRACT
     023400*****END-EXEC.
153  023400      COMPUTE SQLERRD(6) = 00012
154  023400      CALL "QSQROUTE" USING SQLCA  EMPNO OF TEMPRACT
                                                    OF RPT1 PROJ
                 NO OF TEMPRACT OF RPT1 ACTNO OF TEMPRACT OF
                                                  RPT1 STARTDATE
                 OF TEMPRACT OF RPT1  ENDDATE OF TEMPRACT OF
                                                 RPT1 EMPTIME OF
                 TEMPRACT OF RPT1
155  023400      IF SQLCODE < 0 GO TO E020-REPORT-ERROR ELSE 6
157  023400      IF SQLCODE = 100 GO TO A100-DONE1 ELSE 7
159  023400      MOVE SQLCAID TO SQLCAID.
```

Nested IF statements are generated to handle the two WHENEVER conditions. If an error occurs, E020-REPORT-ERROR is performed **6**, and if a NOT FOUND condition occurs, A100-DONE1 is performed **7**.

To cancel the WHENEVER action for a certain severity, a CONTINUE clause is used. For instance

```
WHENEVER NOT FOUND CONTINUE
```

will stop the precompiler inserting the statement "IF SQLCODE = 100 ..." from that point on.

PL/I Programs with SQL WHENEVER: PL/I programs follow a logic quite similar to the examples described above. Statements are added after each SQL call to check the return code at different levels of severity. See "Code Example For Use of SQL WHENEVER in PL/I" on page C-1 for an example of SQL WHENEVER in a PL/I program.

WHENEVER Examples: Despite the GO TO logic of the WHENEVER statement, it can still be used in a structured manner. The following examples show pseudo-code for inclusion of the WHENEVER statement in an application for the SQLERROR and NOT FOUND conditions.

- WHENEVER SQLERROR

  This is very useful in a commitment control environment where a number of statements are required to be processed together. If one statement fails, then the logic can be transferred to an error routine which can display a message to the user, before rolling back to the transaction boundary.

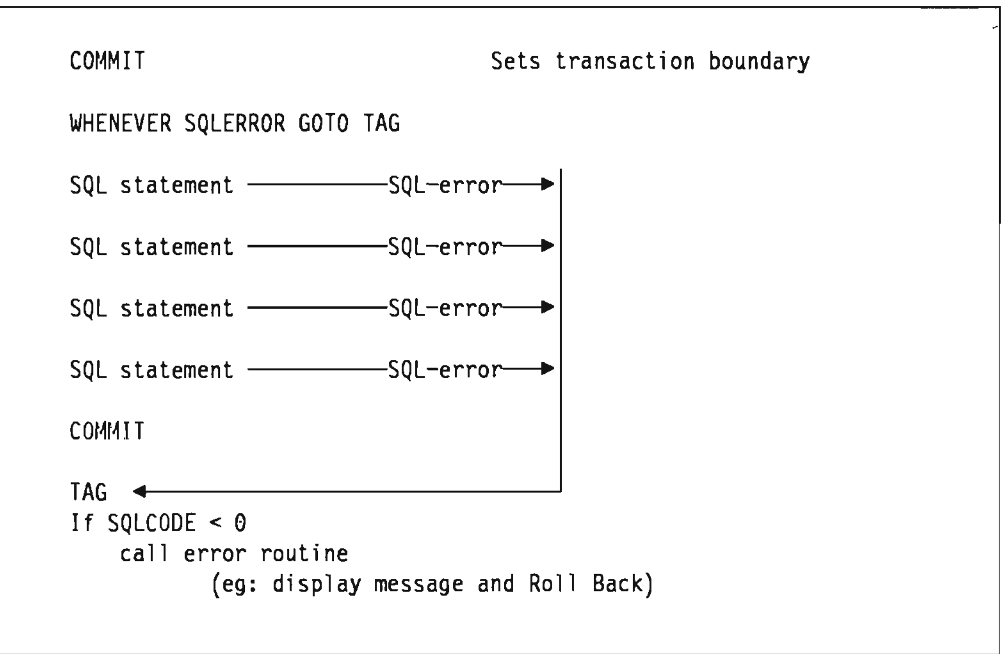


Figure 3-1. WHENEVER SQLERROR

- WHENEVER NOT FOUND

  This can be used when fetching from a selected set, providing a structured way to exit from the processing loop. This is almost exactly the same as reading from a file. If no records are found when the file is opened, then this can be trapped before proceeding to retrieve data from the table.

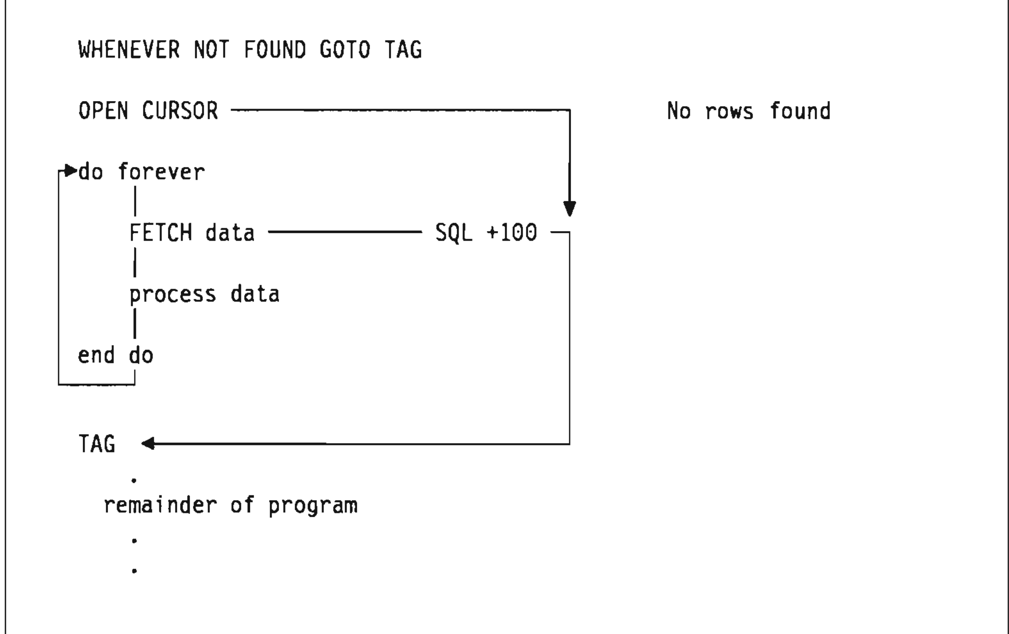


Figure 3-2. WHENEVER NOT FOUND

Using the SQLCA

Despite an attempt at a structured implementation, the WHENEVER statement still basically performs a GO TO. This is not always appropriate, and in some instances a specialized subroutine may be required for handling error conditions. If you wish to do this in an application, then you will need to test the SQL codes explicitly, and then execute the required subroutine.

The nature of this test can be relatively simple (one or two lines of code), or could be more complex, depending on the requirement for the error condition. There are a variety of fields that are available to the programmer in the SQLCA. These can be tested after the SQL statement is executed, or displayed in an error report for the program, or both.

The reporting of SQL exceptions lends itself to the additional fields in the SQLCA. These are usually not displayed in an application when *expected* error conditions occur. In coding an error reporting subroutine for *unexpected* errors in a program, the following SQLCA fields are worth noting. The first-mentioned field is the name for COBOL, C, FORTRAN and PL/I applications, the second for RPG:

SQLCA
Field ***Meaning***

SQLCODE / SQLCOD
: SQL return code. This is the main field to check for an error condition.

SQLERRML / SQLERL
: Length of the SQL error message field data.

SQLERRMC / SQLERM
: SQL error message field data. This is useful to display for an unexpected error condition. You could use the CL command SNDPGMMSG to do this. The message number would be SQLxxxx, where xxxx is the absolute value of the SQL code. The message file is QSQLMSG and the MSGDTA is in SQLERRMC.

SQLERRD / SQLERR (SQLER1-6)
: An array in COBOL and PL/I, six integer fields in RPG which can be checked under a variety of conditions. SQLERRD(3) can be used to obtain the number of rows affected after an INSERT, UPDATE or DELETE. This can be used in an error situation or as a normal condition, to return a message to the user for a set-based operation.

SQLWARN0 / SQLWN0
: Contains a "W" if one of the following conditions has a warning.

SQLWARN1 / SQLWN1
: Contains a "W" if a column (character or numeric) was truncated when assigned to a host variable.

SQLWARN3 / SQLWN3
: Contains a "W" if the number of columns and the number of host variables is not the same for a fetch operation or SELECT INTO.

SQLWARN4 / SQLWN4
: Contains a "W" if a prepared UPDATE or DELETE statement does not contain a WHERE clause. This happens on the PREPARE statement so that, before executing the statement, the program is aware that the operation will affect *all* rows in the set (table or view).

Note: Due to the nature of RPG, the fields in the SQLCA have been renamed to a name length of six characters. As the other languages do not have this limitation, the full name is used.

Inclusion of the SQLCA in the program: Before any processing can occur, the SQLCA must be included in the program. This can be done with the SQL INCLUDE statement. This *must be done* by the programmer for COBOL, C/400, FORTRAN and PL/I. These languages require an INCLUDE SQLCA or a HLL variable with the name of SQLCODE (SQLCOD for FORTRAN). If the variable SQLCODE is coded, the precompiler automatically adds the SQLCA. In RPG, the SQLCA is included automatically. The user can code an INCLUDE SQLCA, but it is accepted and ignored.

Coding a Routine for Unexpected Errors

Coding such a routine is relatively straightforward. We have discussed the significant fields of the SQLCA that should be displayed or printed by the routine. The form of output will depend on your installation standards (that is whether to display a message to a user, or produce an entry in a special log file, or both). There is a significant point regarding the handling of an error situation when running under commitment control.

Take, for example, a transaction step which consists of one or more SQL statements, followed by an SQL COMMIT statement. If an error occurs in the processing, maybe not all of the statements will be executed. The program uses the WHENEVER SQLERROR to go to an error routine. It is important that the error routine reports the SQL error previously encountered *before* performing a rollback. If the rollback is performed first, the SQLCA fields will be lost for the error condition, and will contain values pertinent to the ROLLBACK statement. The user must either report the previous error first, or save the contents of the SQLCA, and later report the error.

Second Level Message Text: If an SQL message is encountered in an application or interactively, and second level text is required, use the following CL command:

```
DSPMSGD RANGE(msg-id) MSGF(QSYS/QSQLMSG)
```

to display the message, or to retrieve into a CL program variable use the following commands

```
RTVMSG MSGID(msg-id) MSGF(QSYS/QSQLMSG) SECLVL(CL-variable) ...
SNDPGMMSG ... MSGDTA(CL-variable)
```

The message number will be something like SQL0109. This is equivalent to the SQL return code of -109.

Indicator Variables

Indicator variables are an extension to the SQLCA error processing facilities when writing programs with embedded SQL. They allow the programmer to check the validity of each host variable as it is given a value from a table in a fetch operation. The use of indicator variables is optional.

Indicator variables in SQL/400 are mainly for compatibility with other SQL products at this stage, as their primary purpose is to allow the programmer to check for null values being returned to a host variable. At Release 3 Modification Level 0, AS/400 SQL does not support null values, hence this test is not likely to be often used. The secondary purpose of the indicator variable is to

check for field truncation, numeric conversion or arithmetic expression errors in a fetch. These may not often occur in simple applications, but may more commonly occur when using dynamic SQL when results are unexpected.

Field Truncation: If a column (character or numeric) is retrieved into a host variable that is smaller in length than the column as defined in the table, then it is likely that some of the data may be lost in the fetch operation. An indicator variable associated with the host variable will return the original length of the column. Note that this will return the length of the column, and not the length of the data in the column. It is up to the programmer to check the indicator variable. In addition, the SQLCA warning field-1 will contain a "W" after truncation has occurred.

Numeric Conversion or Arithmetic Expression Errors: A numeric conversion error will occur for instance when a numeric column is being retrieved into a character field in a program. When this happens, the associated indicator variable will contain -2.

Program Design Guidelines

In many ways, programming with an SQL table can be quite different from working with a normal database file. Consider the three traditional file structure types: sequential, indexed and relative.

Most applications use a form of indexed file organization, where there is one (or more) key fields in a file. To get a record from a file, the programmer specifies a key which uniquely identifies that record. For a relative file, the programmer specifies a record number, which can be generated with a hashing algorithm based on a pseudo key field. Sequential organization limits the programmer to reading from the beginning to end of the file.

Working with an SQL table can be quite different from this. There is only one file organization. In SQL, the programmer does not have all of the retrieval functions that are available with the traditional file structures. Conversely, SQL offers many extra functions that these other file organizations do not provide. This is discussed later.

In SQL, the program can read sequentially through the entire table. This can be achieved by the use of a DECLARE CURSOR statement followed by a FETCH statement enclosed in a program loop. However, the program cannot then perform a read prior (or read previous) operation. The FETCH statement works only in one direction.[11]

Similarly, using SQL, a programmer cannot simply get record number 22345, as a relative file organization allows. There is no equivalent function for processing an SQL table.

The analyst and programmer must consider these points before designing an application. Not all functions available with the older file organizations are available with SQL statements on an SQL table. This is not to say that SQL does not have a major place in most application requirements, just that the

[11] See "SQL Implementation Techniques" on page 3-32 for more information regarding HLL read prior access of an SQL table.

application may look very different when designed around a relational database structure.

The following table summarizes the normal types of functions required by a programmer to maneuver around a file, and how such maneuvers can be represented in SQL. This can be used as an aid to conversion - but for the best use of SQL (from a performance and readability viewpoint), the entire program may need to be rewritten and restructured, rather than simply replacing one HLL I/O statement with one SQL statement. SQL is optimized to operate with rows a-set-at-a-time, and one-to-one code replacement largely defeats the advantages that can be gained by a non-procedural language such as SQL.

Common HLL Programming I/O Techniques	Equivalent Implementation in SQL
Relative record read	Not available directly. The application should be changed so that data is retrieved based on content, rather than physical location.
Position the cursor, followed by sequential reads	Use a DECLARE CURSOR with SELECT statement and a WHERE clause that contains key > = required value. The first FETCH statement positions the cursor and returns the first record that satisfies the WHERE clause. Subsequent fetches will return consecutive rows.
Read prior record	Not available in SQL as such, but depending on whether all processing is going to be in reverse sequence, and there is only one key field, the SELECT statement can specify an ORDER BY key DESCEND clause to get the same effect.
Random processing based on key field	Individual selections or updates specifying the required key field on the WHERE clause.

Table 3-1. Summary of I/O Processing Equivalents Between HLL Programs and SQL Programs

The remainder of this section discusses points related to the design of applications using SQL and considerations for implementation.

Wordiness

Deciding when to use SQL and high-level language (HLL) I/O statements is and will continue to be a debated point in application design. One area which is very noticeable is the wordy nature of some embedded SQL statements. However, this applies in both directions. In some instances, the HLL code may be less concise.

The following example retrieves and updates rows from a table in a loop, and writes the updated record to a printer file. This is accomplished with 24 lines in the embedded SQL example, whereas in RPG example, the same function can be done in nine.

SQL

```
FRPT  O   E      PRINTER
C/EXEC SQL DECLARE CURSOR C1
C+ FOR  SELECT NBR, NAM, SAL
C+ FROM EMP WHERE NBR = :NBR
C+ FOR UPDATE OF SAL
C/END-EXEC
C*
C/EXEC SQL OPEN C1
C/END-EXEC
C*
C           EXSR  SFETCH
C    SQLCOD DOWNE 100
C    SQLCOD IFEQ 0
C/EXEC SQL UPDATE EMP
C+ SET SAL = SAL + :RAISE
C+ WHERE CURRENT OF C1
C/END-EXEC
C           WRITERPT
C           END
C           EXSR  SFETCH
C           END
C*
C* Subroutine to do FETCH
C*
C    SFETCH BEGSR
C/EXEC SQL FETCH C1 INTO
C+ :NBR, :NAM, :SAL
C/END-EXEC
C    SFETCH ENDSR
```

vs.

RPG/400

```
FEMP UP  E   K  DISK
FRPT O   E      PRINTER
C   NBR   CHAINEMP    99
C   *IN99 DOWEQ '0'
C         ADD  RAISE SAL
C         UPDATEMP
C         WRITERPT
C         READ EMP    99
C         END
```

On the other hand, for a more complex SELECT statement example using LIKE:

SQL

```
C/EXEC SQL  DECLARE CURSOR C1
C+ FOR  SELECT NBR, NAM, SAL
C+ FROM EMP WHERE NAM LIKE '%:SEARCH%'
C+ AND DPT BETWEEN (:STRDPT :ENDDPT)
C/END-EXEC
C*
C/EXEC SQL   OPEN C1
C/END-EXEC
C*
C    SQLCOD DOUNE*ZERO
C/EXEC SQL FETCH C1 INTO :NBR,
C+   :NAM, :SAL
C/END-EXEC
C    SQLCOD IFEQ  *ZERO
C           WRITE SFLREC
C           END
C           END
```

-ANY HLL-
MUCH MORE
COMPLEX

But this is not always the case. The programmer's alternative in HLL is to use OPNQRYF to provide the LIKE functions as there is no equivalent in DDS-created logical files.

SQL is very well suited to "set at a time" operations that return or process many records - for example mass updates, mass deletes, filling subfiles and report programs with user selection; or it could be used with complex selection criteria.

In SQL, there is no need for the programmer to navigate the database as with HLL I/O statements, but it does change the design methodology for database structure and maintenance.

The nature of such a complex selection as in the example above is useful for contrasting the performance implications of SQL versus HLL I/O. The database manager which operates in the licensed internal code of the system performs the selection as requested by SQL. Operations which take place at this level of the operating system execute very fast. Hence, in comparing the LIKE function of the SELECT statement in the example, the SQL implementation is much faster than the code in an HLL program to do the same.

Performance is an issue discussed in detail in another topic (see "SQL Performance" on page 5-1). There are some application design points that relate to performance that are relevant for the discussion here. If an application is performance sensitive, then it is better that HLL I/O statements be used for:

- Transaction processing
- File maintenance
- Simple inquiry.

For performance, SQL is better suited for "set at a time" operations like:

- Prompted reports
- Complex inquiry
- Dynamic SQL for front-end query
- Mass insert, update, delete.
- Statistical selects (Count, Minimum, Maximum, Average, Sum)
- Alpha-search selects using Like.

SQL Implementation Techniques

SQL provides you with an option for performing file processing, and also offers you an additional method of creating database files. As a result, you are now faced with the decision of whether to move to an SQL implementation or not. If SQL file/table accesses are to be used in applications, the AS/400 system development environment must be used. However, the question of whether to implement SQL collections and tables, or to continue to use externally described files is less clear-cut, since SQL can even access existing DDS- or IDDU-created files.

This section covers the implications of using SQL tables instead of, or as well as, externally described files, and also covers considerations regarding SQL versus high-level language (HLL) programming of file/table accesses. It is aimed particularly at those who already know the System/36 or System/38 environment well, and also those considering implementing SQL.

Defining Tables and Files

There are several ways of creating tables in an SQL collection. It is always possible to use the CREATE TABLE statement. In addition, for existing files, you can use the Copy File (CPYF), Move Object (MOVOBJ), Create Duplicate Object (CRTDUPOBJ), Restore Object (RSTOBJ) and Create Physical File (CRTPF) commands.

New Table Creation

When creating a new table, for which there is no data already on the system, the SQL CREATE TABLE statement or any of the above CL commands may be used. To put data into the table, the SQL INSERT INTO statement or CL CPYF command may be used.

Moving Existing Files/Tables into an SQL Collection

The CREATE TABLE statement can be used followed by the INSERT command to move data from an existing file into a new table in a collection. However, the CREATE TABLE statement requires you to redefine the columns in the table, although they have already been defined in the existing file definition. On the other hand, the CPYF, MOVOBJ, CRTDUPOBJ and RSTOBJ commands can be used to create/move a file or table into a collection, using the existing definition to produce named columns matching those from the original file (DDS or IDDU, or SQL table). They also copy/move the data in at the same time.[12]

IDDU File Considerations

On the AS/400 system, IDDU can be used in two modes. The first mode is as it works on a System/36. The definition is not only external to the program, but also external to the data.

In the System/36 Environment, the original file has only one large field. The external definition creates a logical template over this file so that it is like many smaller fields. The process for creating such a definition may be like the following:

- Create Physical File (CRTPF) with a record-length entry, that is, *without* DDS
- Create IDDU definitions for the fields in the file
- Link the file to the IDDU definition (LNKDTADFN)
- This creates a program described (not externally described) file which is IDDU-linked.

The second mode is to provide a full external definition as available on the AS/400 system. This is where the definition is stored with the data itself. The steps to create the externally defined file with IDDU are:

- Create IDDU definitions (field, format, file)
- Create the file with IDDU

  This creates an externally described file (instead of DDS-input, IDDU-input) and links the file to the IDDU definitions. This may be verified with the Display File Description command (DSPFD).

[12] The CPYF command can create a new table (physical file) in the SQL collection, leaving the existing file as is if you use CRTFILE(*YES). The MOVOBJ command actually places the file/table into the SQL collection, with no copy left in the original library/collection.

Database Creation Summary

The following summary shows the various options for putting data into a collection, without explicitly using the CREATE TABLE command.

Type	Command	All SQL Entries Created[13]	Object Created	Notes
DDS-def. PF	CPYF CRTFILE(*YES)	Y	Table (PF)	
	MOVOBJ, CRTDUPOBJ, RSTOBJ	Y	Table (PF)	
DDS-def. LF	CPYF CRTFILE(*YES)	Y	Table (PF)	
DDS-def. LF	MOVOBJ, CRTDUPOBJ, RSTOBJ	N	None	All logical files in an SQL collection must be SQL-created. Thus these commands are not allowed to put a DDS-defined logical file into an SQL collection.
SQL table	CPYF CRTFILE(*YES)	Y	Table (PF)	
SQL table	MOVOBJ, CRTDUPOBJ, RSTOBJ	Y	Table (PF)	
SQL view	CPYF CRTFILE(*YES)	Y	Table (PF)	
SQL view or index	MOVOBJ, CRTDUPOBJ, RSTOBJ	Y	View or Index (LF)	Since this view is a logical file, created by SQL, it can be moved into a new collection, and remains a logical file.
IDDU file	CPYF CRTFILE(*YES)	Y	Table (PF)	
IDDU file	MOVOBJ, CRTDUPOBJ, RSTOBJ	Y	Table (PF)	Only externally described files; not program described and linked files.

Table 3-2. Copying and Moving Files into an SQL Collection

Copying and Moving Files/Tables: Journaling Considerations

When a collection is created using the SQL CREATE COLLECTION command, a journal is automatically created (QSQJRN) in the same collection. All activity on tables created in that collection is recorded by journal entries made to that journal, and SQL uses these entries to implement its commitment control environment. When tables are created in a collection through the use of the CRTPF, CPYF, MOVOBJ, CRTDUPOBJ or RSTOBJ commands, the journaling is not implemented in the same way as on a CREATE TABLE statement.

The following table summarizes the journal status of a file/table once the CPYF or MOVOBJ command has been used, as well as how to start journaling if it is not active.

[13] Entries made into the collection *DTADCT, SYSTABLES, SYSCOLUMNS, and so on.

Type	Command	Journaling Status
SQL table	CPYF CRTFILE(*YES), RSTOBJ, CRTDUPOBJ	New table is not journaled. STRJRNPF should be issued, naming the QSQJRN in the new collection. Commitment control will then be as normal.[14]
DDS-def. PF	CPYF CRTFILE(*YES), RSTOBJ, CRTDUPOBJ	Same as for SQL-table above.
SQL table	MOVOBJ	Table will continue to be journaled to the journal in the previous collection.[15] Use ENDJRNPF to end journaling to old journal, and STRJRNPF to QSQJRN in the collection.
DDS-def. PF	MOVOBJ	Same as for SQL table and MOVOBJ above.

Table 3-3. Copy and Moving Files/Tables: Journaling Status

Indexes

An index should be created on a table whenever access to records in a certain order is going to be requested frequently. SQL indexes have to be created in an SQL collection. However, the base table can be any AS/400 physical file. Without a user-created index, the database optimizer may create a temporary index to handle certain requests. This is particularly important for large files. Refer to "SQL Program Design Considerations" on page 3-1 for more details on when and how the database optimizer uses indexes.

SQL Indexes cannot be created on DDS-created physical files or IDDU files if the files are not stored in an SQL collection. If a file is frequently accessed by SQL statements in a certain order, it can be moved into a collection, and then an index may be created. Creating a DDS keyed logical file will not overcome this since SQL will disregard the key when selecting data unless ORDER BY with the same key is specified.

An index may be stored in a collection other than the one which contains the table upon which the index is based. Furthermore, the database optimizer will use this index when required, even if the library that contains the index is not in the library list.

Join Files

SQL tables can be "joined" by views, which are similar to DDS-defined join-logical files. "Join" views cannot be updated, and have read-only access; join-logical files have the same restriction. But note that DDS-defined join-logical files support partial outer join through the JDFTVAL keyword.

A DDS-defined join-logical file cannot be stored in an SQL collection since it is a logical file, and only SQL-defined logical files (views) can be stored in a collection. However, a join-logical file can still be accessed by SQL.

---

[14] Since the QSQJRN journal is not automatically attached to this new table, it is very easy to omit journaling unintentionally. In the case where a program is run with commitment control intended through SQL but no journal attached, the program will still execute, but a negative SQLCODE will be returned on each attempt to INSERT, UPDATE and DELETE rows in the unjournaled tables. If the SQLCODE is not checked, the error may not be immediately visible.

[15] Journaling to the previous collection's journal is not in itself a problem. However, journaling two tables under commitment control to different journals in the same program is not allowed. Since two tables may be in the same collection following a MOVOBJ but may unintentionally still be journaled to different journals, a negative SQLCODE will be returned.

Changing File/Table Structure

Sometimes it is necessary to change the original structure of a table or file, for example, by adding or deleting a column. In such cases, the following steps can be taken:[16]

- To change the structure of a DDS-defined file:

```
CPYF - to move the data into a temporary file
STRSEU - to change the original source DDS
ENDJRNPF - to end journaling of physical files
DLTF - including the logical files based on the file
CRTPF - to create the new file
CPYF - to move the data back into the file*
CRTLF - for logical files required
JRNPF - to recommence journaling
```

- To change the structure of an SQL table:

```
CPYF - to copy the data to a temporary file
DROP TABLE tablename (which drops dependent views and indexes)
CREATE TABLE tablename
      (field1  type(n)  NOT NULL WITH DEFAULT,
       field2  type(n)  NOT NULL...)
CPYF or SQL INSERT - to copy the data back into the table*
CREATE VIEWs and INDEXes
```

* The *MAP parameter in the CPYF command can be used when the order of the fields/columns has changed, to ensure that the values are correctly matched from the fields in the old file to the new one.

Note: In both cases, if a new column is added, blanks or zeros will be placed into it.

The difference between the two methods is the way the file/table is changed. While the DDS-created file structure can be changed by editing the source code, and then recreating the file, the SQL CREATE TABLE statement requires redefinition of all the columns in the table. For a large table with many columns this can be error-prone and time-consuming. Therefore, it would be useful to keep "table-creation" code in source files, which could be executed from within a program, and edited and re-executed easily if necessary. This would be very similar to coding a CL program to create device descriptions and user profiles, instead of entering the commands one by one interactively, except that since SQL statements cannot be executed from a CL program, they would have to be placed into an HLL program.

SQL Objects and Override Data Base File (OVRDBF)

You know that the data in an SQL table is actually stored in a physical file. The nature and structure of SQL provides for accessing only the first member of this physical file by default. When designing applications, this is something that, wherever possible, should be adhered to as a standard. If required, to change the member being processed, the OVRDBF command can be used. However this is not a recommended design practice, as SQL allows for only one member to be used at once, and it is a cleaner solution to have members in unique physical files (that is, one member per physical file). In the data definition for an

[16] Changing the structure of a table or file could also be achieved by creating a new table/file, copying the data into that file, deleting the old one, and then renaming the new one. However, since a new file will be created with the same DDS, one must be careful not to lose track of the name under which the DDS is stored.

application, the analyst must also keep in mind that SQL will not work with multiple format logical files.

The OVRDBF command can be successfully used to run an SQL application against another file with a similar data structure without the need for recompilation. This is particularly useful when testing an application against a variety of standard test data files.

OVRDBF can be used to override SQL data manipulation statements only.

Using Tables and Files in High-Level Language Programs

This section discusses several aspects of using tables and files in HLL programs.

Using SQL to Access DDS- and IDDU-Defined Files and SQL Tables

SQL can be used on IDDU- or DDS-defined files or on SQL tables. Most file accesses within an HLL program can therefore be performed using SQL input/output statements instead of the HLL input/output statements.

Traditionally, an RPG program requires an F-specification to identify each file which a program uses. Field definitions can be copied into the program automatically. In the C-specifications, files are opened, read, processed and closed.

In general, a COBOL program requires a "SELECT - ASSIGN" statement, and an FD statement, for each file to be accessed within a program. A COPY DDS statement can be used to retrieve externally defined field definitions and SQL table descriptions. In the PROCEDURE-DIVISION statements the file will be opened, read - generally into some working-storage variables - processed, and closed.

When using only SQL to retrieve and process records from any table or file, the file identification statements can be omitted. The SQL DECLARE CURSOR and FETCH statements are used to define the data and read it. See Figure 3-3 on page 3-38 for an RPG example of how to code a basic SQL file access in a program and Figure 3-4 on page 3-38 for a COBOL example.[17]

[17] Each SQL statement must be preceded by an "EXEC SQL", and RPG and COBOL SQL statements must also be followed with an "END-EXEC" statement.

For an RPG program

```
DECLARE cursorname cursor for
SELECT field1, field2, field3...
FROM library/(table name or DDS-file or IDDU-file)*

OPEN cursorname

FETCH cursorname into Host variables (1) (2) (3)...

CLOSE cursorname
```

Figure 3-3. SQL File/Table Access in an RPG Program

For a COBOL program

```
WORKING-STORAGE SECTION.
   INCLUDE SQLCA


PROCEDURE DIVISION.

   DECLARE cursorname cursor for
   SELECT field1, field2, field3...
   FROM library/(table name or DDS-file or IDDU-file)*

   OPEN cursorname

   FETCH cursorname into Working storage variables (1) (2) (3)...

   CLOSE cursorname
```

Figure 3-4. SQL File/Table Access in a COBOL Program

* To receive the records from the file in a certain order, the ORDER BY clause can be used here. If not, the database optimizer will create or use an index based on its requirement for best performance, and the order of the records will be unpredictable and can change if reoptimized.

File and Table Field Names in COBOL/400

Both DDS and SQL allow field names to contain the underline character. However, the COBOL/400 COPY DDS-ALL-FORMATS statement does not allow replacement of hyphens by underline characters through the REPLACING clause. Therefore, underline characters should be used with care in field names of tables and files, except when used in ALIAS fields.

Use of ALIAS in DDS Statements

"ALIAS" is a DDS parameter which permits a field name to be known by another name within a program (for example, PARTNUM alias "PART_NUMBER"). This function can be used with SQL, when accessing DDS-defined files. In COBOL, the "COPY DD" statement copies the aliases rather than the original field names into the program. This is not the same as the COPY DDS statement. These aliases will generally be used for host variables in working storage. SQL then

uses the aliases for host variables within the program. Alias names cannot be used in SQL statements to reference column names.

RPG does not support aliases, and renaming of fields must be done by the programmer within the programs. See Figure 3-5 for an example of use of the ALIAS parameter in a COBOL/400 program.

```
WORKING-STORAGE SECTION.

   01  GROUPNAME.
   COPY DD-ALL-FORMATS of filename.

   PROCEDURE DIVISION.

   DECLARE cursorname CURSOR FOR
   SELECT field1, field2, field3,...
     .
     .
     .
   FETCH cursorname INTO :field1-alias, :field2-alias, :field3-alias...
```

Figure 3-5. Use of ALIAS Parameter in a COBOL Program

SQL Field Names

SQL permits the use of the underline character in its table definitions (table name and column name - maximum of 10 characters for a column name). In this case, in SQL statements, the field names must be written with the underline character, rather than the hyphen ("-") as is used in COBOL.

```
SELECT FIELD_1, FIELD_2, FIELD_3...
```

RPG Host Variable Definition Considerations

In an application, a programmer usually gets the definition of database file variables from the external definition of that file. But with SQL in RPG there is often no need for an F-specification, as all I/O is done with SQL.

If RPG file processing statements are not required in the program, then the preferred approach is to use an externally described data structure. This provides the same automatic provision of column/field definitions to the program, but avoids the overhead of opening the file and maintaining an open data path during program execution if this is not required for other purposes. A sample externally described data structure based on a table would look like this:

```
     I          E DSTABLE
```

The precompile process will then retrieve the field definitions. However, in SQL the size of a column name may be up to 10 characters. In RPG you are restricted to six; therefore if required you can change the field name in the program by specifying an I-spec for every field redefinition. The RPG maximum

of six characters for the data structure name could also be related to a longer table name via the data structure specifications. This may look like this:

```
ITABLE      DSLONGTABLE
I             LONGFIELD                        SFIELD
```

In this case the data structure TABLE will contain the field names from the table or file LONGTABLE. The field SFIELD has been given the longer name LONGFIELD.

For flexibility, it is often better to design the SQL table with column names that are six characters or less in length. This will help overcome the problem of having to rename fields in every program if RPG is used.

The programmer can use the OVRDBF command if F-specifications are used in the RPG program to retrieve the definitions and the table name is longer than the RPG-allowed eight characters.

SQL Table/File Access versus HLL Table/File Access

Since it is now possible to use an SQL table or a database file within a program, and to access the data through an HLL program with either SQL statements or HLL statements, there are many coding possibilities. Figure 3-6 on page 3-41 summarizes the difference between coding using a table/file and HLL processing, and a table/file processed by SQL.

COBOL	RPG
Table/file with no SQL processing SELECT file FD file COPY DDS- WORKING STORAGE (COPY DDS- optional) OPEN file READ file (INTO) CLOSE file	 FFILENAME IF E DISK OPEN file READ file CLOSE file
Table/File with only SQL processing WORKING STORAGE INCLUDE SQLCA (COPY DDS- optional) DECLARE cursor SELECT* OPEN cursor FETCH* (into fields named in copy DDS statement - optional) CLOSE cursor	 (optional) I E DSFILENAME DECLARE cursor SELECT OPEN cursor FETCH (into fields copied via external DS) CLOSE cursor

```
* Optional: If access is random, then this could be omitted and a
single SQL statement executed; for example:
     UPDATE table
     SET field1 = 0
     WHERE field2 = 'SMITH'.
```

Figure 3-6. HLL and SQL Access of Files/Tables

SQL INCLUDE

The text of an SQL statement can be created outside a program, and then copied into the program at compile time, using the SQL "INCLUDE" statement, in the same way as a COPY DDS command is used to copy DDS statements. The SQL statement(s) are written into a separate file, for example, "FILEX", surrounded by EXEC SQL and END-EXEC delimiters, and copied into a program thus:

```
EXEC SQL                  FILE X:
INCLUDE FILEX               EXEC SQL
END-EXEC.                   FETCH cursorname into :field1, :field2
                            END-EXEC.
```

This function is useful when a certain complex command or set of commands are used in several programs, and can be copied in. If the commands are changed, only the programs copying in the text need to be recompiled, as

opposed to changing each program separately if the commands were "hard-coded" in.

HLL (Non-SQL) Access of SQL Tables

While the use of SQL statements within HLL programs to access SQL tables or other files is definitely advantageous and easy to code, there are some functions which are not available from SQL directly (such as the READP command in RPG, or the READ PRIOR statement in COBOL). In these cases, it is possible to access SQL tables like files, through HLL processing in the following ways:

- *If the order of the records is important*

  If record order is important, a logical file can be created in a separate library, using DDS to select some or all of the fields and also to specify an order for the file. The HLL program will be coded to access the table through the logical file in exactly the same way as it does a logical file based on a DDS-created physical file. To create the logical file based on the SQL table, the SQL table is named in the PFILE parameter of the DDS, in the same way as a physical file usually is.

- *If the order of the records is not important*

  If record order is unimportant, the HLL program can name the table in its file definition statements in exactly the same way as for a DDS-created file. For example, in RPG,

  ```
  FFILENAME UF E   DISK
  F OLDREC       KRENAME NEWREC
  ```

  can be used, and in COBOL,

  ```
  SELECT filename ASSIGN to DATABASE-tablename
  ```

  can be used, followed by a

  ```
  COPY DDS-ALL-FORMATS of tablename
  ```

  statement, to declare the file. After that, regular READ and WRITE statements can be used as for sequential files. Any indexes on the table are ignored, so that records will be provided in their physically-stored order, and the RECORD KEY IS EXTERNALLY-DESCRIBED-KEY clause is not allowed.

- *Read Prior*

  There is no command in SQL to allow for fetching the previous record in a file. However, it is possible to use the RPG READP or the COBOL READ PRIOR statements when reading a table sequentially with HLL I/O statements. (To use a read prior function with random processing, see the next section "Random Access" below.) In RPG, READP will read the previous record from a full procedural file. The C-specifications are coded as if the SQL table were just a sequential file. The F-specifications must include the renamed record format. To use this function in COBOL, the following statements should be used:

```
SELECT filename ASSIGN to DATABASE-tablename
ORGANIZATION is INDEXED
ACCESS is DYNAMIC
RECORD KEY is fieldname.
```

In the Procedure Division, file reads are performed sequentially, starting with a READ FIRST statement, and then a READ NEXT statement. A READ PRIOR statement can be used at any time.

The above ORGANIZATION, ACCESS and RECORD KEY clauses *must* be used, even though the statements imply that the table is an indexed file, and it is not. In fact, if an index exists, it is ignored. Any field name can be used in the RECORD-KEY parameter.

- *Random Access*

  It is not possible to use an SQL table for HLL random access directly. A keyed logical file created over the SQL table must be used in such cases. This could be created either as an SQL index or a system logical file. It is then possible to randomly access the table through an HLL, and also to use the READ PRIOR statement on this logical view of the table.

Workstation Files

The Screen Design Aid allows the user to select fields from a database file when defining a screen format. This is useful as the required fields with definition attributes (such as headings, keyboard shift) can be simply placed on the SDA work screen without the need for redefinition. The definitions from an SQL table can also be copied onto the SDA work screen in this manner.

Declarative SQL Statements

There are a certain number of embedded SQL statements which can be seen as declarative. Generally, declarative statements are those which are not actually executed as a part of the normal program logic, but are used when the program is compiled. This is similar in concept to the declaration of new variables.

The common declarative statements in SQL include: DECLARE CURSOR, DECLARE STATEMENT, INCLUDE, and WHENEVER. A noticeable difference between a declarative and a procedural statement in SQL is that declarative statements do not generate calls to QSQROUTE. Declarative statements are not executed.

When to Use SQL

There are many other aspects to SQL which may cause the user to question whether he should use SQL or not. The following summarizes some of the more important aspects, and indicates where further information on each can be found in this document.

SQL and SAA

If applications are to be run on other IBM systems, then using SQL and following the SAA guidelines may facilitate this sharing of applications. Similarly, applications from other systems which use SAA can be moved to the AS/400 system. See "SQL Standards" on page 10-1 for more details on this consideration.

Knowledge of SQL

If the application developers have no knowledge of SQL, but are very experienced in a high-level language (HLL), then initially SQL programming will be a slower process than programming with the HLL. However, if the user is not an experienced programmer, SQL may prove easier to learn. If knowledge of SQL and an HLL is equal, SQL programming and debugging should be faster.

Multiple AS/400's

When an application is being developed for use on several AS/400s, only one machine requires the SQL/400 licensed program (5728-ST1). A compiled program (object) incorporating SQL statements can be installed on another AS/400 without the SQL/400 licensed program, and will execute normally, since run-time support of SQL is included in the OS/400. This is similar to what is applicable to the HLL compilers.

DDM Files

SQL only supports access to files on the system on which it is run. For this reason, it may be necessary to use HLL access to remote files. See "Distributed Data Management (DDM) Considerations" on page 6-1 for more details.

SQL Commitment Control

SQL supports COMMIT and ROLLBACK statements in addition to those already found in the AS/400 system. When a high-level language program incorporating SQL is compiled (using CRTSQLCBL, CRTSQLRPG, CRTSQLC, CRTSQLFTN and CRTSQLPLI), SQL commitment control is automatically added to the created program. The default commitment control level is *CHG, meaning that all records updated in a program will be locked until commit or rollback occurs. If the job completes without executing a "COMMIT" SQL statement or CL command, the changes will be rolled back. The commitment control level may be changed to lock all records read (*ALL) or to make changes to a file immediately (*NONE).

If a job is interrupted (for example, due to a power failure) and is running under commitment control *ALL or *CHG, all changes made to the file since the last "COMMIT", or start of job if no commit has been issued, will be rolled back. For those users who are not familiar with AS/400 commitment control, this SQL implementation offers a simple method of implementing an important recovery function. Full details of the implications of this are found in "Rollback Considerations" on page 7-10.

Error Handling

All SQL statements return values to the SQLCA (SQL communication area), which indicate the success or failure of each command, with error/warning codes and messages. Therefore, in the same way as it is possible to program in a high-level language to check whether a file access was successful or not (for example, the use of indicators or checking in INFDS after I/O statements in RPG, or SELECT... FILE STATUS is data-name in COBOL), an SQL code can be checked for information regarding execution of a statement.

The SQL WHENEVER statement can be used to handle errors, not-found conditions, and warnings. If an SQL "WHENEVER SQLERROR" statement is placed at the start of a program, and an SQLCODE of less than zero is received, the second half of the statement (CONTINUE/GO TO) will be executed. This is similar to the MONMSG statement in CL, and is again a very simple way of

handling unexpected conditions. For further details see "WHENEVER" on page 3-21.

Performance of SQL versus High-Level Language Accesses

Generally speaking, SQL will perform more efficiently than HLL equivalent coding for multiple record updates and deletes or complex selection. However, performance will be slower if single record data manipulation statements are compared with their HLL equivalents. You should only consider replacing *all* HLL data manipulation statements with SQL code if SAA compatibility is required.

Performance is a very important consideration. The use or misuse of SQL can affect the way in which an application performs. It is important that this issue is investigated thoroughly before embarking on a major development or conversion project using SQL. Read "SQL Performance" on page 5-1 for more details.

Which DDL?

When moving from a System/38 where all files are already defined with DDS, the functions to be gained by moving the files into an SQL collection are those of the SQL catalogs, which provide an easy method of summarizing table layouts, fields and indexes.

SQL provides a very easy method of creating files, which is perhaps preferable for those not familiar with DDS. IDDU is also easy-to-use, but does not provide the facility to create logical files. It can also be used to create tables in an SQL collection. For some applications, when SQL tables cannot be accessed as desired, a DDS logical file can be created over an SQL table, to provide the necessary function.

SQL I/O statements are easy to code, and can be used in conjunction with high-level language code already used. Therefore future coding could use SQL, while existing applications can remain as they are.

For further comparison, see the section on "DDL Recommendations" on page 2-9.

Considerations for Data Conversion of System/36 Files

In order to be processed by SQL (or to reside in an SQL collection), database files must be externally described. The System/36 does not provide this facility but uses only program described database files, where the files may or may not be linked to a data dictionary. Regular data management, as invoked by HLL programs, does not make use of the data dictionary links. However, in order to use SQL, System/36 files must be converted to AS/400 externally described files.

Note: If the files are not converted, SQL access is limited to plain "one record = one field" processing. Non-converted files cannot reside in an SQL collection.

System/36 File Library

Following System/36 to AS/400 migration, the System/36 files reside in a library (usually QS36F) where they can be accessed by System/36 Environment programs. This library is the equivalent of the System/36 VTOC. Within the invocation of the System/36 Environment (STRS36), the System/36 Environment programs have access to only one System/36 file library. The Change System/36 command (CHGS36) can be used to switch the System/36 file library between System/36 Environment invocations.

System/36 File Types

In the System/36 file library, you find all the conventional System/36 file types: sequential(S), indexed(I), direct(D) and alternate indexes(X). The "externally described" tag of a file, shown by the CATALOG procedure, just means "IDDU-linked" and *not* AS/400 externally described. Group file names (with a period in the file name) have to be enclosed in double quotes when used in AS/400 commands. The System/36 file library is a regular AS/400 library and therefore conventional System/36 files may be intermixed with AS/400 files.

Conversion Methods

When using SQL for System/36 files, you can consider two different conversion methods, described below.

Assumptions

- Use of SQL is limited to application extensions only (that is, new programs).
- The existing System/36 application should continue to run as is in the System/36 Environment.
- No changes, or very few, should be made to programs and procedures in the existing System/36 application.
- No permanent data duplication is wanted, such as the same data in an SQL library *and* in a System/36 file library.

Method 1: All files in the application are moved (and converted) into an SQL collection. Before running the application, the System/36 Environment is reconfigured (CHGS36) to have the System/36 file library being the SQL collection.

Method 2: Only selected System/36 files are converted and put back to the standard System/36 file library which is not an SQL collection (library).

The choice of either of the above methods depends on the nature of the application and its context. The suggested approach is to start with method 2, and then, if applicable and when ready, switch to method 1. Table 3-4 on page 3-47 summarizes the major aspects of the two methods.

Table 3-4. System/36 File Conversion Methods

METHOD 1	METHOD 2
Full SQL support - ie automatic catalog (data dictionary) maintenance and commitment control.	Conversion may be done in steps by converting only data which is involved in a new application.
All application data is externally described.	No implications to the rest of the application or to other applications.
	Small step-by-step conversion effort.
	No SQL support for data dictionary.
	No SQL support for commitment control.
Large (once but all together) conversion effort.	Only part of application data is externally described.

Data Conversion Steps

The following steps should be used as a reference when converting data. The general sequence that should be followed is: If the System/36 file in question has alternate indexes, then convert the parent file first. Go through all the steps for every parent file. Steps 2 - 6 and Step 8 apply only to parent files (S,I,D).

Step 1: Save the file you plan to convert.

Step 2: When you create an SQL collection (library), all the necessary entries for a data dictionary are created automatically. In addition, the library is flagged "SQL" in order to host only AS/400 externally described files.

Step 3: It is suggested that you create or move all the necessary IDDU definitions into the above data dictionary. By doing so, you will not have duplicate data dictionaries when switching to method 1.

Step 4: For non-IDDU defined files you will have to create the definitions. Already existing IDDU definitions should be moved to the SQL data dictionary. To find out which definition is used, you may use the Display File Description command (DSPFD) or use the IDDU cross-reference information facility. Before moving (copy and then delete from original data dictionary), unlink the files (LNKDTADFN command or IDDULINK procedure).

Step 5: Either by menu options or by the command "WRKDBFIDD" go to "Work with Data Base Files using IDDU". Select option "CREATE". When using method 1, create the file into the SQL collection; when using method 2, create the file into the original System/36 file library using a new file name. The IDDU file definition to be used is located in the SQL data dictionary (see step 4). This "Create with IDDU operation" will result in an AS/400 externally described file.

Note that this procedure will only apply to single format and non-keyed files. The procedure for multi-format and keyed files is more complex and not described here.

Step 6: Using the Copy File command, copy the data from the original file to the new file:

```
CPYF FROMFILE(lib/filename) TOFILE(lib/filename)
     MBROPT(*ADD) CRTFILE(*NO) FMTOPT(*NOCHK)
```

Step 7: If the original parent file had alternate indexes, then these indexes have to be recreated in the converted version. When using method 1, you have to create an SQL index (or logical file if you need to substring fields) within the SQL collection for every alternate index file, then delete the alternate indexes for the original parent file. When using method 2, delete all the alternate indexes for the original parent file; then recreate them in the same library by chaining them to the new (renamed) parent file; for this purpose use the BLDINDEX procedure.

But be aware that where alternate indexes were allowed to be built over partial fields, this is not possible with an externally described file.

Step 8: For both methods, delete the original file. When using method 2, rename (RENAME procedure or RNMOBJ command) the new file to the old file name.

Conclusion

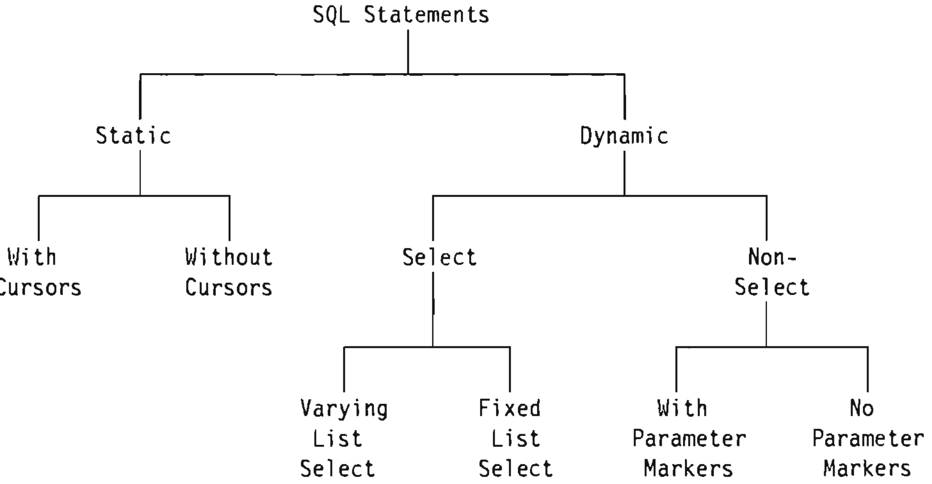
When converting System/36 files with the above methods, the existing programs in the System/36 Environment do not have to be changed. They can be run against System/36 files (program described) and also against AS/400 files (externally described).

If you plan to use SQL with HLL programs, consult the appropriate paragraphs in this chapter. The programs have to be written in COBOL/400, RPG/400, C/400, FORTRAN/400 or PL/I. Existing System/36 programs must be migrated to one of the above languages.

4. Static and Dynamic SQL

SQL statements can be run in two environments - static and dynamic. A complete list of the statements which are allowed in each environment appears in the "Dynamic SQL Applications" chapter of the *Programming: Structured Query Language Programmer's Guide (SC21-9609)*. It is important to note that programs which run under commitment control in either static or dynamic environment cannot execute SQL data definition statements (for example CREATE TABLE or CREATE or DROP INDEX).

The following diagram gives a clear representation of the methods available with SQL, covering static and dynamic:



Static SQL

Static SQL statements are embedded within a program and are prepared during the precompile process and executed later. Rows can be processed in static SQL in one of two ways - either using cursors or without cursors.

Processing Without Cursors

SQL cursors allow the retrieval of more than one row from a single SELECT statement (at one time). However, it is not always appropriate to process with cursors. The following are some of the considerations when processing without a cursor.

Retrieving

A single row SELECT statement can be used if it is known that there will be only one row in the result set. For example, if a row is being accessed by a unique index (employee number in this case):

```
SELECT EMPNO, LASTNAME, WORKDEPT, SALARY
INTO :EMPNO, :LASTNAME, :WORKDEPT, :SALARY
FROM TEMPL
WHERE EMPNO = :host-variable
```

© Copyright IBM Corp. 1990

Another example of a selection that would return only one row is a SELECT statement that does a built-in function, for example, find the minimum and maximum salaries earned by all female employees in a particular department.

```
SELECT MIN(SALARY), MAX(SALARY)
INTO :MINSAL, :MAXSAL
FROM TEMPL
WHERE SEX = 'F'
AND DEPTNO = :host-variable
```

A single row selection might also be a join. In the following example we find the employee's first and last name and the name of the department in which the employee works:

```
SELECT FIRSTNAME, LASTNAME, DEPTNAME
INTO :FIRSTNAME, :LASTNAME, :DEPTNAME
FROM TEMPL, TDEPT
WHERE TEMPL.EMPNO = :host-variable
AND TEMPL.DEPTNO = TDEPT.DEPTNO
```

If more than one row is returned from this type of select, the program will receive a negative SQLCODE. However, data will be returned to the host variables and the program will continue to execute.[1]

Updating

Without using a cursor you can do either a single row update if the value in the WHERE clause is unique, or a mass update if more than one row matches the value in the WHERE clause.

For example, to give all employees in department 'D01' a 10% salary increase:

```
UPDATE TEMPL
SET SALARY = SALARY + (.1*SALARY)
WHERE DEPTNO = 'D01'
```

To give employee number 000020 a 20% salary increase:

```
UPDATE TEMPL
SET SALARY = SALARY + (.2*SALARY)
WHERE EMPNO = '000020'
```

Joined tables cannot be updated.

If you are using commitment control, remember that a maximum of 32767 rows can be locked within one unit of recovery. If your mass update affects more than 32767 rows you will get a negative SQLCODE returned to the program, and the statement results are rolled back. In other words, the rows involved in the mass update would be rolled back, but not the rest of the unit of recovery.

[1] SAA SQL tells us that the variables are returned, but the row used is unpredictable.

Deleting

You can delete a single row from a table if the value in the WHERE clause is unique (employee number is unique), again without using a cursor.

```
DELETE FROM TEMPL
WHERE EMPNO = :host-variable
```

A mass deletion can be done if the value in the WHERE clause is not unique. In the following example, all projects for department D11 are deleted:

```
DELETE FROM TPROJ
WHERE DEPTNO = 'D11'
```

If no WHERE clause is specified at all, all the rows in the table will be deleted. But the catalog description of the table is retained:

```
DELETE FROM TEMPL
```

If you are using commitment control, remember that a maximum of 32767 rows can be locked within one unit of recovery. If your mass delete affects more than 32767 rows you will get a negative SQLCODE returned to the program, and the statement results are rolled back. In other words, the rows involved in the mass delete would be rolled back, but not the rest of the unit of recovery.

Inserting

Single rows can be inserted into a table. The following example adds a new department to the department table:

```
INSERT INTO TDEPT VALUES ('F11','EDUCATION','000110','D01')
```

A mass insert can be done by using a subselect within the INSERT. The following example makes a new copy of the department table:

```
INSERT INTO NEWDEPT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT)
SELECT * FROM TDEPT
```

A common use for mass insertions is to create a temporary table in a program, and insert the resulting set from a complex join into it, for further processing within that program. The next example will select all employees who are working on projects, and look up their department name, and the names of all the projects they are working on, and put all this information into a temporary table.

The advantage of having this information in a table is that the joined information can be worked on with UPDATE and DELETE. A normal cursor select for update cannot do updating if the SELECT statement includes a join. However, note that you are not updating the base table, only the copied version.

```
CREATE TABLE TEMP (FIRSTNAME, LASTNAME, DEPTNAME, PRNAME)

INSERT INTO TEMP (FIRSTNAME, LASTNAME, DEPTNAME, PRNAME)
SELECT FIRSTNAME, LASTNAME, DEPTNAME, PRNAME
FROM TEMPL, TDEPT, TPROJ, TEMPRACT
WHERE TEMPL.EMPNO = TEMPRACT.EMPNO
AND TEMPL.DEPTNO = TDEPT.DEPTNO
AND TEMPRACT.PROJNO = TPROJ.PROJNO
```

If you are using commitment control, remember that a maximum of 32767 rows can be locked within one unit of recovery. If your mass insert attempts to add

more than 32767 rows, you will get a negative SQLCODE returned to the program, and the statement results are rolled back. In other words, the rows involved in the mass insert would be rolled back, but not the rest of the unit of recovery.

Processing With Cursors

SQL is better suited to "set-at-a-time" operations, rather than single row operations, when compared to high-level languages. SQL "set-at-a-time" operations in HLL programs require the use of cursors. Cursors are processed sequentially and can be in ascending or descending sequence.

Up to a maximum of 32767 rows can be locked in one unit of recovery. When this number is reached, and you are processing with a cursor, you will get a negative SQLCODE and should then execute a commit with hold, which will allow processing to re-commence from the point where the negative SQLCODE occurred.

The following are examples of how cursors are used in applications.

Retrieval

If you are going to perform a selection that will result in more than one row in the result set, you will need to use a cursor to process the rows. All the rows that you want to process are selected with a DECLARE CURSOR and SELECT statement. After that you fetch the rows sequentially, and process them one at a time. The SELECT statement can specify a join across multiple tables.

```
DECLARE CURSOR1 CURSOR FOR
SELECT EMPNO, FIRSTNAME, LASTNAME, DEPTNAME
FROM TEMPL, TDEPT
WHERE TEMPL.DEPTNO = TDEPT.DEPTNO
ORDER BY EMPNO

OPEN CURSOR1

FETCH CURSOR1
INTO :EMPNO, :FIRSTNAME, :LASTNAME, :DEPTNAME
```

Updating

An extension to SQL/400 allows cursors that are going to be updated to also have an ORDER BY clause specified in the SELECT statement. Other implementations of SQL and SAA do not allow this. The maximum size of the fields that you are going to be ordering by cannot exceed 256 bytes. If the ORDER BY field size is greater than 120 bytes, the cursor is considered to be read only and you cannot also specify FOR UPDATE OF.

If ORDER BY is specified in a cursor SELECT statement for which you plan to perform a subsequent cursor FETCH statement and a cursor UPDATE statement, you *must* also code a FOR UPDATE OF clause in the SELECT statement, otherwise you will get a negative SQL return code when you attempt the update.

In this example, note the ORDER BY **8** and FOR UPDATE OF **9** clauses.

```
DECLARE CURSOR1 CURSOR FOR
SELECT PARTNO, DESCRIPTION, ONHAND
FROM INVENTORY
WHERE PARTNO BETWEEN 209 AND 280
ORDER BY DESCRIPTION 8
FOR UPDATE OF ONHAND 9

OPEN CURSOR1

FETCH CURSOR1 INTO :WPARTNO, :WDESCR, :WONHAND

UPDATE INVENTORY
SET ONHAND = :WONHAND+100
WHERE CURRENT OF CURSOR1
```

If an ORDER BY clause is not specified in the SELECT statement, you have the option of coding the FOR UPDATE OF clause in the SELECT, or omitting it. Both ways will allow subsequent updating of the cursor.

The FOR UPDATE OF clause limits the columns that can be updated to those listed. If the clause is omitted, all columns can be updated including columns on which there are indexes. The omission of the FOR UPDATE OF clause could have performance implications because SQL/400 will not use an index when selecting a row. SQL will not use an index over columns that are to be updated because an update to that column could cause it to be processed again if the update changes the column value to be different to the original value.

A cursor UPDATE statement can update columns omitted from the select list, but the INTO clause of a FETCH statement can only reference columns in the select list.

In this example the ONHAND column is updated without it being specified on the selected column list:

```
DECLARE CURSOR1 CURSOR FOR
SELECT PARTNO, DESCRIPTION
FROM INVENTORY
WHERE PARTNO BETWEEN 209 AND 280
ORDER BY DESCRIPTION
FOR UPDATE OF ONHAND

OPEN CURSOR1
FETCH CURSOR1 INTO :WPARTNO, :WDESCR

UPDATE INVENTORY
SET ONHAND = ONHAND+100
WHERE CURRENT OF CURSOR1
```

If the cursor is considered update capable, rows are locked as they are read. If the program is executing under *CHG and a row is not updated, the lock is released as the next row is read. If the program is executing under *ALL, all rows are locked after they are read. In addition, under all commitment control levels, including *NONE, if the cursor is considered update capable, the row that the cursor is currently pointing to is locked against all other updates from the same program except for UPDATE WHERE CURRENT of the cursor (that is, the same one that did the fetch).

A cursor is considered update capable if it has a FOR UPDATE OF clause, an UPDATE WHERE CURRENT clause, a DELETE WHERE CURRENT clause, or there is an EXECUTE or EXECUTE IMMEDIATE statement in the program and the SELECT statement has no JOIN, UNION or GROUP BY. A cursor is considered read only if it has no FOR UPDATE OF clause and no EXECUTE or EXECUTE IMMEDIATE statement in the program, or if it has a JOIN, UNION or GROUP BY. In all cases, a cursor declared with an ORDER BY and omitting the FOR UPDATE clause, is read only.

When a program is executing under *CHG, the cursor is considered read only and no locks are placed on any of the selected rows. However, if the program is executing under *ALL, the locks will prevent access of the rows from another job.

Notes Regarding the FOR UPDATE OF Clause

- A cursor is considered update capable if it has either a FOR UPDATE OF clause, or if there is an EXECUTE or EXECUTE IMMEDIATE statement in the program, or an UPDATE WHERE CURRENT OF clause, or a DELETE WHERE CURRENT OF clause, and the SELECT statement has no JOIN, UNION and GROUP BY clauses.
- A cursor is considered to be read only if it has no FOR UPDATE OF clause and no EXECUTE or EXECUTE IMMEDIATE statement in the program, or if it has a join or a UNION or a GROUP BY clause.
- If the columns listed in the FOR UPDATE OF clause include the index, or if both the FOR UPDATE OF and ORDER BY clauses are omitted, thereby making all the fields potentially update capable, the *index will not be used* when the selection is performed. This could result in performance implications.
- Use the FOR UPDATE OF clause only when you are going to do the update using UPDATE WHERE CURRENT OF CURSOR, and specify *only* the fields that you want to update.
- If the update is going to be done using UPDATE WHERE = :host-variable, rather than UPDATE WHERE CURRENT OF CURSOR, leave the FOR UPDATE OF clause out of the select. If this is not done you may get a record lockout (SQLCODE -913) when you try to do the update, because an update lock was taken on the row at fetch time, and can only be updated by the same cursor. This should be avoided if at all possible for data integrity and performance.
- You cannot use FOR UPDATE OF if the SELECT statement includes:
  1. AVG, MAX, MIN, SUM or COUNT
  2. GROUP BY or DISTINCT
  3. Union
  4. A join of two or more tables.

  In these cases, the cursor is always read only.

Deleting

You can delete the current row of a cursor by declaring the cursor with a FOR UPDATE OF clause as in updating a cursor. After the row that you want to delete has been fetched, you then specify DELETE WHERE CURRENT OF CURSOR. The row that the cursor is pointing at will be deleted and the next FETCH statement will then point at the next row to be fetched.

```
DECLARE CURSOR1 CURSOR FOR
SELECT PARTNO, DESCRIPTION
FROM INVENTORY
FOR UPDATE OF ONHAND

OPEN CURSOR1
FETCH CURSOR1
INTO :WPARTNO, :WDESCR

- display confirmation screen to user -

DELETE FROM INVENTORY
WHERE CURRENT OF CURSOR1
```

Dynamic SQL

Dynamic SQL statements are prepared and executed within a program while the program is executing. Dynamic SQL can be used by more advanced application programs, to build SQL statements that are not known at compile time, including SQL statements that support host variables by the use of parameter markers.

Authorization to run dynamic statements is checked at execution time during the processing of the OPEN CURSOR statement, during preparation time and during the processing of the statements that reference the open cursor.

Interactive SQL is a good example of dynamic SQL. Neither the form of the SQL statement, nor the identity of the tables that are going to be used is known until execution time. The SQL statement is contained within host variables rather than being coded into the program. Examples in this section show applications which use the following SQL statements dynamically:

```
SELECT,  DESCRIBE,  DECLARE CURSOR,
EXECUTE IMMEDIATE,  PREPARE  and  EXECUTE
```

Dynamic SELECT Statements

Dynamic SELECT statements are of two basic types: fixed-list selections and varying-list selections.

Fixed-List SELECT statements

Dynamic fixed-list SELECT statements are supported by RPG, COBOL, C/400, FORTRAN/400 and PL/I. Fixed-list SELECT statements return data of a predictable type and number, and you can therefore design host variables to accommodate the data as you would for any static select. The columns that are going to be selected must be fixed at compile time. However, the WHERE clause may be specified at execution time.

A program that would be a good candidate for being written as a Dynamic fixed-list selection would be one that had so many different options in the

WHERE clause that it would not be feasible to code a different SQL SELECT statement for each possibility. For example, if you did not want to give a user access to the full power of Interactive SQL, but they did have a need to perform a variety of queries on a specific table, you could design a program where all the columns in the table would be accessible and the user could enter his own WHERE clause on the screen. See Figure 4-1 for an example screen for such a program.

The user would key in a column name from the choices displayed, an operator, and a constant enclosed in quote marks, which together would form the WHERE clause. This WHERE clause would then be combined with the constant 'SELECT * FROM TEMPL WHERE' into one host variable (now holding an SQL statement) that would then be prepared at execution time. After the statement has been prepared, you then declare the cursor, open the cursor, fetch each row into your predefined host variables, and continue processing as if it were a static program.

```
SCREEN1          EMPLOYEE SELECTION

                                            EMPNO
                                            FIRSTNME
                                            LASTNAME
SELECT ALL EMPLOYEES WHERE _____________    DEPTNO
                                            PHONENO
                                            HIREDATE
                                            SEX
                                            BIRTHDATE


                                            =
                                            ¬=
                                            <
            COMPARISON OPERATOR _____       >
                                            <=
                                            >=

             CONSTANT _________________
          (Enclose constant with single quotes)

F3=EXIT
```

Figure 4-1. Example Screen For a Dynamic SQL Program

Following is a sample RPG program showing the use of a dynamic fixed-list select. To get a list of all employees whose surnames were from "M" to "Z", the user would key LASTNAME in the first screen field, > = in the comparison operator field, and "M" in the constant field. The SELECT statement would be

```
SELECT * FROM TEMPL WHERE LASTNAME >= 'M'.
```

To further sophisticate the program, an AND and OR line could be included on the screen after the constant field.

```
     H
     FPROG06  CF  E                    WORKSTN

     E                    ARRAY   1   1 26

     IWHERE      DS
     I                                 1  8 COLUMN
     I                                 9  9 BLANK1
     I                                10 11 OPERAT
     I                                12 12 BLANK2
     I                                13 27 CONST

     C                     MOVE ' '       BLANK1  1
     C                     MOVE ' '       BLANK2  1
     C                     EXFMTSCREEN1
     C          *INKC      DOWNE'1'
     C                     MOVELARRAY     HOSTVR 53
     C                     MOVE WHERE     HOSTVR
     C/EXEC SQL
     C+         PREPARE S1 FROM :HOSTVR
     C/END-EXEC
     C/EXEC SQL
     C+         DECLARE C1 CURSOR FOR S1
     C/END-EXEC
     C/EXEC SQL
     C+         OPEN C1
     C/END-EXEC
     .
     .         Dowhile not end of cursor
     .
     C/EXEC SQL
     C+         FETCH C1 INTO :EMPNO, :FIRST, :INIT,
     C+         :LAST, :DEPT, :PHONE, :HIRE,
     C+         :JOB, :EDUC, :SEX, :BIRTH,
     c+         :SALARY
     C/END-EXEC
     .
     .Print or display row then fetch next row
     .
     C/EXEC SQL
     C+         CLOSE C1
     C/END-EXEC
     C                     EXFMTSCREEN1
     C                     END
     C                     SETON                        LR

**
SELECT * FROM TEMPL WHERE
```

Figure 4-2. Program to Illustrate the Use of Dynamic Fixed-List SELECT

Varying-List SELECT statements

Dynamic varying-list SELECT statements are only supported by PL/I and C/400. RPG, COBOL and FORTRAN may use varying-list SELECT statements, but they must have PL/I or C/400 set up the parameters in the SQL descriptor area (SQLDA). Varying-list selections return data of an unpredictable format and an unpredictable number of columns; therefore the SQLDA needs to be used in the program to pass information about the database back to the application, so that storage can be allocated for the retrieved data at execution time. Neither RPG, COBOL nor FORTRAN have the capability of allocating storage at execution time. However, all of these languages could call a PL/I or C/400 subroutine to execute the SELECT. Detailed steps using DESCRIBE and SQLDA can be found in the *Programming: Structured Query Language Programmer's Guide (SC21-9609)*.

Dynamic Non-SELECT Statements

Dynamic non-SELECT statements can be executed by RPG, COBOL, C/400, FORTRAN and PL/I programs. The only data that is returned to the program by these statements is a return code. Dynamic non-SELECT statements can be of two types: those that contain parameter markers and those that do not.

Statements Containing No Parameter Markers

Statements containing no parameter markers can be run dynamically using EXECUTE IMMEDIATE :host-variable. The host variable (named HOSTVC in this example) would contain a complete SQL statement such as:

```
HOSTVC = "CREATE INDEX PARTX1 ON INVENTORY (PARTNO)"
  or
HOSTVC = "UPDATE INVENTORY SET ONHAND = 500 WHERE PARTNO = 207"
```

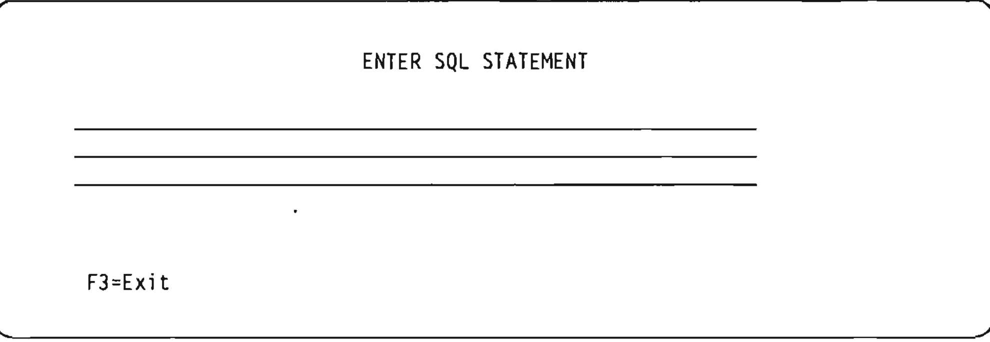
Authorization is checked at execution time. The run time authorization ID must have the authority to run the SQL statement specified by the EXECUTE IMMEDIATE statement.

Here is a sample RPG program using EXECUTE IMMEDIATE to dynamically execute an SQL statement entered on the screen:

```
H
FPROG03  CF  E                    WORKSTN

C                     EXFMTSCREEN1
C            *INKC    DOWNE'1'
C/EXEC SQL
C+           EXECUTE IMMEDIATE :HOSTVC
C/END-EXEC
C                     EXFMTSCREEN1
C                     END
C                     SETON                     LR
```

Figure 4-3. Program to Illustrate the Use of EXECUTE IMMEDIATE

HOSTVC is externally described in the screen specifications as a 256 character input field, into which the SQL statement is entered.



Restrictions for Use of EXECUTE IMMEDIATE: In the above example the program would have to be running without commitment control in order to execute Data Definition Language (DDL) statements like the CREATE INDEX statement; however, it could be running under commitment control to execute the UPDATE statement, for example.

Statements Which Contain Parameter Markers

Statements which contain parameter markers (indicated by a "?") can be run using PREPARE and EXECUTE statements. The statement is prepared once, and each time it is executed, different values can be put into the parameters.

Authorization is checked at execution time. The run time authorization ID must have the authority to run the SQL statement specified by the EXECUTE statement.

An example is a program that is used to enter new inventory items into the inventory table. We can set up a host variable named HOSTVG with the following value:

```
HOSTVG = "INSERT INTO INVENTORY VALUES (?,?,?)"
```

Then prepare the statement as follows:

```
PREPARE INSERTP FROM :HOSTVG
```

The PREPARE statement only needs to be done once during execution of the program.

We then set up three more host variables, PARTNO, DESCR and ONHAND, and move different values into these fields for each new inventory item that you want to add to the inventory table.

```
PARTNO = 222
DESCR  = 'WASHER'
ONHAND = 250
```

Then execute the statement as follows:

```
EXECUTE INSERTP USING :PARTNO, :DESCR, :ONHAND
```

If the input to the host variables comes from the screen and the EXECUTE statement is updating a table, it is advisable to commit with hold periodically after the EXECUTE statement, in order to release locks. If HOLD is specified, the prepared statement is preserved. If HOLD is *not* specified, the statement would have to be prepared again before the next EXECUTE statement.

This is a sample RPG program using PREPARE and EXECUTE to dynamically execute SQL statements which insert new items into the inventory table.

```
    H
    FPROG04  CF  E            .       WORKSTN

    E                    ARRAY   1   1 35

    C                     MOVEAARRAY     HOSTVG 35
    C/EXEC SQL
    C+        PREPARE INSERTP FROM :HOSTVG
    C/END-EXEC
    C                     EXFMTSCREEN1
    C         *INKC       DOWNE'1'
    C/EXEC SQL
    C+        EXECUTE INSERTP USING :PARTNO, :DESCR, :ONHAND
    C/END-EXEC
    C/EXEC SQL
    C+        COMMIT HOLD
    C/EXEC SQL
    C                     EXFMTSCREEN1
    C                     END
    C                     SETON                        LR

**
INSERT INTO INVENTORY VALUES(?,?,?)
```

Figure 4-4. Program to Illustrate the Use of PREPARE and EXECUTE

The variables PARTNO, DESCR and ONHAND are externally described in the screen specifications and contain the values that are substituted for the ? parameter markers.

```
          Enter New Inventory Items

   Enter part number  ___

   Description        ________________________

   Qty on hand        ____

  F3 = Exit
```

Note that the only statements that can be prepared are data manipulation language statements. Data definition and data control statements, for example CREATE or GRANT, cannot be dynamically prepared.

Dynamic SQL Performance

The execution time overhead for running dynamic SQL is equal to that of an SQL precompile and run. It is therefore recommended to use dynamic SQL only in cases where static SQL *cannot* be used. For a SELECT statement referencing 30 columns from one table with 3 WHERE conditions, the dynamic SQL overhead at run time is 2 clock seconds on a B30 with nearly all of the 2 seconds being CPU time. This measurement assumes the SQL dynamic system code is in memory.

Note: The SQL dynamic overhead could be greater for longer or more complex SQL statements.

5. SQL Performance

Good application design includes the efficient use of machine resources. As such, any programmer can write a program, but for the program to execute in a manner that is acceptable to the end user, it must be elegant in operation, and must execute with adequate response time.

This chapter discusses many topics related to programming for optimizing performance of an application. These not only apply to SQL, but are also a good guide for other products that use the AS/400 database techniques, including OPNQRYF, and OS/400 Query Management.

The following topics are discussed:

1. The Nature of Database I/O

   An introduction to overall processes, functions, and terms used in processing requests for data.

2. Design Guidelines

   This topic covers techniques that can be used in SQL programs to achieve better performance. There is also a section that discusses general AS/400 database design guidelines.

3. Data Management Methods

   This covers the algorithms used to retrieve data from the disk and includes a discussion of access paths, row selection techniques, and reuseability of Open Data Paths (ODP).

4. Optimizer

   The Optimizer is the facility that decides how to gather data which should be returned to the program. This section covers the techniques and rules employed by the Optimizer for performing this task including cost estimating, access plan validation, join optimization and subquery optimization.

5. Analyzing Performance Problems

   If you have already coded an SQL application and are interested in improving its performance then you can use this topic to guide you through the system modules called by each step. This discusses Job Logs, Interactive SQL, Job Traces, and Performance Tools. This will help in identifying when indexes are created or used, and what can be done to improve the performance.

6. Positioning

   The topic compares SQL performance at Release 3.0 with other approaches, and offers performance advice for various application scenarios.

7. Release 3.0 Performance Enhancements

   To correctly position some of the changes that have been made to the database support in OS/400 Release 3.0, this section outlines those changes made that will improve performance of certain SQL functions.

© Copyright IBM Corp. 1990

Special Note

This chapter contains a lot of information that may take some study. Please allocate plenty of time for reading. It may be helpful to have access to an AS/400 while studying so that you can experiment and verify some of the information provided here.

The Nature of Database I/O

This topic introduces the fundamental concepts for data access to the AS/400 database. Understanding database access techniques is an important prerequisite to understanding the implications of database performance.

The first part of this topic deals with the processes involved in identifying the structure of the data as it exists in the database, and its relationship with a query request. This introduces the concepts of binding and the access plan. The second part discusses the tools used for retrieval of the data itself.

Creating the Access Plan

In creating a static SQL program, the precompilation phase creates an important internal control structure called an *access plan*. The access plan contains two main things:

- A list of objects used by the SQL statement required to create an open data path (ODP)
- A list of tasks that need to be performed to create and open a temporary "logical file" resulting in an ODP.

Essentially creating an ODP is like opening a file. Note that the ODP is not actually created at precompile time. As access to the file is only required at runtime, all that happens at precompile time is working out the steps the system will take when the file is eventually opened (that is, the ODP is created).

This process is very similar to the process that would occur had a programmer issued a CRTLF command and then had a program open this logical file. The difference is that some internal objects, such as the format object and file objects which are used by OS/400 to manage permanent files, are not created.

The access plan is actually stored with the compiled SQL program object. The process of creating this access plan is called *binding*.

Other query facilities on the AS/400 which use similar techniques to SQL to access data, have different methods of binding and storage of the access plans.

Dynamic SQL Programs: Dynamic SQL is a little different to static SQL, as there is extra work done at runtime to cater for the added flexibility offered by the dynamic functions. The binding process is done at runtime, when the program must access the nominated table. It is only at execution time that the SQL runtime support knows what the full SQL statement will be, so only then can binding complete.

OPNQRYF: OPNQRYF binds and creates a temporary access plan when the OPNQRYF CL command is issued. If an OPNQRYF statement is re-executed in the same program, the access plan cannot be shared between jobs, as it is temporary, and rebuilt every time the command is executed. This is different to permanent access plans created by other facilities, like SQL. There is a performance overhead associated with having to recreate the access plan every time the OPNQRYF command is executed.

AS/400 Query: AS/400 Query uses the same technique as OPNQRYF when the user presses F5=Report to show the current status of the query being developed. In addition, binding occurs when the query is saved. A permanent access plan is created and is stored in the *QRYDFN object.

PC Support File Transfer: The transfer request, when executed, is similar to OPNQRYF. When it is saved, it remains as a PC source file, and therefore cannot contain an access plan. When re-executing a stored transfer request, the request is bound and the ODP is created.

OS/400 Query Management: OS/400 Query Management uses SQL runtime support for binding and execution. Only the source statement is stored when the *QMQRY object is created. To implement the flexibility of Query Management, all binding occurs at runtime. This is quite similar to Dynamic SQL.

Physical and Logical Files: A program which uses a physical file or logical file operates differently to the methods described above. There is a binding process, but instead of creating an access plan, the object created is called a *prototype ODP*. At runtime, this prototype ODP is copied into the actual ODP. The prototype ODP can be considered as a *limited function* access plan, as it locks the user into a particular data access method.

Rebinding: A prototype ODP is different to a static SQL program using an access plan. With an access plan, the user is not locked into the data access method chosen at bind time. If before execution there is a change to the structure of the database, the access plan validation routine may cause the SQL statement to be rebound. This means that the access plan can adapt to the database environment, and take advantage of any new indexes that will help it to improve statement execution performance. This is called *rebinding.* Rebinding generally occurs when there is a change to the structure of the objects related to the table, such as the creation or deletion of an index.

Not only are SQL programs able to take advantage of rebinding, but so can any query tool that stores an access plan before execution, such as AS/400 Query and Query/38.

AS/400 Query has the capability to run with a temporary plan should the original permanent copy become out of date. However, the permanent copy of the access plan will not be updated again until a programmer saves the *QRYDFN object again.

Late Binding: Late binding is a process only available on SQL/400. It allows the bind process to complete after the program is precompiled. This means that the precompile will successfully complete if a table is not available when the precompile occurs (either not on the system or not in the library list). For a normal HLL program, the file must be available at compile time so field names can be used in the program. With Late Binding, the resolution of column names

waits until the program is first executed. If the table is still unavailable, then the program returns a negative SQLCODE when executing the first SQL statement.

Late binding can only occur when the HLL program does not need to copy the column definitions for its own use. If the host variables are defined in some other way like: explicit definition, definition from the workstation file, or if there is no need for the host variable definition, then late binding can occur.

SQL Views: An SQL view, while having the attributes of a logical file, when created actually creates an access plan. The access plan is built to optimize access to the underlying table. This provides greater flexibility than a logical file.

This access plan will be used when a HLL program accesses the view. It creates an ODP similar to that which would be created when the view is referenced by SQL. At bind time an SQL program which references the view will cause the view access plan to be merged with the SQL statement access plan. This composite access plan will then be stored in the program object.

This capability to create access plans in a view logical files allows HLL programs to benefit from the same rebinding capabilities available to SQL programs.

The following table summarizes the points just discussed.

Table 5-1. Summary of Access Plan Creation

Facility	Permanent or Temporary Access Plan	When is the Access Plan Created	Where is the Access Plan Stored
Static SQL	Permanent	Precompile Time	With Program
Dynamic SQL	Temporary	When executed	n/a
OS/400 Query Management	Temporary	When executed	n/a
AS/400 Query	Temporary and Permanent	F5 = Report When saved	 In *QRYDFN object
Query/38	Permanent	When saved	In *PGM (QRYEXC) object
OPNQRYF	Temporary	When executed	n/a
PC Support File Transfer	Temporary	When executed	n/a
SQL Views	Permanent	CREATE VIEW	In *FILE (LF) object
Physical and Logical Files	Creates a prototype ODP	CRTLF, CRTPF	In *FILE object

Multiple Access Plans: The discussion so far has been limited to one access plan per SQL program. Often, a program will have multiple access plans. Each access plan refers to a single ODP. If an SQL program contains, for example, two SQL update statements then two ODPs are required and two access plans are created at precompile time. Generally speaking, an access plan is created (thus ODP available at runtime) for each statement in the SQL program. For instance, each of the following requires a separate ODP even if they are all together in the same program:

- An OPEN statement
- A SELECT INTO statement

- An INSERT INTO statement with a VALUES clause
- An UPDATE statement with a WHERE condition
- An UPDATE statement with a WHERE CURRENT OF cursor[1] and SET clauses that refer to operators or functions
- A DELETE statement with a WHERE condition
- An INSERT statement with a subselect (requires two ODPs).

One objective of performance tuning of an application is to minimize the number of opens. That is, to minimize the number of ODPs created.

Data Retrieval

At runtime, applications which have an access plan use a special optimization routine to return data in the most efficient manner. The data returned is based on a query request, such as an SQL SELECT, and may return none, one, or many rows. A *query* is a general term that refers to any operation that uses an access plan to access data.

Traditional HLL program I/O statements access data one record at a time. We can use them in conjunction with logical files to provide relational operations such as:

- Record selection
- Sequence
- Join
- Project.

This is often the most efficient manner for data retrieval.

Query products are best used when logical files are not available for our data retrieval requests, or if there are functions we require that logical files cannot support, are too difficult to write, or would perform poorly including:

- Distinct
- Group by
- Subquery
- Like.

The query products use something more sophisticated to perform these functions. It is done with the access plan in combination with a high-function query routine. The advantage of this facility is that because the query requests are created at runtime, there are often fewer permanent access paths than are required for multiple logical files.

OS/400 Query Component: This specialized query routine is internal to OS/400, and is referred to as the OS/400 Query component in this document (not to be confused with the licensed program product, AS/400 Query).

[1] **Cursor:** A cursor in SQL is a named control structure used by an application program to position to a row of data within a table or view.

There are nine common facilities on the AS/400 system that use this OS/400 Query component. These are:

- OPNQRYF
- SQL/400 runtime support
- AS/400 Query runtime support
- Query/38 runtime support
- PC Support File Transfer
- OS/400 Query Management
- Office (for document searches)
- Performance Tools (for report generation)
- Q&A Database.

The difference between the terminology of SQL/400, AS/400 Query, Query/38 **versus** SQL/400 *runtime support*, AS/400 Query *runtime support*, Query/38 *runtime support* is that the former group refer to the names of the licensed program products. What we are really interested in is the support which enables programs in each of these environments to be executed. The licensed program products are not required to execute each of these, as the runtime support comes with OS/400 and not the licensed program product.

Figure 5-1 on page 5-7 helps to explain the relationship between the various facilities that use the OS/400 Query component at runtime, and the way in which traditional HLL program I/O requests are satisfied. Note that HLL program I/O requests go directly to the data base support to retrieve the data. Query product requests call upon the OS/400 Query component, which uses the Optimizer before calling the data base support to create the ODP to the data. Once an ODP has been created, there is no difference between HLL I/O requests, and I/O requests of these query products. Both send requests for data to the Database Support.

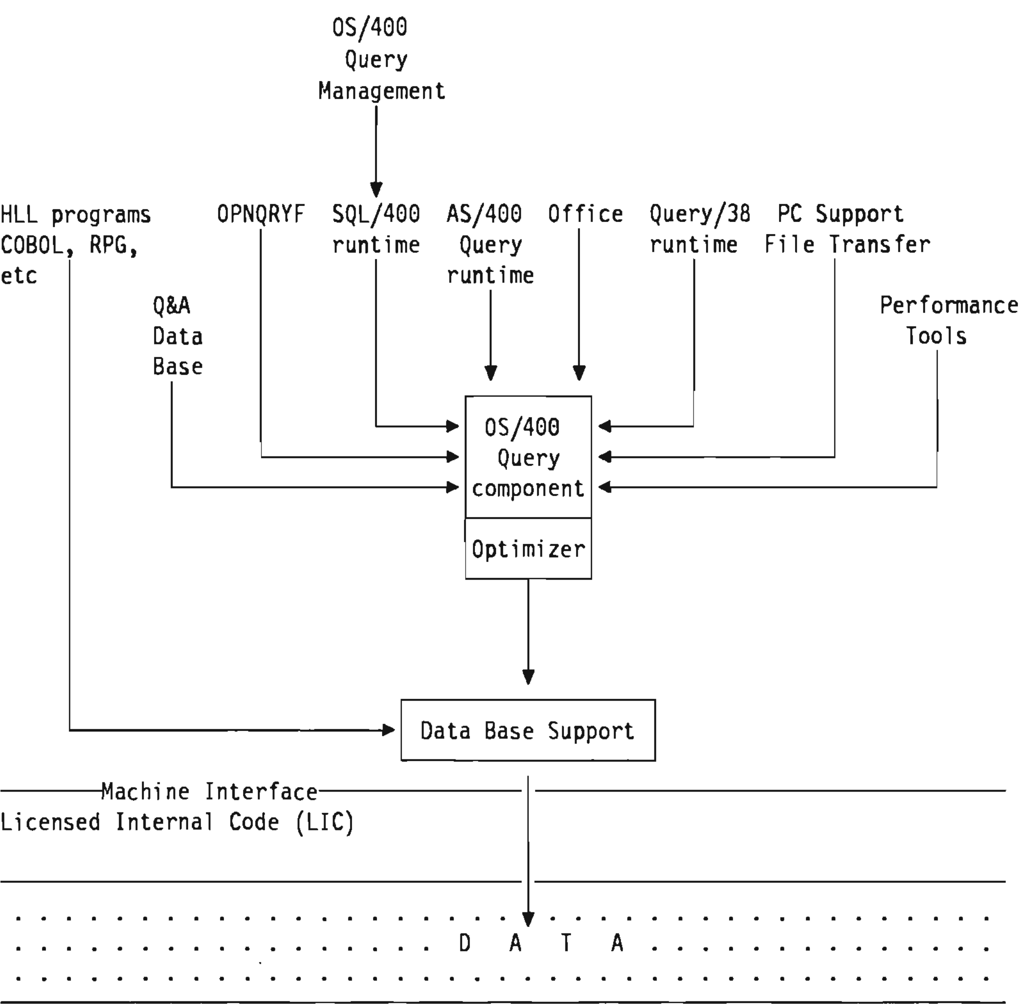


Figure 5-1. Methods of Accessing AS/400 Data

The Optimizer: The Optimizer is a major part of the OS/400 Query component. It makes decisions to improve performance of a query. In some queries, the Optimizer may decide to build a temporary index over the data. When this happens, the time required for the index build will delay the SELECT statement processing. The database contains information such as which indexes are already built on a table. If necessary the Optimizer uses these indexes to assist performance. The main objective of SQL performance analysis is the correct use of indexes. The programmer is responsible for creating indexes that the Optimizer will use. See "Indexes" on page 5-11 for more information on correct index usage.

When compiling, the SQL precompiler builds a *Query Definition Template* (QDT) for embedded SQL programs. The QDT is comprised of two main parts: an internal representation of the SQL statement, and the access plan. This QDT is validated and optimized by the OS/400 Query component. The QDT is stored with the program object and used at execution time.

A more detailed discussion on the Optimizer, Access Plans and the QDT appears later in this chapter. See "The Optimizer" on page 5-44 for more information.

Design Guidelines

In order to achieve satisfactory system performance using SQL, it is important to look carefully at:

- Database design
- Application design
- Program design.

These topics will be discussed in the following sections of this chapter. This information will consider the performance aspects for SQL **only** and although applicable to other database functions, will not go into *general* database, application, and program design techniques.

Introduction

Performance of SQL in application programs is important to ALL system users, because inefficient usage of SQL can considerably waist system resources (CPU and I/O). You will often need to decide whether, for database functions, SQL or HLL I/O statements in a user program are more appropriate, which will depend upon the nature of the application.

Note that the information provided here is in addition to that provided in Chapter 15 of the *Programming: Structured Query Language Programmer's Guide (SC21-9609).* This provides several other important considerations for performance with SQL, and should be referred to in conjunction with the concepts provided here. In addition, you may refer to chapter 9 describing *Opening a Database File*, and especially the section entitled *Performance Considerations* of the *AS/400 Database Guide (SC21-9659)* for more performance hints on OPNQRYF. OPNQRYF uses the OS/400 Query component, and therefore many of the hints in this chapter of the Database Guide can also be applied to SQL.

In the following sections of this chapter, you will often read about the *Optimizer.* This performance optimizing facility is discussed in detail in "The Optimizer" on page 5-44.

General Considerations

The main goal in using SQL is to get the **correct result** for your database request and then, secondly, get that result in a reasonable time frame.

Before you start designing for performance you should think about the following considerations, according to this **checklist**:

1. When to consider performance:
   - Over 10,000 rows - Performance impact: *"noticeable"*
   - Over 100,000 rows - Performance impact: *"**concern**"*
   - With complex queries used repetitively
   - Using multiple work stations with high transaction rates.
2. What resources to optimize:
   - I/O Usage

- CPU usage
- Effective usage of indexes
- OPEN/CLOSE processing
- Concurrency (COMMIT)

3. How to design for performance:
   - Database design
     - Table structure
     - Indexes
     - Table data management
     - Journal management
   - Application design
     - Structure of programs involved
   - Program design
     - Coding practices
     - Performance monitoring.

These topics are addressed in this chapter. The relevant tips and techniques for SQL performance improvements are underlined in the following sections for better reference and summary.

Database Design

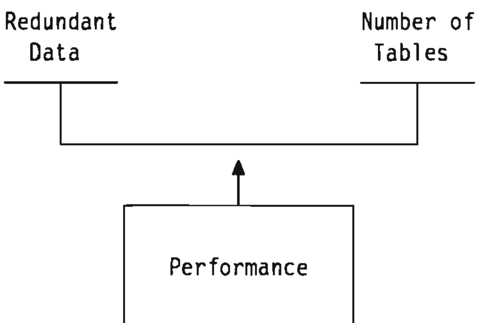
One of the first things you want to look at is determining what tables you require in your database and the relationship between those tables. This leads you to the considerations discussed below.

Normalization

One of the most critical design decisions that can affect performance is the number of tables that may have to be "joined" in an SQL statement to satisfy application requirements. The trade-off is between SQL performance and the anomalies that result in tables which have not been designed to be in their fully normalized form.

Numerous design methods are available that enable you to design "technically correct" databases. Most of these can be used to design good relational database structure. One important part of these techniques is to eliminate the storing of redundant or duplicate data. The main objective is therefore, to avoid problems of updating such redundant data.

If this design approach of normalization, for instance 3rd Normal Form (3NF), or more is taken to its ultimate conclusion, the result is a large number of tables, many of which will be involved in "join" operations. This can lead to poor OS/400 database performance. You should take care to design tables that will not create problems caused by redundant data, while at the same time not causing significant performance problems. Therefore you have to find a good balance between:



For instance, try to minimize the use of "code" tables where very little is gained from their use. For example, suppose an employee table contains a JOBCODE column, with data values 054, 057, and so on, which must be joined with another table to "translate" the codes to "Programmer", "Engineer", and so on. The cost of this join could be quite high compared to the savings in storage and potential update anomalies resulting from redundant data.

For example:

- **Normalized** data form

EMPLOYEE Table

Employee No	Jobcode
000010	057
000020	054
000030	057
...	...

JOBCODE Table

Jobcode	Job Title
054	Programmer
057	Engineer
...	...
...	...

- **Redundant** data form

EMPLOYEE Table

Employee No	Job Title
000010	Engineer
000020	Programmer
000030	Engineer
...	...

Note that the set level (or mass operation) nature of SQL greatly lessens the "danger" of a certain redundant data form. For example, the ability to update a set of rows with a single SQL statement greatly reduces this risk. In the following example, the job title ENGINEER must be changed to TECHNICIAN for *all* rows which match this condition.

SQL can easily be used to update JOBTITLE:

```
UPDATE EMPLOYEE
  SET JOBTITLE = 'TECHNICIAN'
  WHERE JOBTITLE = 'ENGINEER'
```

Table Size

The size of the table(s) your application program is accessing can have a significant impact on the performance of the application program. Consider the following:

- Large Row Length

  If a table accessed sequentially has a large row length because it has many columns (say 100 columns), in some cases you could achieve better performance by splitting it into two or more tables. This assumes that your application is not accessing all columns. The main reason for the better performance is that I/O may be reduced because you will get more rows per page. Splitting the table will affect applications that access all columns because they will incur the overhead of joining the tables back together again. You must decide where to split the table, based on the nature of the application and frequency of access to various columns.

- Large Number of Rows

  If a table has a large number of rows, it is crucial for good performance that your SQL statements are constructed in such a way that the Optimizer will use an index in accessing the table. The use of indexes is very important for achieving the best possible performance.

Indexes

Indexes (or access paths) are probably the most important facility that you have in controlling and tuning the performance of your applications, because the Optimizer uses them for performance optimization. They are created in five different ways:

1. CREATE INDEX (in SQL)
2. CRTPF, with key
3. CRTLF, with key
4. CRTLF, as join logical file
5. CRTLF, with select/omit specifications, without a key, and without dynamic selection (DYNSLT). Before OS/400 Release 3 the Optimizer used to ignore such an access path.

Indexes are used to enable row selection via an index versus table scanning, which is usually slower. Table scanning sequentially processes all rows in a table. Prior to Release 3, indexes were required for cursors to be left open on repetitive execution of SQL statements. If a permanent index is available, building a temporary index can be avoided. Indexes are required for:

- Join tables
- ORDER BY
- GROUP BY

and will be built, if no permanent index exists.

The number of indexes however has to be managed because of the extra system cost of maintaining the indexes during update types of operations. Below are index **"rules of thumb"** for particular types of tables:

1. Primarily read-only tables:

   Create indexes over columns as needed.

   Only consider creating an index if such a table is greater than approximately 1000 rows or is going to be used with ORDER BY, GROUP BY, or join processing. Index maintenance could be more costly than occasionally scanning the entire table.

2. Primarily read-only tables, with low update rate:

   Create indexes over columns as needed.

   Avoid building indexes over columns which are updated frequently. INSERT, UPDATE, and DELETE will cause maintenance to all indexes related to the table.

3. High update rate tables (INSERT, UPDATE or DELETE):

   Avoid creating many indexes. An example of a table which has a high update rate (INSERT) is a logging or a history table.

Implicit sharing of SQL indexes:

Implicit access path sharing is only important to users running OS/400 Releases prior to Release 3. Since Release 3, access paths will only be shared if they are identical.

OS/400 supports a feature called access path sharing. When the access path for an SQL index is shared, then one or more SQL indexes may be sharing the same actual "index object". OS/400 will attempt to reduce the number of redundant access paths maintained over a data space. An access path is redundant if there is an access path which has the same keys, in the same order, with possible additional key fields where duplicate key ordering is not specified.

An example:

```
CREATE INDEX INDABC ON TABLEXY (A ASC, B ASC, C ASC)

CREATE INDEX INDAB  ON TABLEXY (A ASC, B ASC)
```

The access path for INDAB is redundant because of the access path of INDABC. Using index INDABC for ordering will order the rows in A and B sequence just like index INDAB. The only difference is the ordering of rows which have the same values for A and B.

Since SQL indexes do not specify a duplicate key ordering the access paths for SQL indexes could be shared (prior to Release 3). When an access path is shared, then only one access path is created. It is shared by both indexes. The

owner of the access path is the logical file that originally created the access path.

You can see if an access path is shared via the output of the Display File Description (DSPFD) CL command. In the above example, a DSPFD of INDAB will indicate that the access path is shared and the owner is INDABC.

Since Release 3, when you create an index like INDAB above, you can be certain that the index contains only the fields A and B. This index would be chosen by the Optimizer to process the following statement:

```
UPDATE TABLEXY
  SET C = 'NEW'
  WHERE A = 12
  AND   B = 'CARO'
```

Prior to Release 3, an index would **not** have been used, because the only index was INDABC, and this one was not eligible for use, because there is a column (column C) included which is updated in this statement. Indexes cannot be used for update if one of the key fields is updated. Index INDAB would not be used, as data is really accessed via INDABC.

In order to see what indexes are available to you and what attributes are used, run a query over SYSINDEXES and SYSKEYS catalog tables. For an example of this, see "Determining Indexes" on page 5-53.

For a further discussion on indexes, see "Indexes" on page 3-5.

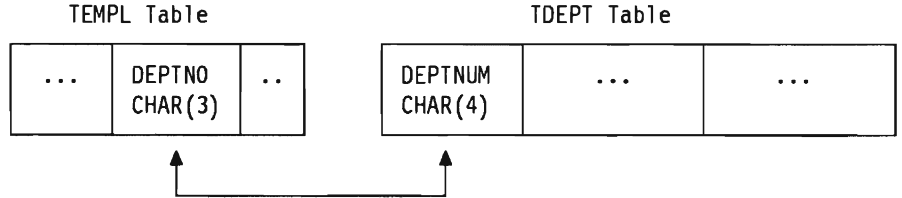
Matching Attributes of Join Fields

Columns in tables which are joined should have identical attributes, that is, the same column length, same data type (character, numeric, and so on). Non-identical attributes may result in temporary indexes being built, even though indexes over corresponding columns may exist already.

In the following example,

"join" will build a temporary index, and ignore an existing one:

```
SELECT EMPNO, LASTNAME, DEPTNAME
  FROM TEMPL, TDEPT
  WHERE TEMPL.DEPTNO = TDEPT.DEPTNUM
```



Database File Management

Use the following techniques to minimize the number of deleted rows in a table. This will provide better query performance.

1. Use Reorganize Physical File Member (RGZPFM) for tables:

   (but remember: RGZPFM can take hours if the table is large, and the table must be available for exclusive use)

   - This compresses rows which are marked for deletion. A table scan will now be faster, because it will not have to process any rows marked for deletion.
   - This process can be used to change the physical sequence of rows in the table to match the most frequently used index. When selecting based on the index sequence, the Optimizer will recognize that the data is physically in the sequence it requires. Therefore the pages will be pre-fetched from auxiliary storage into single level storage. This provides substantial performance enhancements, see "Data Management Methods" on page 5-31.

2. Avoid multiple rows being marked as deleted by using the CL command:

   ```
   CLRPFM  Lib/Table
   ```

   instead of the SQL statement:

   ```
   DELETE  FROM Lib/Table
   ```

   The benefit of the CL command CLRPFM is that it will result in an empty file where as the result of the SQL command DELETE will be a table with many rows marked as deleted.

   You have to decide about the following trade-offs:

   - With CLRPFM, commitment control is not possible. Moreover CLRPFM command requires exclusive update right to the table. Note that CLRPFM command is not an SAA database function.
   - SQL DELETE only marks rows as deleted and the table later should be reorganized, as mentioned above
   - CLRPFM is faster than SQL DELETE.

Journal Management

Place journal receivers in User Auxiliary Storage Pools (ASP) which are separated and not checksummed. With the following restrictions, allocate a User ASP:

- With only 1 disk unit (actuator)
- For journal receivers attached to the same journal
- So the journal receiver is the only active object contending for use of the disk actuator, so journal entries can be sequentially written to disk.

For further considerations see *Programming: Backup and Recovery Guide, SC21-8079.*

Application Design

What we have just discussed are general concepts that can be used for good database design and management. This topic discusses an important application structure technique that can be used to considerably improve SQL performance.

There is one main improvement regarding SQL performance in Release 3 which can be noticed if application design is done according to the following guidelines. This improvement was available prior to Release 3 with PTF SF04173 (now superseded by PTF SF04941). This improvement allows an ODP to be reused.

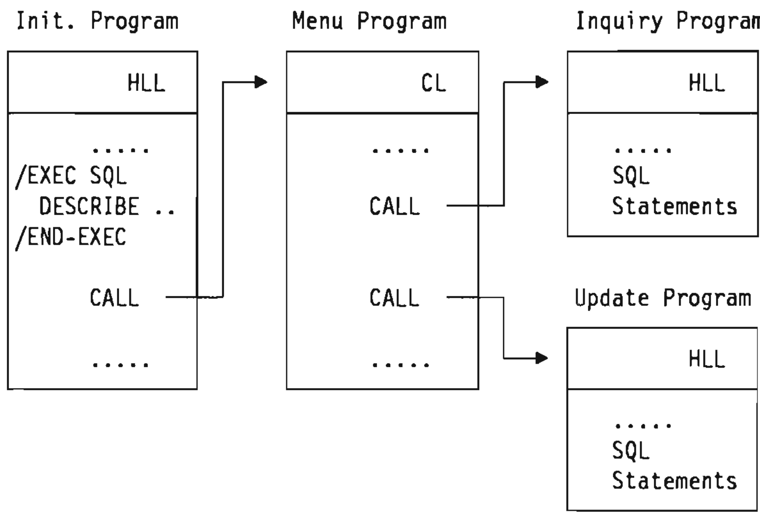
Reusable ODP's Across Invocations

In Release 1.2 and Release 2, without PTF, repetitive opens and closes could be avoided for static SQL I/O statements if the program containing the SQL statements remained active (on the invocation stack). When the program ended (left the invocation stack), all open files were closed. For this reason, you were advised to keep all SQL statements used by an application within one program.

Since Release 3, the files that are open for an SQL program, when that program leaves the invocation stack, will not be closed. This happens if there is at least one other program remaining on the invocation stack that has previously issued an executable SQL statement. This will enable the user to separate SQL statements into different programs and call them repetitively as long as the above condition is true.

Even though an ODP remains open across multiple invocations of a program, the declared cursor must be treated as if it is closed in each invocation. That is, the program must issue an SQL OPEN statement for that cursor before using it again and an SQL CLOSE before the program ends. Essentially, you code your program as if the ODP is not reused.

Here is an example of an application flow. It shows an *always-active* HLL (High Level Language) initialization- or dummy-program which executes an SQL statement before the menu CL-program is called. Note that SQL statements cannot be executed in CLP's. The initialization program, would not be necessary if the menu program was written in a HLL, because the dummy SQL statement could be embedded into that HLL program.



A good candidate for the SQL statement in the initialization program is DESCRIBE, even in a static SQL environment. If the host variable used is invalid, SQL runtime will set a negative SQL return code and complete in a very short time, such that the impact is negligible. The statement could look like this:

```
DESCRIBE DUMMY INTO :DUMMY
```

Note 1: Host variable :DUMMY could be declared in the HLL program as a one-byte character field.

Note 2: Ignore error message SQL0804 "SQLDA not valid" generated by this dummy DESCRIBE.

Note 3: This approach of putting an SQL statement into an always-active initialization program in front of the menu program would leave ODP's open in case of disconnect job (DSCJOB). Commands which operate on files which are issued from a different job could fail because the SQL job is keeping the files open. Reclaim Resources (RCLRSC) can be used to close all files.

Note 4: If the initialization program is a CSP program, avoid using CSP XFER and DXFER operations within the program.

The CSP XFER and DXFER operations will transfer control to the specified program and will remove, from the invocation stack, the program that invoked the transfer operations. If the program that is removed is the initialization program, then all of the files will be closed.

Use the CSP CALL operation instead of XFER and DXFER in this case.

Program Design

In order to achieve the best performance with either programs containing embedded SQL statements or with your Interactive SQL session, it is very important that you code SQL with an understanding of methods explained in this topic. It will go into some detailed SQL program design techniques and provide you with some tips to improve SQL performance.

Optimizing CPU Usage - Avoid Dynamic SQL

You should use dynamic SQL only when necessary. Most often, when an application program is being designed, you will know the functions it must perform. Consequently, you will know how to construct your embedded SQL statements. These will be **static** SQL statements. In other cases, you may want to postpone the construction of SQL statements until the execution of the application. This can be done by using **dynamic** SQL statements.

If you use dynamic SQL statements, you should be aware that tasks normally done at precompile time will have to be done at *execution time*. This will have a significant negative effect on the performance of your application. The tasks that will have to be performed are:

1. Read the SQL statement constructed by the user
2. Verify that the statement can be executed dynamically
3. Validate syntax, tables, host variables
4. Prepare the statement for execution
5. Optimize access method
6. Check for proper authorization
7. Execute the statement.

If the same statement was coded as a static SQL statement, only authorization checking and statement execution would be done at program execution time, that is, only items six and seven of the above list would be performed. The other tasks would be done at precompile time.

As a general guideline, you should avoid the use of dynamic SQL statements in your application programs as much as possible. The main reason is the overhead needed to prepare the statement.

In addition, more CPU and I/O resources will be used because of the following:

- Programs containing EXECUTE statements will open all cursors with update and delete intent, even though there are no UPDATE or DELETE statements in the program. This would effect CPU and I/O resources and do unnecessary locking.
- Re-PREPARE of a statement will cause any related ODP's which SQL/400 has kept open for possible reuse, to be closed.

Optimizing Index Usage

You should use redundant predicates in a join, if possible, because this allows the Optimizer to select the best join order, depending on the estimated number of returned rows. Refer to "Key range estimates" in section "Cost Estimation" on page 5-45. If you are using two tables and at the same time qualifying your search condition with a specific value, you should provide redundant search information to SQL. This will aid the Optimizer to choose the best way to do the join. The following example selects names, phone numbers and department name for all employees working in department 'E11':

```
SELECT  LASTNAME,  PHONENO, DEPTNAME
  FROM   TEMP, TDEPT
  WHERE  TEMP/WORKDEPT = TDEPT/DEPTNO
  AND    TEMP/WORKDEPT = 'E11'

.............

SELECT  LASTNAME,  PHONENO, DEPTNAME
  FROM   TEMP, TDEPT
  WHERE  TEMP/WORKDEPT = TDEPT/DEPTNO
  AND    TEMP/WORKDEPT = 'E11'
  AND    TDEPT/DEPTNO  = 'E11'          Redundant predicate [2]
```

If you do not duplicate the search condition as shown, then you should put the search condition on the table which will select the fewest rows for the predicate.

If the two tables being joined have a one to many relationship, put the search conditions on the table with unique rows.

The example above assumes that TDEPT/DEPTNO is unique and therefore the Optimizer will choose a join order as TDEPT primary and TEMP as secondary join table. For such a case you should therefore build an index over the secondary table (TEMP).

See also "Join Optimization" on page 5-48 for more details about this topic.

Minimize the Number of SQL Statements

- Do not do more than your program really needs. That is, try to keep the number of SQL statements to a minimum. If you can loop on an SQL statement, do this, rather than coding the statement several times. The loop would be much faster because the ODP can be reused. Without looping, each statement has a separate ODP. See "Reusability of ODP's" on page 5-38.
- For read-only operations, and if the row to be retrieved is unique, use the SELECT INTO operation.
- For updates or deletes where the row does not need to be inspected by the program, use UPDATE or DELETE without a cursor.

[2] This comment is shown reversed in order not to confuse a comment on a SQL statement line, which is not supported in SQL/400.

- When rows must be inspected by the program, use UPDATE or DELETE with CURRENT OF cursor, as in the next section.
- Use subqueries or joins instead of multiple cursors.

Updating via Cursor Operation

Though the previous section stated that you should keep the number of SQL statements to a minimum (that is, use UPDATE or DELETE without a cursor), there is a pitfall, because the cursor operation which has more statements may still be faster.

In the following example we want to read a row and update that row with input from the user. This example *appears* to use fewer SQL statements. However, it requires two ODP's.

```
------  read  SCRENNO  from display screen

SELECT name, salary
  INTO :SCRENNAM, :SCRENSAL
    FROM SALARYTAB
    WHERE EMPLOYNO = :SCRENNO

------  prompt user with name and salary and read new salary  -----

UPDATE SALARYTAB
  SET salary = :NEWSALARY
  WHERE EMPLOYNO = :SCRENNO
```

Therefore, instead of this example, you should use a cursor operation. In the code above there are two problems:

1. Poor performance, because when SQL and OS/400 database management execute these two statements, they fetch the row from the table with read only intent and when they process the update statement they have to go back to the database and re-fetch with update intent and update the row. That will take three operations to the database.
2. This coding exposes you to serious database integrity problems. That is, when the row is selected, it will not be locked (unless you use COMMIT *ALL) and you may not be updating data which may have been changed by another job.

Instead of the previous example, you should DECLARE a cursor and OPEN, FETCH along with UPDATE, WHERE CURRENT OF cursor. This requires only one ODP.

Therefore **use the cursor operation**:

```
DECLARE C1 CURSOR FOR
  SELECT name, salary
    FROM SALARYTAB
    WHERE EMPLOYNO = :SCRENNO

------  read  SCRENNO  from display screen

OPEN C1

FETCH C1 INTO :SCRENNAM, :SCRENSAL

------  prompt user with name and salary and read new salary

UPDATE SALARYTAB
  SET salary = :NEWSALARY
  WHERE CURRENT OF C1

CLOSE C1
```

The UPDATE ... WHERE CURRENT OF cursor is much more efficient, because the row is available and it can be directly updated. There is also the advantage that the row is fetched with update intent, so the row will be locked.

Partial Update Capable Join with Subqueries

Prior to Release 3 you may have coded a primary table update like this:

```
DECLARE C1 CURSOR FOR
  SELECT TABLE1.COL1
    FROM TABLE1, TABLE2
    WHERE TABLE1.COL1 = TABLE2.COL1

OPEN C1

--- loop on ---
FETCH C1 INTO :HCOL1

UPDATE TABLE1
  SET COL2 = 'new value'
  WHERE COL1 = :HCOL1
--- end loop ---

CLOSE C1
```

Since Release 3 you can code this using subquery:

```
UPDATE TABLE1
  SET COL2 = 'new value'
  WHERE EXISTS (SELECT *
                FROM TABLE2
                WHERE TABLE1.COL1 = TABLE2.COL1)
```

The subquery is the better choice not only because the number of statements can be reduced, but also because it will require only one ODP instead of two.

This technique can be used any time you need to do updates on the primary table of a join and you don't need column values from the secondary table.

Include Selection Columns in ORDER BY and GROUP BY

In some situations you can improve performance of a SELECT that uses an ORDER BY or GROUP BY when selecting a *specific* column value or group.

In the following examples, note that the second SELECT in each case would be better, because they have the selection column as the first key field. This will allow *Key Row Positioning* to be done instead of just Key Row Selection. These are data access methods and are explained in "Row Selection Options" on page 5-32. Therefore instead of:

```
SELECT LASTNAME, EDUCLVL
  FROM TEMPL
  WHERE EDUCLVL = 16
  ORDER BY LASTNAME
```

add first key field:

```
SELECT LASTNAME, EDUCLVL
  FROM TEMPL
  WHERE EDUCLVL = 16
  ORDER BY EDUCLVL, LASTNAME
```

or, instead of

```
SELECT LASTNAME, AVG(SALARY)
  FROM TEMPL
  WHERE EDUCLVL = 16
  GROUP BY LASTNAME
```

add first key field:

```
SELECT LASTNAME, AVG(SALARY)
  FROM TEMPL
  WHERE EDUCLVL = 16
  GROUP BY EDUCLVL, LASTNAME
```

Assuming there is no permanent index available, a temporary index will be built, and in the first example of each of the above two cases Key Row Selection will occur. In the second example Key Row Positioning will be done, which is usually much faster.

OR and IN Predicates

SQL/400 may **not** use an index if the key field is referenced by OR conditions or IN parameters. Here are a number of examples using IN predicates. Remember that IN is the same as multiple OR predicates with an equal condition for the same column.

You can force the Optimizer to use an index, if you:

1. Specify ORDER BY, because OS/400 database can do Key Row Positioning on multiple ranges. However the Optimizer is not always able to recognize where this can be applied. In this example the ORDER BY clause forces the Optimizer to choose an index, and once it has done so, it will choose the multirange type Key Row Positioning. See "Key Row Positioning" on page 5-33.

```
SELECT LASTNAME, WORKDEPT
  FROM TEMPL
  WHERE WORKDEPT IN ('D11','D12')        Index NOT used

............

SELECT LASTNAME, WORKDEPT
  FROM TEMPL
  WHERE WORKDEPT IN ('D11','D12')
  ORDER BY WORKDEPT                      Index used
```

---

2. Change the predicate to use LIKE (if the values fit), because, if the leading characters of the value that you select in the IN predicate match a pattern, then the LIKE predicate is a good alternative to the IN predicate.

```
SELECT LASTNAME, WORKDEPT
  FROM TEMPL
  WHERE WORKDEPT IN
        ('D11','D12','D13','D14','D15','D16','D17','D18','D19')
                                         Index NOT used

............

SELECT LASTNAME, WORKDEPT
  FROM TEMPL
  WHERE WORKDEPT LIKE 'D1%'              Index used
```

---

3. Change the predicate to use BETWEEN If the values in the IN clause define all the values in a range, the BETWEEN predicate is a good alternative to the IN predicate.

```
SELECT LASTNAME, WORKDEPT
  FROM TEMPL
  WHERE WORKDEPT IN
        ('D11','D12','D13','D14','D15','D16','D17','D18','D19')
                                          Index NOT used

............

SELECT LASTNAME, WORKDEPT
  FROM TEMPL
  WHERE WORKDEPT BETWEEN 'D11' AND 'D19'
                                          Index used
```

Index Usage with the LIKE Predicate

An INDEX will **not** be used when the string is in the LIKE predicate:

- Is a host variable,
- Starts with a "wild card", that is:
  - Starts with a "%" character
  - Starts with a "_" character.

Specify a BETWEEN clause on keys whenever possible

Specifying a BETWEEN clause on key fields of a WHERE clause can significantly reduce the amount of data read by the system and may reduce the execution time.

```
SELECT WORKDEPT, LASTNAME
  FROM TEMPL
  WHERE WORKDEPT >= 'D11'
  ORDER BY WORKDEPT
```

If you know the desired high bound value, specifying a BETWEEN clause will be more efficient.

```
SELECT WORKDEPT, LASTNAME
  FROM TEMPL
  WHERE WORKDEPT BETWEEN 'D11' AND 'D19'
  ORDER BY WORKDEPT
```

Join Optimization

The following is a list of join performance tips to help you specify more efficient join operations.

1. Specify join predicates on the WHERE clause to avoid a cartesian product operation.

   ```
   SELECT ...
   FROM      TABLE1, TABLE2, TABLE3
   WHERE     TABLE1.FIELDX = TABLE2.FIELDX AND
             TABLE2.FIELDX = TABLE3.FIELDX
   ```

   In the above example there are two join predicates specified in the WHERE clause.

   Each secondary file should have at least one join predicate in the WHERE clause that references one of its fields as a 'join-to' field.

In the above example the secondary files, TABLE2 and TABLE3, both have join predicates that reference FIELDX as a 'join-to' field.

2. Create an index over each secondary file

   An index is required over each 'join-to' field of the WHERE clause. If one does not exist, then it will be built during the execution of the select statement, which could take a considerable amount of time.

   Since a join in SQL is always an inner join, the Optimizer may decide to switch the order of the files specified on the join operation to a more optimized join order.

   To ensure that existing indexes will be used for the join operation, create an index over the 'join-from' field and another index over the 'join-to' field of each join predicate. This will cover the case where the Optimizer could switch the join order.

```
SELECT .....
FROM     TEMP, TDEPT
WHERE    TEMP.DEPT = TDEPT.DEPTNO
     AND TEMP.WORKBLDG = '015'
```

   The join predicate is TEMP.DEPT = TDEPT.DEPTNO.

   In the above example, the Optimizer could perform the join in one of two ways:

   a. The Optimizer could perform the join from file TEMP to file TDEPT in which case an index on TDEPT.DEPTNO, the 'join-to' field, is required.

   b. The Optimizer could perform the join from file TDEPT to file TEMP in which case an index on TEMP.DEPT, the 'join-to' field, is required.

   Creating indexes on both possible 'join-to' fields, TEMP.DEPT and TDEPT.DEPTNO, covers the case where the Optimizer could change the order of joining the files.

3. On the WHERE clause, specify as many record selection conditions on each file as possible.

4. Specify redundant WHERE predicates in a join, if possible.

```
SELECT .....
FROM     TEMP, TDEPT
WHERE    TEMP.WORKDEPT = TDEPT.DEPTNO
     AND TEMP.WORKDEPT = 'E11'
.....

SELECT .....
FROM     TEMP, TDEPT
WHERE    TEMP.WORKDEPT = TDEPT.DEPTNO
     AND TEMP.WORKDEPT = 'E11'
     AND TDEPT.DEPTNO  = 'E11'     Redundant predicate
```

5. Make the primary file of the join the file with the fewest number of selected records.

   If you specify an ORDER BY containing fields from one file only, that file becomes the primary file.

   If an ORDER BY references fields from one file and that file has a larger number of selected records than its secondary file, consider performance tip 7 - 'Force a temporary result to get faster join performance', below.

You can also force the Optimizer to use a file containing the smallest number of selected records as the primary file by specifying an ORDER BY clause that references one of the file's fields.

6. Attempt to join the files from smallest to largest, depending on the estimated number of records selected from each file.

   The system will process a join of two files with different numbers of selected records most efficiently when the smaller file is joined with the larger file.

   For the following example, assume that all of the 4 files contain the same common field named DEPT.

```
TAB1 has   100 selected records
TAB2 has  1000 selected records
TAB3 has  5000 selected records
TAB4 has 10000 selected records

This WHERE clause is more efficient:
  WHERE     TAB1.DEPT = TAB2.DEPT AND
            TAB1.DEPT = TAB3.DEPT AND
            TAB1.DEPT = TAB4.DEPT

This WHERE clause is not as efficient:
  WHERE     TAB1.DEPT = TAB2.DEPT AND
            TAB2.DEPT = TAB3.DEPT AND
            TAB3.DEPT = TAB4.DEPT
```

   The join predicates specified in the WHERE clause can affect the performance of the join. Always specify the join from the file with the smallest number of selected records to the file with the largest number of selected records if possible.

7. Forcing a temporary result file to be used could result in faster join performance.

   If **all** of the following apply,

   - Fields from only one file are specified on the ORDER BY and that file has a larger number of selected records than the other files.
   - A file with a larger number of selected records is being joined to a file with a smaller number of selected records
   - A small number of records are returned from the join operation.

   then add a column from another file to the ORDER BY clause to force the use of a temporary file.

   This type of ORDER BY will cause a temporary result file to be used to hold the join result, and more importantly will allow the Optimizer to join the files in the most efficient join order.

   For the following SELECT, the ORDER BY forces TAB2 as the primary file which is joined to TAB1. If the >= condition on TAB2.COL2A results in a larger number of records than the = condition on TAB1.COL1A, then the join will be done from the larger file (TAB2) to a smaller file (TAB1). This join order is not as efficient as joining from TAB1 to TAB2.

```
SELECT   TAB1.COL1A, TAB2.COL2A
FROM     TAB1, TAB2
WHERE    TAB1.COL1X = TAB2.COL2X AND
         TAB1.COL1A = 99 AND
         TAB2.COL2A >= 10
ORDER BY TAB2.COL2A
```

Adding a column from the the other file to the ORDER BY clause will force a temporary result file to be used to process the join and will allow the Optimizer to join the files in the most efficient join order, TAB1 to TAB2.

```
SELECT   TAB1.COL1A, TAB2.COL2A
FROM     TAB1, TAB2
WHERE    TAB1.COL1X = TAB2.COL2X AND
         TAB1.COL1A = 99 AND
         TAB2.COL2A >= 10
ORDER BY TAB2.COL2A, TAB1.COL1A
```

Note: this technique should only be used when a few number of records are returned.

Avoid Numeric Conversion

If the table your application program is accessing contains numeric columns, you should avoid numeric conversions. As a general guideline, you should always use the same data type for columns, literals, and host variables used in a comparison. If the data type of the literal or the host variable has greater precision than the data type of the column, the Optimizer will **not** use an index created on that column. To avoid problems, for columns, literals, and variables being compared, use the:

- Same data type
- Same scale, if applicable
- Same precision, if applicable.

In the following example the data type for the EDUCLVL column is INTEGER. If we assume that an index has been created on that column, then the Optimizer will not use this index in the first SELECT. This is because the precision of the literal is greater then the precision of the column. In the second SELECT, the Optimizer will consider using the index, because the precisions are equal.

Example where EDUCLVL is INTEGER:

```
SELECT LASTNAME, EDUCLVL
  FROM TEMPL
  WHERE EDUCLVL > 16.3                Index NOT used

............

SELECT LASTNAME, EDUCLVL
  FROM TEMPL
  WHERE EDUCLVL >= 17                 Index used
```

Avoid String Truncation

In general, when literals or host variables are compared to character columns, you should use the same length for the literals or host variables as the one specified for the column:

- If the literals or host variables are **longer** than the column length, the Optimizer will **not** use an index created on that column.
- If the literals or host variables are **shorter** than the column length, the Optimizer will *pad* the literals or host variables with *blanks*. In the latter case, the Optimizer will try to use any available index on the column.

In the following example, the WORKDEPT column is defined as CHAR(3). Assume that an index has been created on that column. The Optimizer will **not** use this index in the first SELECT, because the literal compared to WORKDEPT is four bytes long. In the second SELECT the literal is specified as three bytes, so the Optimizer will consider using the index.

Example where WORKDEPT is CHAR(3):

```
SELECT LASTNAME, WORKDEPT
  FROM TEMPL
  WHERE WORKDEPT = 'E11 '            Index NOT used

                        └──────── note the blank!
............

SELECT LASTNAME, WORKDEPT
  FROM TEMPL
  WHERE WORKDEPT = 'E11'             Index used
```

Avoid Arithmetic Expressions

You should never have an arithmetic expression as an operand to be compared to a column in a WHERE clause. The Optimizer will **not** use an index on a column that is being compared to an arithmetic expression. If a host variable must take part in a calculation, this calculation should be done **outside** the SQL statement.

The following example assumes that an index has been created on the SALARY column. The first SELECT will **not** use this index, because an expression is being compared to SALARY. In the second SELECT, the Optimizer will consider using the index.

Example of a host variable calculation:

```
SELECT LASTNAME, SALARY
  FROM TEMPL
  WHERE SALARY < :SAL + 700          Index NOT used

............

TEMP = SAL + 700            /* Native language statement */

SELECT LASTNAME, SALARY
  FROM TEMPL
  WHERE SALARY < :TEMP               Index used
```

Index usage with UPDATE

You should use care in updating fields which are (part of) an index.

The Optimizer will **not** use an index that has a key field which can potentially be updated. The reason for this rule is that you do not want to build a query that could end up being a never ending query. Therefore:

1. An index is **not** used when the index field appears in a SET clause or in the FOR UPDATE OF clause.
2. An index is **never** used for cursor UPDATE if no FOR UPDATE clause is specified on DECLARE CURSOR statement.
3. Be aware of implicitly shared indexes, which were created prior to Release 3. A simple index may actually be owned by a composite index which includes a column being updated.

In the following three examples, an index is **not** used, because the statements could be candidates for this problem:

1. If you update an indexed field:

   ```
   CREATE INDEX X1 ON TEMPL (SALARY)

   UPDATE TEMPL
     SET SALARY = SALARY + 1000
     WHERE SALARY < 10000
   ```

2. If there is no FOR UPDATE OF clause:

   ```
   CREATE INDEX X1 ON TEMPL (DEPT, NAME)

   DECLARE C1 CURSOR FOR
     SELECT NAME
     FROM TEMPL
     WHERE DEPT = 'B01'

   OPEN C1

   FETCH C1  INTO :HNAME

   UPDATE TEMPL
     SET SALARY = SALARY + 10
     WHERE  CURRENT OF C1

   CLOSE C1
   ```

   Index X1 is **not** used in this example, because there is an implied FOR UPDATE OF clause, which names *all* of the columns in table TEMPL for update.

3. If an update capable column is part of a composite index, an index is **not** used:

   ```
   CREATE INDEX X1 ON TEMPL (DEPT, NAME, SALARY)

   UPDATE TEMPL
     SET SALARY = SALARY + 10
     WHERE  DEPT = 'B01'
   ```

Optimizing Concurrency

In order to limit the number of record locks you should choose the lowest commitment level applicable to your application. This will reduce commitment control processing and lock wait times, and therefore will improve the performance not only of your job but of all users. The lowest level is *NONE, the highest is *ALL.

***ALL** All rows which are updated, deleted or inserted are locked until COMMIT or ROLLBACK. Rows you read, can still be read by other users.

***CHG** Rows currently positioned on, with an update capable cursor, and rows updated, deleted or inserted are locked, until COMMIT or ROLLBACK.

***NONE** Rows currently positioned on, with an update capable cursor, are locked.

You should limit the number of active record locks by running frequent COMMIT HOLD or COMMIT statements.

For more information on concurrency see "SQL Commitment Control" on page 7-1.

Use COMMIT HOLD

The HOLD parameter on the COMMIT statement is a very useful feature, because it commits the transaction, but leaves all cursors open and leaves the positions unchanged. This is very useful in a batch program where, after for instance 3000 updates, you want to do a COMMIT, release the rows, but still want to continue processing.

By using COMMIT HOLD you do not have to put any code in your program that will reopen cursors and do any repositioning. See section "COMMIT and ROLLBACK with HOLD Option" on page 7-6.

Note: Keep in mind that HOLD is not an SAA parameter, so it cannot be ported to other systems.

Optimizing I/O with Blocking

1. Use the Override with Data Base File (OVRDBF) CL command to allow you to control blocking that will be used by OS/400 database management to implement your queries.

   - For files referenced in INSERT with VALUES clause:

     ```
     OVRDBF FILE(TEMPL) SEQONLY(*YES)

     INSERT INTO TEMPL (NAME, SALARY, WORKDEPT)
           VALUES('MET', 100000, 'B01')
     ```

   - To increase the number of rows per block:

```
OVRDBF FILE(TEMPL) SEQONLY(*YES 1000)

INSERT INTO TFILE1 (NAME, SALARY, WORKDEPT)
       SELECT NAME, SALARY, WORKDEPT
         FROM TEMPL
```

2. Avoid using commitment control, if possible:

   - Input enabled cursors are never blocked under commitment control (unless a join or GROUP BY is specified).
   - Output only cursors can be blocked only if OVRDBF SEQONLY(*YES) is specified:

```
OVRDBF FILE(TEMPL) SEQONLY(*YES 1000)

INSERT INTO TEMPL (NAME, SALARY, WORKDEPT)
       VALUES('MET', 100000, 'B01')
```

3. Select only the columns being used.

   This enables more rows in a smaller block. SELECT * (select all columns) would be very easy and quick coding, but it selects *and maps* every column out of the table, which is expensive in terms of CPU and I/O. This is extremely important for FETCH operations because they occur very frequently.

Reduce the Number of Rows Processed

1. Use column functions rather than providing logic in your program that would require extensive HLL coding. Calculating the sum of a column is an example.

2. Specify conditions in WHERE clause instead of in HAVING clause, where possible.

   This example will process all rows in TEMPL:

```
SELECT DEPTNO, MAX(SALARY)
  FROM TEMPL
  GROUP BY DEPTNO
  HAVING DEPTNO = 'B01'
```

   This example will process only rows belonging to department B01:

```
SELECT DEPTNO, MAX(SALARY)
  FROM TEMPL
  WHERE DEPTNO = 'B01'
  GROUP BY DEPTNO
```

Data Management Methods

AS/400 Data Management provides various methods to retrieve data. This topic introduces the fundamental techniques implemented in OS/400 and the Licensed Internal Code. These methods or combinations of methods are used by SQL/400 runtime support to access the data.

For complex query tasks you can find different SQL solutions that satisfy your requirements to retrieve the data from the database. This is not a *cookbook* that helps to find the best performing variation of a SQL statement. You have to understand enough about:

- Creation of the access plan
- Decisions of the Optimizer (discussed in "The Optimizer" on page 5-44)

to find it by yourself.

For this reason this chapter discusses the following topics which are fundamental instruments for data retrieval from the AS/400 database:

- Access Path
- Row Selection Options
- Reusability of ODP's.

Access Path

Definition: An access path is:

- The order in which rows in a table (or records in a database file) are organized for processing.
- The path used to locate data specified in SQL statements. An access path can be indexed, sequential, or a combination of both.

Arrival Sequence: An arrival sequence access path is the order of records as they are stored in the file. Processing files using the arrival sequence access path is similar to processing sequential or direct files on traditional systems.

Keyed Sequence: A keyed sequence access path provides access to a database file which is arranged according to the contents of key fields (indexes). The keyed sequence is the order in which rows are retrieved. The access path is automatically maintained whenever records are added to or deleted from the table, or whenever the contents of the index fields is changed. The best example of a keyed sequence access path is an SQL index.

In SQL, columns which are good candidates good candidates for creating keyed sequence access paths (SQL indexes) are:

- Those frequently referenced in predicates of SQL statements
- Those frequently referenced in GROUP BY or ORDER BY
- Those used to join tables (see "Join Optimization" on page 5-48).

For a further description of access paths, refer to the AS/400 Data Management Guide, SC21-9658.

Row Selection Options

The implementation of row selection methods is divided between the Licensed Internal Code (LIC) and the SQL/400 runtime support. LIC is code which sits below the Machine Interface. The LIC does the low level processing for example: selection, join functions, and access path creation. These low level functions actually involve reading and checking the data. Records that meet the selection criteria are passed back to the SQL/400 runtime support. (See Figure 5-1 on page 5-7 for an illustration).

The query optimization process chooses the most efficient row selection method for each SQL statement and keeps this information in the access plan. The type of access is dependent on the number of records, the number of page faults[3] and other criteria (refer to "The Optimizer" on page 5-44).

This topic discusses the possible methods the Optimizer can use to retrieve data. The general approach is to either do a data scan (defined below) or use an index. Selection can be implemented through:

- Dynamic Row Selection
- Key Row Selection
- Key Row Positioning
- Column Selection
- 'Index from Index' Selection
- File Management Row Selection.

Definition of terms used in the following section:

- The internal object that contains the data in a table is referred to as a *Data Space*.
- The first field of an index over multiple columns is referred to as the *primary* or *left-most* key.

Note: Literal values are shown in the following examples to keep them simple. However, you could just as easily code host variables instead. Remember to precede each host variable with a colon.

Dynamic Row Selection: The rows in the table are processed in no guaranteed order. They will be in arrival sequence. If you want the result in a particular sequence, you must specify an ORDER BY clause in the SQL statement. As indexes are not used in arrival sequence processing, all rows in the table are read. This operation is referred to as a *data space scan*. The selection criteria is applied to each row, and only the rows that match the criteria are returned to the application.

Dynamic Row Selection can be very efficient for the following reasons:

- It minimizes the number of page I/O's because all records in a given page are processed, and once the page has been retrieved, it will not be retrieved again.

[3] An interrupt that occurs when a program refers to a (512 byte) page that is not in main storage.

- Because it is easy for the database manager to predict the sequence of pages from the data space for retrieval, it can schedule asynchronous I/O of the pages into main storage from auxiliary storage (commonly referred to as *pre-fetching*). The idea is that the page would be available in main storage by the time the database manager needs to examine the data.

This selection method is very good when a large percentage of the rows are to be selected (greater than approximately 20%).

Dynamic Row Selection can be adversely effected when selecting rows from a table containing deleted records. As you may recall, the delete operation only marks records as deleted. For Dynamic Row Selection, the database manager is going to read all of the deleted rows, even though none will be selected. You should use the Reorganize Physical File Member (RGZPFM) CL command to eliminate deleted records.

Dynamic Row Selection is not very efficient when a small number of rows will be selected. Using Dynamic Row Selection, all rows in the table are examined leading to consumption of wasted I/O and CPU resources.

Key Row Selection: This row selection method requires keyed sequence access paths. The entire index is read and all selection criteria are applied to the index. The advantage of this method is that the data space is only accessed to retrieve rows which match the selection criteria.

Key Row Selection can be very expensive if the search condition applies to a large number of records, because the whole index will be processed, and for every matched row, a random I/O to the data space occurs.

If you specify the ORDER BY clause and none of the order columns is the left-most column in an index, the Optimizer is forced into Key Row Selection. SQL queries that do not require an index for Grouping, Ordering, or Join operations will allow the Optimizer to attempt to find an existing index for selection. If no existing index can be found, it will abandon any attempt to use keyed access to the data. The reason is that it would be faster to use Dynamic Row Selection than to build an index and then perform Key Row Selection.

If the selection criteria contains a range predicate[4] using the primary key of an existing index some additional optimization is possible, and makes the following data retrieval method more efficient than Key Row Selection.

Key Row Positioning: This access method requires a keyed sequence access path. Unlike Key Row Selection where processing starts at the beginning of the index and continues to the end, Key Row Positioning will use an index to position directly to the range of rows which match the selection criteria.

The following SQL code illustrates an example of a query where the Optimizer could choose the Key Positioning selection method:

[4] Range predicates are: =, >, > =, <, < = and BETWEEN. For example:

```
col1 = 'A'  is a range from 'A' to 'A'
col1 >= 'A' is a range from 'A' to infinity.
```

```
CREATE INDEX index1 ON sometable
       (column1 ASC)
```

```
SELECT * FROM sometable
WHERE column1 = 'C'
```

In the above example, the database support will use *Index1* to position to the first index entry with *column1* value equal to 'C'. Next, it will select all of the rows matching *column1 = 'C'* by randomly accessing the data space. Finally it will end the query when it moves beyond the key value of 'C'. Note that for this query all index entries processed and rows retrieved met the selection criteria.

You may note that once the positioning is done, processing continues similar to Key Row Selection in that random I/O is done to the data space. For that reason, this selection method is most efficient when a small percentage of rows is to be selected (less than approximately 20%).

Key Row Positioning has additional processing capabilities. One such capability is to perform range selection across more than one value. For example:

```
SELECT * FROM sometable
WHERE column1 BETWEEN 'C' AND 'D'
```

In this example, the selection will be positioned to the first index entry equal to value 'C' and then process rows until the last index entry for 'D' is processed.

An further extension of this selection method is available. We will refer to this as *multi-range* Key Row Positioning. It allows for selection of rows for multiple ranges of values for a given primary key:

```
SELECT *
FROM sometable
WHERE column1 BETWEEN 'C' AND 'D'
   OR column1 BETWEEN 'F' AND 'H'
```

In the above example, the positioning and processing technique will be used twice, once for each range of values.

Important: Key Row Positioning works only with the primary key field. The efficiency of Key Row processing depends on how selective the left-most index column is. For example:

```
CREATE INDEX yearidx
ON sometable
(year asc, month asc, day asc)
```

```
SELECT * FROM sometable
WHERE year  = 90
      month = 09
      day   = 15
```

While processing the above example, the database manager will position to the first index entry for *year = 90.* It will then process the index entries using Key Selection until all index entries for *year = 90* have completed. If there are few entries to *year = 90* then this will be very efficient. However, if there are many index entries for *year = 90* a lot of processing will have to be done to read past the entries which do not match the *month* and *day* selection criteria.

Column Selection: This selection method is very similar to Key Row Selection except that the selection column is not a key field.

For this method, processing may start at the beginning of the index and continues to the end. For each index entry, the database manager must randomly retrieve the row from the data space and apply the selection criteria.

When this method is used alone, it is rarely more efficient then Dynamic Row Selection. The Optimizer selects this method when an index is required for other reasons, such as Ordering, Grouping or Join Processing.

Index from Index: This method is not actually a selection method, but rather a preparation step which the Optimizer can put into an access plan. The database manager can build an index from a table without having to read all of the rows in the data space. The Optimizer will choose this step when:

- The query requires a keyed sequence access path because it uses Grouping, Ordering, or Join processing.
- A permanent index does not exist to satisfy the Grouping, Ordering, or Join processing requirements.
- A permanent index exists which has a selection column as the primary key field, and the primary key is very selective.

To process this, the database manager firstly uses Key Row Positioning selection on the permanent index. Secondly, selected row entries will be used to build index entries in the new temporary index. The result is an index containing entries in the required key sequence for rows which match the selection criteria.

For example:

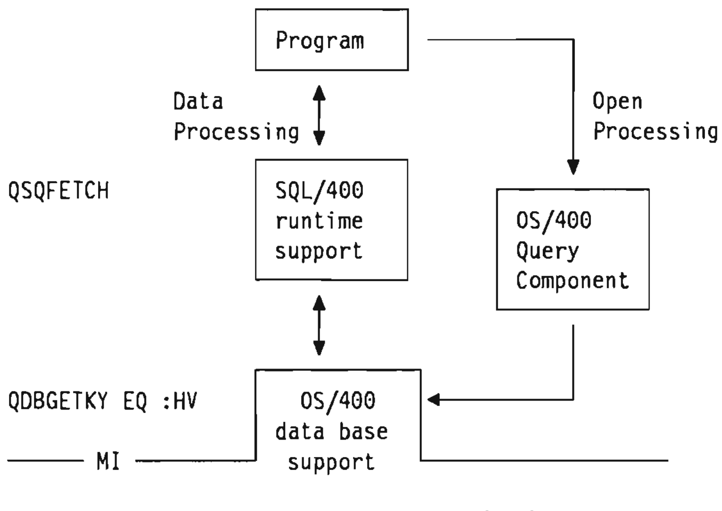
```
CREATE INDEX indx1 ON sometable (column1 asc)

SELECT *
FROM   sometable
WHERE  column1 = 'C'
ORDER BY column2
```

For the above example, a temporary select/omit access path will be created, containing index entries to rows which have *column1* = *'C'*. Once the temporary index is created an ODP will be created using the keyed sequence access path.

File Management Row Selection: SQL runtime support allows an additional selection method exclusively for queries using host variables in the WHERE clause: *File Management Row Selection*. The sole reason for the implementation of this method is to allow queries to reuse ODP's (thus avoid redundant file opens and closes).

Prior to OS/400 Release 3.0, an ODP could not be created which handled variables. Instead, the current value of the variable was defined as a literal in the definition of the ODP at open time. This restriction imposed a significant penalty for queries which were executed in a loop specifying different values for the host variables. (See Figure 5-2 on page 5-39).

Instead of storing the variable values in the ODP, SQL runtime support uses a *generalized* ODP and resolve the selection using an approach similar to Key Row Positioning. However, unlike Key Row Positioning, SQL runtime support controls the positioning by making specific requests to the database manager for records with a key value (See figure below).



Since SQL runtime support needs to position to the records to be selected, an index is required. The requirements upon this index are very specific. All columns being compared to host variables must be include as key fields in the access path. In fact these columns must be the left-most key fields.

In most transaction environments, the File Management Row Selection would be a good performing variation. That is why you should construct your statement to force the Optimizer to choose this mode. You have to follow some rules to enable the Optimizer to choose File Management Row Selection. The rules described below apply to OS/400 prior to Release 3.0. For Release 3.0 restrictions refer to "Reusable Open Data Path" on page 5-39.

- No column functions in SELECT statements or VIEW definitions
- An index must be available and all columns in the WHERE clause
  - have to be contiguous in the index
  - must be the left-most key fields
- Columns in the ORDER BY clause have to match the index columns
- No index columns targeted for update
- The predicates referenced to host variables have to be in the form *column-name operand :host variable.* Each column referencing to host variables can only be used once in the WHERE clause. All of the host variable predicates (see form above) must be ANDed.

The following table represents a summary of the data management methods discussed.

Table 5-2. Summary of Data Management Methods

Method	Process	Watch out for	Good for	Bad for	Selected when	Why good
Dynamic Row Selection	Reads all rows. Selection criteria applied to data.	Deleted records	> 20% rows of table processed	< 20% rows of table processed	No ORDER BY, GROUP BY, join and no index available	Minimizes page I/O thru pre-fetching
Key Row Selection	Selection criteria applied to index. Usually used with temporary index.		ORDER BY, GROUP BY and join operation	Large number of rows processed	ORDER BY columns match left-most index fields.	Data space only accessed when row matches selection criteria.
Key Row Positioning	Processes ranges of index entries.		< 20% rows of table processed	> 20% rows of table processed		WHERE clause refers to left-most index fields
Column Selection	Index processed AND additional criteria applied to data	Operation requires index AND criteria don't match index		Any situation	Index required. Usually done in combination with Key Row Selection and Key Row Positioning.	
Index from Index	Key Row Positioning on permanent index. Builds temporary index over selected index entries.	Temporary index	Ad hoc queries and infrequent transactions	Transaction environment because of NRODP mode	Existing index does not satisfy processing requirements.	No maintenance for permanent index
File Management Row Selection	Index is used to position to the row.		Transaction processing		Index available, candidate for reusable ODP mode.	Only way in OS/400 Rel. 2 for reusable ODP mode.

Reusability of ODP's

An open data path (ODP) is a path created when a table is opened. This path contains information about the merged file attributes and information returned by input or output operations. The ODP only exists when the table is open. For HLL programs the OS/400 data management allows more than one program to share the same path to the data. You can specify that if a table is opened more than once and an ODP is active for it in the same job, the active ODP for the

table can be used with the current open of the table. A new ODP does not have to be created.

For SQL/400, the possibility of shared open data path is not available, as there is a 1:1 relationship between an SQL statement and the ODP. In a job trace you can see a program call to QDBOPEN when you first encounter an SQL statement. At this time an ODP is created for the SQL statement. This happens for each SQL statement in your program that requires an ODP. Refer to "Multiple Access Plans" on page 5-4 for a list of SQL statements that require an open. When the SQL statement in your program is executed, the information stored in the access plan is used to create the open data path. Depending on the information the Optimizer provides at program compilation (or validation), the open data path will be created as *reusable* or *non-reusable.* These are the two forms of ODP's that we distinguish. The reason for these two implementations is that an SQL statement which has to create an ODP will take about 10-20 times more CPU as an SQL statement which is going to reuse an ODP.

Non-reusable Open Data Path: In the case of non-reusable ODP (NRODP) mode the Licensed Internal Code layer directs the database access. The NRODP's are created for each unique statement execution.

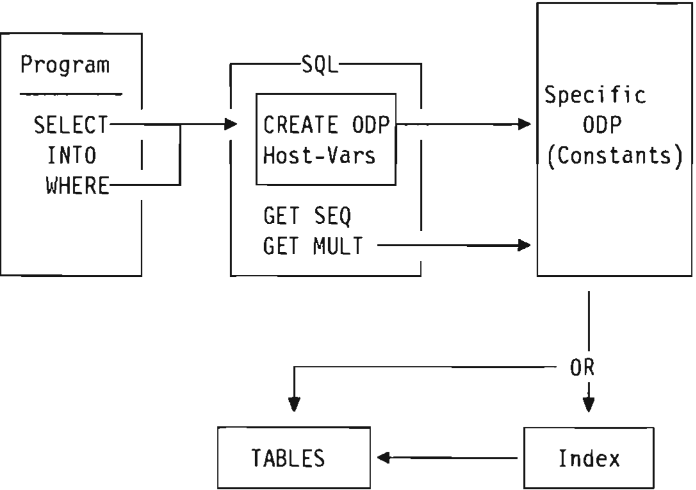


Figure 5-2. Non-reusable Open Data Path Mode

SQL/400 runtime support processes the selection criteria in the WHERE clause in the creation process of the ODP. The host variable value is turned into constants and stored in the ODP. The result is a very specific ODP. As you can see, this form of an ODP is not reusable. If the value changes for the next execution the constant in the ODP does not match it. A new ODP has to be created. The SELECT statement is resolved to the database request GET MULTIPLE or GET SEQUENTIAL. Depending on the Optimizer's decision, these instructions can access through an index or perform data space scan.

Reusable Open Data Path: The reusable ODP (RODP) is created on the first case execution of the SQL statement and used for each subsequent execution. Note that the reusability of the ODP is restricted to the same statement. SQL only reuses ODP's opened by the same statement number, that is, even if you have two identical statements in your program: each statement has it's own

ODP. Therefore it is better to loop around the same statement, than code it multiple times in line.

In *dynamic* SQL the first usage of a prepared statement will require SQL runtime support to create an ODP. Subsequent executions of an already prepared statement can reuse the ODP within the same restrictions as those for static SQL. Keep in mind that when a dynamic statement is re-prepared, all ODP's for the old prepared statement are closed.

The reusability of ODP's is not limited to one invocation of a compilation unit. It works also across invocations as long as the calling program encountered an executable SQL statement. Be aware that ODP's can only be reused if the status of the SQL cursor that created that ODP is closed. Make sure that you explicitly close the cursor before ending the program (see "Reusable ODP's Across Invocations" on page 5-15 for additional information).

There are two forms of reusable ODP's:

1. RODP using a generalized ODP
2. RODP implemented with Interface Supplied Values (available since release 3.0).

A comparison of these two methods will follow:

Starting with OS/400 Release 3.0 the database manager can now support variable data as part of the ODP definition. Therefore it is now possible to reuse an ODP for almost any SQL query that contains host variable references if the statement is run again unless:

- The SQL statement requires a temporary result table (for example the DISTINCT clause)
- A host variable is used to build an temporary 'Index from Index' (See "Index from Index" on page 5-35)
- A host variable is used as a LIKE pattern.

This Interface Supplied Value (ISV) method is initiated at the first running of an SQL statement. At that time the OS/400 Query component will make a copy of the host variable values. It will then define the ODP with pointers which allow LIC to address the copied host variable values. On subsequent reuse of the ODP, the OS/400 Query component refreshes the copied host variable values with the new values supplied by the application program. The ODP still has the address of the area containing the copied values so it is now visible.

This method is very significant in that it releases the application programmer from the entanglement of restrictions imposed with the reusable ODP method via a generalized ODP. It is recommended that that release 3.0 be installed before doing performance testing for new or ported applications. Otherwise, a great deal of effort may be extended modifying the application which is unnecessary under release 3.0 support.

Since the ODP with ISVs is a specific ODP, there are no restrictions on the types of row selection options which can be used.

The clue that can be seen in a JOB TRACE which indicates that a reusable ODP with ISVs is being used is by the evidence that module QQQISVSU is invoked.

This module is part of the OS/400 Query component which is responsible for copying the current host variable into the ISV space. It will be invoked on the first and all subsequent openings and reuses of the ODP. For more information on Interface Supplied Values, see "Module Names to Look for in Trace" on page 5-60.

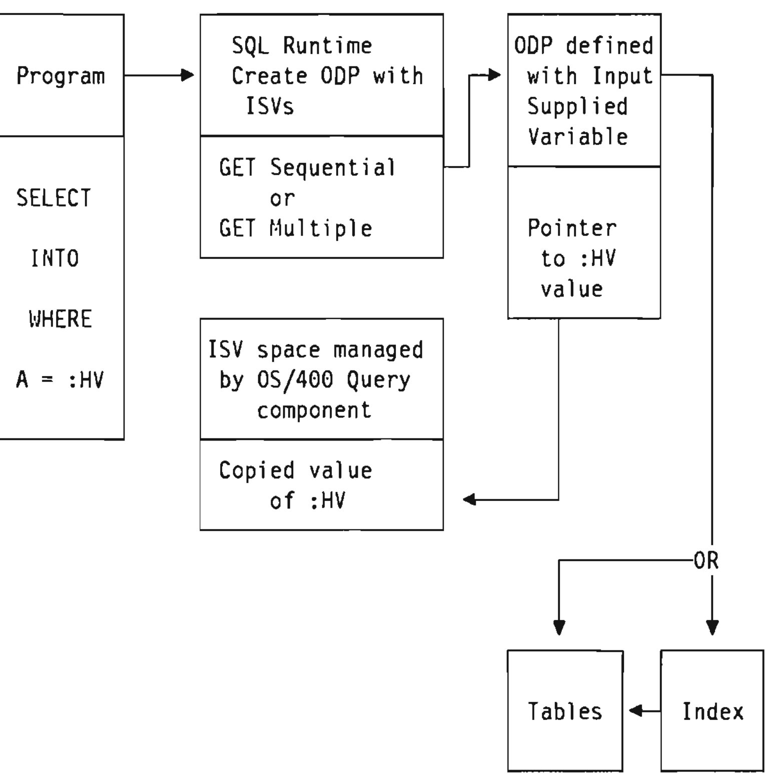


Figure 5-3. Reusable Open Data Path using ISVs

Any ODP that is a generalized ODP for use of File Management Row Selection is reusable (See "File Management Row Selection" on page 5-36 for more information).

The reusable ODP which uses a generalized ODP is the second mode for reusability. This method was the only reusable method available prior to release 3. One must take great care in defining the ORDER BY and WHERE clauses in addition to having the required indexes available in order to utilize this method. Porting SQL interactive applications from another platform to OS/400 may require significant performance tuning in order to match the retrieve nature of the generalized ODP method. However, once the restrictions have been met, performance is very good.

A *generalized* open data path will be created at the first execution of the SELECT statement. It is an ODP where any selection criteria which references a host variable is omitted. In this situation the SELECT is resolved to the database request GET BY KEY. The same type of request is used in the case of an RPG chain operation or COBOL random read.

SQL/400 runtime support hands the value of the host variable to the GET BY KEY request using the ODP to do the data base access. That means *SQL runtime support* directs the operations.

An index is necessary for this access through GET BY KEY. The generalized ODP is not destroyed after the statement execution. It is kept for subsequent executions of the same statement. The decision if the ODP is kept for possible reuse is based on program statistics, which are described below.

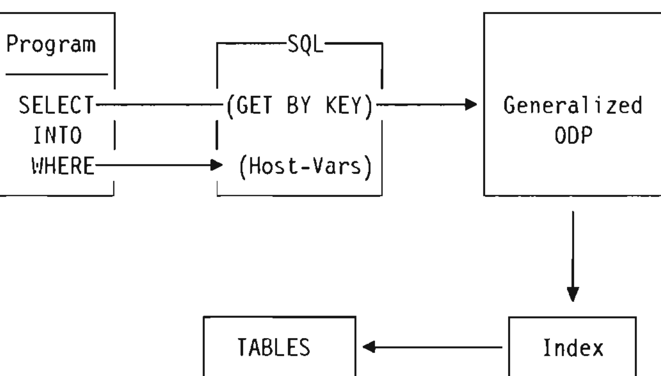


Figure 5-4. Reusable Open Data Path Mode

Reusable versus Non-reusable ODP: Both forms of the open data path have their place. In non-reusable mode, you have to pay the OPEN/CLOSE cost for each SQL statement execution. On the other side, you have higher row processing cost for reusable open data path. Therefore, if you work in the Transaction Processing Environment with only few rows processed per open and many opens, then reusable ODP mode is the recommendation. In a Query Processing Environment, where you select many rows per open, the non-reusable ODP mode would be better.

When the SQL query is a valid candidate for RODP mode, SQL provides program statistics to the Optimizer to enable it to determine the type of ODP to create. The program statistic consist of two ratios:

- *Number of Opens* per *Number of Application Invocations*[5] and
- *Number of Rows retrieved* per *Number of Opens*.

The Optimizer will weigh the cost of avoiding redundant *average number of opens* versus the cost of retrieving the *average number of rows* in order to determine the most efficient mode. The **Rule of Thumb** is if only retrieving a few rows, and the average number of opens is greater than one then reusable ODP mode will be selected.

In some cases the program statistics don't accurately reflect whether the program iterates on an SQL statement:

---

5 The Number of Application Invocation shows how often a program is invoked by an application. That is, if you code in program MAIN:

```
For 1 to 100 do
   call PGMA;
```

The counter for PGMA is increased by 1.

```
WHILE condition DO
   SELECT column list
   FROM yourtable
   WHERE selection criteria
```

Assume that in the above example the condition in the WHILE loop becomes true very seldom (for example once a month). Since OS/400 Release 3, the SQL runtime support keeps track of SQL statements which create and then destroy an ODP during a given application. When SQL runtime support sees that a statement being run a second time and the statement is a valid candidate for RODP mode it will tell the Optimizer to ignore the program statistics and create an reusable ODP. SQL never makes the wrong decision twice.

The following table summarizes the points just discussed.

Table 5-3. Summary of Reusabilty of ODP's

Mode	Reusable ODP	Non-reusable ODP	HLL I/O
Created at (for a given job)	1st execution of the SQL statement	Each execution of the SQL statement	File open
ODP able to be used by	Multiple executions of the same statement in the same program	Used only once, destroyed after statement execution	any I/O statement in a given application
What Row Selection Option	All, except: Temporary result Index from Index Host variable in LIKE	All	Any HLL I/O
ODP closed in static SQL	No SQL programs on invocation stack. When CL command RCLRSC is issued.	At the completion of the SQL statement	Explicit CLOSE or end-of-job
ODP closed in dynamic SQL	Already 'prepared' statements can reuse ODP's, closed when 'reprepared'.	At the completion of the SQL statement	n/a
What happens at reopen	Cursor repositioned	Full open	ODP is unchanged
Suitable environments	Transaction Processing: Few rows per open	Query Processing: Many rows per open	Any
Cost	Higher row processing cost	Higher open/close cost	Lower functions used
Effect on PAG	ODP remains in the PAG until ODP is closed or SQL application ends	The same as left, but the ODP is closed sooner	

The Optimizer

The Optimizer is an important module of the OS/400 Query component as it makes the key decisions for good database performance. Its main objective is to find the cheapest access path to the data. This topic discusses how the optimizer works in general. The exact algorithms are too complex to be described in detail here, and are subject to change from release to release.

Query optimization is a trade off between the time spent to select a query implementation and the time spent to execute it. Query optimization must handle differing user needs:

- Quick interactive response
- Efficient use of total machine resources.

In deciding *how* to access data, the Optimizer:

- Determines possible implementations
- Picks the optimal implementation for the OS/400 Query component to execute.

Precompile Optimization

Figure 5-5 on page 5-45 is an extension of Figure 5-1 on page 5-7 and shows the relationship between the OS/400 Query component, the optimizer and the processes and objects used at precompile time. The SQL precompiler builds the Query Definition Template (QDT) for embedded SQL programs.

The HLL source code created by the SQL/400 precompiler also includes some associated internal control structures. When the HLL compiler is called, it invokes the OS/400 Query component to perform the SQL bind. The compiled SQL program consists of executable code, SQL internal control structures, and possibly multiple QDTs containing access plans.

As mentioned previously, an access plan is created for the processing of each query in a program. Similarly, there is a QDT for each query.

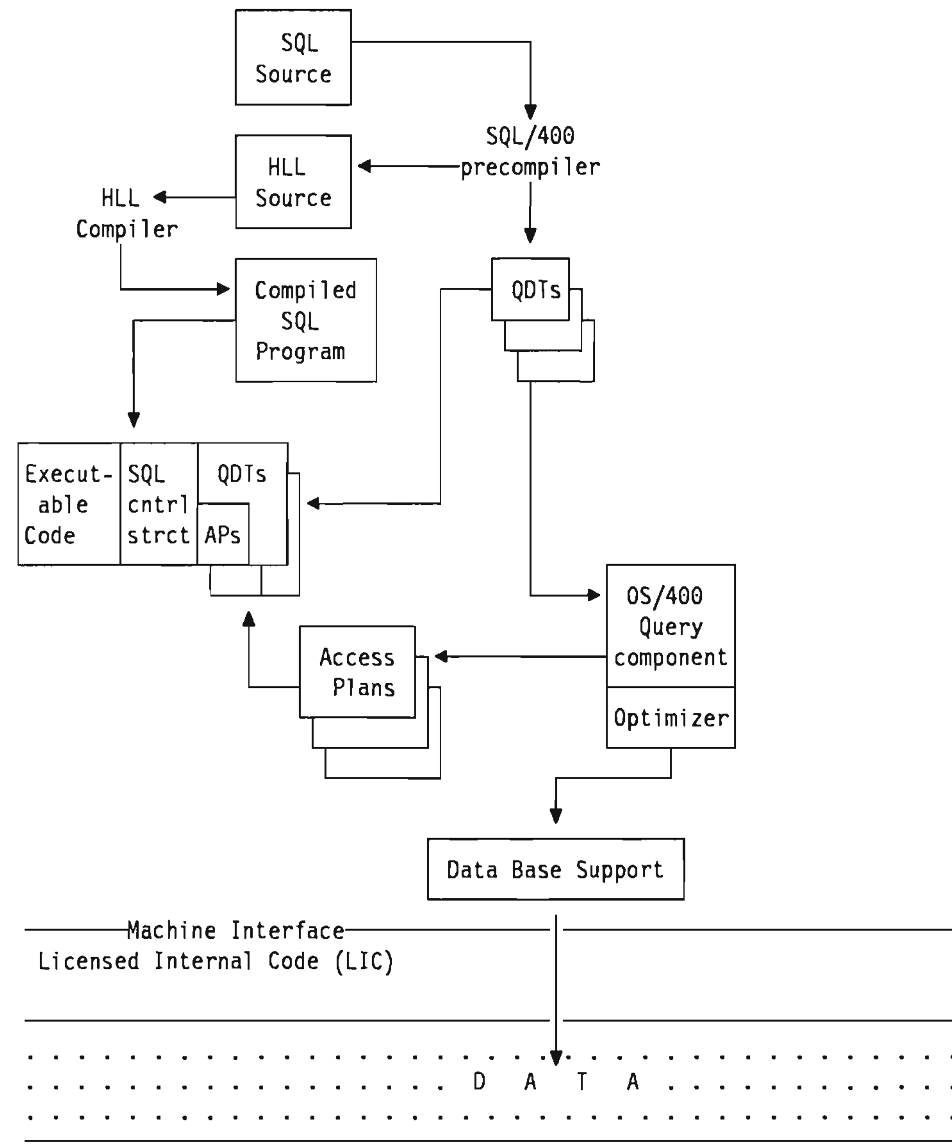


Figure 5-5. Building the Query Definition Template and Creating the Access Plan

Cost Estimation

At runtime, the Optimizer will choose an optimal implementation of the query by calculating an *implementation cost* given the current state of the database. The Optimizer models the access cost of each of the following:

- Reading records directly from the file (Dynamic Row Selection)
- Reading records via an access path (can use any row selection option)
- Creating an access path directly from the file (builds a temporary access path, then can use any row selection option)
- Creating an access path from an access path (Index from Index).

The cost of a particular process is the sum of:

- The start-up cost

- The optimize parameter (*ALLIO, *FIRSTIO or *MINWAIT)[6]

  FIRSTIO Minimize the time required to create the ODP and retrieve the first buffer of records from the file. Biases the optimization towards not creating an index. Either a data scan or an existing index is preferred. This method is used by Interactive SQL/400 and AS/400 Query when output is to the screen. Most embedded SQL statements use FIRSTIO except those listed below under ALLIO. FIRSTIO can be specified for the OPTIMIZE parameter of OPQNRYF CL command.

  ALLIO Minimize the time to process the whole query assuming that all query records are read from the file. Does not bias the optimizer to any particular access method. This method is used by Interactive SQL and AS/400 Query when output is to a Database file or a printer. This is the default option for OPQNRYF OPTIMIZE parameter. Embedded SQL statements that perform set operations, that is UPDATE, DELETE and INSERT with subselect all use ALLIO.

  MINWAIT Minimize delays when reading records from the file. Minimize I/O time at the expense of open time. Biases optimization towards creating an index. Either an index will be created or an existing index used. Only used by OPNQRYF.

- The cost of any access path creations
- The expected number of page faults
- The expected number of records to process.

Page faults and number of records processed may be predicted by:

- Statistics it can obtain:

  From the database objects themselves:

  - File size
  - Record size
  - Index size
  - Key size
  - and so on.

  From the program object for each SQL statement (Refer to "Reusable versus Non-reusable ODP" on page 5-42 for more information on program statistics).

  - Number of open and close operations
  - Number of returned rows per open.

- A weighted measure of the expected number of records to process based on what the relational operators are likely to retrieve:

  These are called *default filter factors*

[6] See the AS/400 Database Guide, SC21-9659, OPNQRYF command for more information on these parameters.

- 10% for equal
- 33% for less-than, greater-than, less-than-equal-to, or greater-than-equal-to
- 90% for not equal
- 25% for RANGE
- 10% for each IN value

- Key Range Estimates

  The Key Range Estimate is based on the primary (left-most) columns of the existing indexes. About six to seven indexes are estimated during the optimization process. The default filter factors may then be further refined by the estimate based on the key range. If an index exists over a column with a primary key, this can be used to estimate the number of keys in that index within a certain range or equal to a given value. The estimate of the number of keys is based on the number of pages and key density of the machine index without actually accessing the keys. Full indexes over columns used in predicates can significantly help optimization.

Page faults and number of records processed are dependent on the type of access the optimizer chooses. Refer to "Data Management Methods" on page 5-31 for more information on access methods. For queries that do not require an index, access can be a Dynamic Row Selection or reading via an existing index. For queries that require an index, for example ORDER BY or GROUP BY is specified, the optimizer may decide to use an existing index, create a new one or create an index from another index.

Access Plan and Validation

At precompile time, or whenever optimization occurs, an optimized plan of how to access the requested data is developed or updated. The information from this optimized plan is kept in what is called a *miniplan*. The miniplan, along with the Query Definition Template (QDT) that is used to interface with the Optimizer, make up an *access plan*. The access plan is saved away as part (in the *associated space*) of the program containing SQL statements.

Access plans are not visible. There is no way to tell what they contain.

An access plan is created with

- CRTSQLxxx, for static SQL (dynamic SQL does not have a stored access plan)
- WRKQRY and save, for AS/400 Query

and stored for future use.

Each time the SQL program is executed, the OS/400 Query component will validate the access plan (for example verify that the tables and indexes used in the plan still exist). If the plan is valid, SQL/400 will use it to access the data. The Optimizer will re-optimize an access plan, if the file or its environment changes considerably (new indexes created, greater than 10% change in number of rows in underlying tables), and SQL will store the access plan with the program.

Re-optimization occurs "only as appropriate" for changes in the structural elements of the tables. That is, an attempt is made to compromise between optimization every time and never, with a simple and fast check to determine when it might be suitable. The underlying idea is that most of the time the stored access plan would be used.

Optimizer Decision-Making Rules

In performing its function, the optimizer works by a general set of guidelines in choosing the best method for accessing data. The general strategies of this process are:

- Determine (default) filter factor for each predicate in the WHERE clause.
- Materialize attributes of the table
- If an index is required, determine the cost of creating an index over the table
- If an index is not required, determine the cost of Dynamic Row Selection
- For each index while *"haven't spent too much time"*
  - Materialize index attributes
  - Determine if the index meets the selection criteria
  - If a selection predicate matches primary key of the index: Perform estimate key range
  - Determine the cost of using the index: Use the estimated page faults and the predicate filter factors to help determine the cost
  - Compare the index cost with previous cost (current best)
  - Pick the cheaper one.
- Choose a reusable or non-reusable ODP based on the program statistics (See "Reusable versus Non-reusable ODP" on page 5-42 for more information on program statistics).

The *"haven't spent too much time"* factor controls how much time is spent picking an implementation. It is based on how much time has been spent so far versus the current best implementation found.

For small files the OS/400 Query component will spend little time in query selection. It will arbitrarily pick one of the first 'good' implementations it finds. For large files, the OS/400 query component will consider more of the indexes. Generally, the optimizer will interrogate five or six indexes (for each file of a join) before running out of optimization time.

Join Optimization

Join is a complex function that requires special attention in order to get good performance. For all join operations, the OS/400 database manager requires the use of an access path over the secondary files in the join. If no usable access path exists, the OS/400 database manager will build it. A keyed access path (index) is not required for the primary file unless there are sort fields selected from this file. It is important to create indexes to match frequently used join selection criteria. The index should match the join-to fields selected from the secondary file.

```
SELECT column list
FROM table01, table02
WHERE table01.fielda = table02.fieldx
```

For the above example, an access path would be required over table02.fieldx. Note that for an inner join, the optimizer may decide to switch the join order of the files. In this case, the previously created index may not be used.

Be as selective as possible on all files to be joined in order to narrow the number of records that will result from the join operation. This can significantly reduce the amount of I/O required to run the query.

Avoid joining files without a WHERE clause. The result would be the *Cartesian Product*. **All** of the records in one file would be joined to **every** record in the other files. This type of operation could result in a large amount of I/O and may affect the overall system performance. The result table of non-equi joins could also contain a large number of records (See chapter "Greater-than, Less-than and Non-equi Joins" on page 3-10 for more information on joins).

OS/400 database manager will always attempt to order the files from *smallest* to *largest*, depending on the estimated number of rows returned. The query will run more efficiently when the files are ordered in this way.

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

1. Determine an access mode for each table. Estimate the number of rows returned for each table.
2. For each join combination, determine a cost based mainly on the expected number of rows returned after the join.

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

   `1-2  2-1  1-3  3-1  1-4  4-1  2-3  3-2  2-4  4-2  3-4  4-3`
3. The combination with the lowest cost determines the primary and first secondary table (for example 2 3 x x).
4. For each remaining table, determine the cost of joining to the first secondary table (for example 3-1 3-4).
5. The file with the lowest join cost when joined to the first secondary file, is the next secondary file (for example 2 3 1 x).
6. Repeat the join cost calculation until the full join order has been determined (for example 2 3 1 4).

Specifying ORDER BY for one of the files always makes this file the primary file for the join operation. This can be useful when trying to determine which file will be used as the primary. Note that if you use the ORDER BY on a column of the *larger* file, this may result in less efficient processing than if the *smallest* file were the primary.

On the other hand you should avoid sequencing on columns of more than one file when a large number of records will be returned.

If a few number of records is being returned, then this may result in a faster response time for the join operation. See "Join Optimization" on page 5-23 for more details on this join performance tip. This will result in sort criteria being specified on at least one secondary file. All selected records from all files must be copied into a temporary file and then an access path must be build over that temporary file to sort the rows. No existing access path will be used during this type of operation. This can be expensive in terms of CPU and I/O, and it can also be a lengthy process if many records are involved.

If there is a necessity for this type of join, the row selection and join criteria become important. The fewer the number of rows selected, the fewer have to be copied into the temporary file for the sort, thus saving CPU, I/O, and response time.

Subquery Optimization

Like Join, Subquery is also a complex operation that requires special facilities to provide good performance. SQL/400 did not support subquery prior to OS/400 Release 3. You could simulate the function of a subquery operation by using multiple SQL statements in a program. You may want to consider using subqueries in place of a current multiple SQL statement approach, because in many cases an SQL subquery will perform better than current user implementations.

Note that this does not mean that subqueries will outperform *all* current methods of simulating these types of operations. You should evaluate the performance of each subquery-type operation before making a decision on whether or not to replace current methods with a subquery. Two different methods for the implementation of subqueries are available. You should understand that the subquery implementation is very complex and all scenarios cannot be shown here. To find the best performing variation of a subquery, and of SQL statements in general, you have to understand enough about the implementation. The choice of which implementation to use will be based on performance with priority given to those subqueries you will most likely run.

Join Implementation: There are two different types of implementing subqueries. The first one, the *join implementation* will be illustrated with the following example. Queries which contain subqueries can be reformulated into a single outer-level SELECT with join conditions replacing the subquery function. The result is the same. You could code the resolved join query in your source program, but it may be more complex than the subquery. The performance difference is minimal.

```
SELECT deptno, lastname, salary
FROM   templ
WHERE  salary IN
         (SELECT salary
         FROM   templ
         WHERE  deptno   = 'D11'
           AND  hiredate > '680101' )
```

Figure 5-6. Statement Entered by the User

The join fields will be x.SALARY = y.SALARY and the additional search conditions specified in the inner SELECT will be y.DEPTNO = 'D11' AND y.HIREDATE > '680101'.

```
SELECT x.deptno, x.lastname, x.salary
FROM   templ x, templ y
WHERE  x.salary = y.salary
  AND  y.deptno   = 'D11'
  AND  y.hiredate > '680101'
```

Figure 5-7. Statement as Converted by the Optimizer

The implementation of a subquery as a join requires an index with the join fields as primary key on the secondary table (refer to "Join Optimization" on page 5-48). Note: If you specify a GROUP BY clause, existing indexes may not be used because a temporary result will be created. How can you find out that a temporary result is created? Test your subqueries in Interactive SQL and use the system request key to display your job. If the "Display open files" screen contains in the File, Library and Member columns an entry '*SUBQUERY' a temporary result for one of the inner SELECT statements is created.

Non-Join Implementation: Some types of queries cannot be reformulated into a join query. For example when the subquery contains an aggregate function (that is SUM, MAX, AVG, and so on.). Subqueries with aggregate functions can be broken down into two outer-level SELECT statements, one of which references the result of the first query. Since Release 3.0 variable data as part of the ODP definition is supported. In general you can say, that for every nested level a reusable ODP with this capability is necessary. As this implementation is very complex, it is shown in the following example and not discussed in detail.

The subquery lists the departments whose payroll budget is smaller than the SUM of it's employee's salaries:

```
SELECT deptno, deptname
FROM tdept x, templ y
WHERE budget <   ( SELECT SUM(salary)
                     FROM templ
                     WHERE x.deptno = y.deptno)
```

Figure 5-8. Statement Entered by the User

The subquery support of SQL/400 provides variables used to establish the connection between inner- and outer-level SELECT.

A temporary table is created from the result of the following SELECT statement:

```
Temporary Table TEMP(T1,T2)

SELECT deptno, SUM(salary)
FROM templ
GROUP BY deptno.
```

The original query can then be reformulated as shown below:

```
SELECT deptno, deptname
FROM tdept, templ
WHERE TEMP.T1 = tdept.deptno
AND   tdept.budget < TEMP.T2
```

The Optimizer could decompose the nested SELECT statement prior to the general query optimization. This would allow for a higher level of optimization and better performance by avoiding the nested execution of the subquery and taking better advantage of query implementation optimization. That is, in order to get an overall optimized subquery you have to find the best performing variation for every inner-level select.

Analyzing Performance Problems

The preceding sections discuss the nature of the AS/400 database implementation, including techniques that you can adopt when writing SQL to attain better performance, and also fundamental constructs used in data retrieval. This section builds on these principles by providing a methodology for tracking down performance problems that you may have with an SQL application. It offers examples and techniques for examining system activity related to your program, with the view to implementing some of the techniques discussed in "Design Guidelines" on page 5-8 to assist in improving performance.

Methodology

Correct use of indexes is the key to improved performance of SQL applications. For good performance, we want to ensure that a query will use the index we create. This topic examines four techniques that help us to identify events that occur when we execute our SQL program. In understanding which OS/400 modules are called, and how long they are active, you can determine whether an existing index is being used or not, and if a temporary index is created.

As the programmer or analyst, you are responsible for the creation of indexes that will help particular SQL statements to perform better. It is important to note that merely creating an index without thought for the structure of the index will not always improve, and sometimes may degrade performance of a particular SELECT. The key tasks in determining correct index usage are:

- Is there an index?
- Does the index include all of the columns required by the optimizer?
- Are there many indexes?

- Are there too many indexes?
- Is the best index being used by the optimizer?
- Are there redundant indexes?
- Are redundant indexes impacting insert, update, delete performance?

Determining Indexes

There are a number of ways of determining the index structure for tables. There are basically three approaches:

- ANZDBF and ANZDBFKEY CL Commands
- Select on SQL Catalogs
- DSPDBR CL Command.

ANZDBF and ANZDBKEY: These are two CL commands that produce reports summarizing the relationships between physical and logical files. ANZDBFKEY produces the report that we really need for identifying indexes, but ANZDBF must be run first. Both commands are part of the Performance Tools (5728-PT1), so will not be available on every AS/400 system.

The Analyze Database File (ANZDBF) command produces two reports that show the physical and logical files in a set of libraries and the relationships between the files. It saves the information in a database file for further analysis by the Analyze Database File Keys (ANZDBFKEY) command. The data is saved in member QAPTAZDR of the database file QPFRDATA/QAPTAZDR.

The Analyze Database File Keys (ANZDBFKEY) command produces, from the data generated by the Analyze Database File (ANZDBF) command, two reports showing the key structure of the database files. Be aware that if you analyze a library that is a collection, the report will contain extra details regarding the catalog objects, as they are logical files built over the data dictionary objects in the collection.

Select on SQL Catalogs: This method provides a way to determine what indexes exist for a table, or group of tables. It makes use of the catalog tables in an SQL collection. The catalog contains information about indexes and keys of indexes in two main views:

- SYSINDEXES
- SYSKEYS.

If your table is not in an SQL collection, then for problem solving purposes it is worthwhile to temporarily move or copy it into the collection containing the indexes. You can then perform the following join to retrieve the structure of these tables and the indexes built upon them:

```
SELECT TBNAME, DBNAME, NAME, COLNAME, COLSEQ, ORDERING
  FROM collection/SYSINDEXES, collection/SYSKEYS
  WHERE NAME = IXNAME
  AND TBNAME = 'YOURTABLE' 10
```

Note that the last WHERE condition **10** is optional, depending on whether you wish to list indexes for just one table, or all tables in the collection. Your table name should be in uppercase.

Display Data Base Relations (DSPDBR) CL Command: The Display Data Base Relations (DSPDBR) CL command can also be used, but this does not list index column names and sequence. You can use the Display File Description (DSPFD) CL command for each index to list this information.

Identifying Problem Code

One of the most significant steps that can be used when solving *any* programming problem, is to isolate problem code into a smaller unit. This offers many advantages, including being more manageable, and removing distracting details that are not related to the problem itself.

For instance, if you decide that you have a problem with an SQL SELECT in a program, then you should copy only the embedded SELECT statement and any required host variable definitions into a new program, which is likely to be much smaller. Re-execute this smaller program first, to see if you still have a performance problem with the SQL statement. If you have established this, then you can progress with the various problem analysis methods described here.

This isolation process is especially useful when performing a job trace, as trace tends to produce copious quantities of output. Too much information in a trace is misleading, and this isolation process helps to speed up problem analysis.

Job Log and Debug Mode

Most SQL error messages and return codes are displayed in the joblog. This offers the programmer a facility to interrogate status for most SQL statements. Therefore you do not have to write any user code to extract this information from the SQLCA, and display it on the screen or in a report when developing or testing an application. In debug mode, more SQL information is externalized to the user, that is, more information is displayed in the job log. In this mode, completion messages are also displayed. They provide status of successful execution of SQL statements.

You will find that all SQL statements except SELECT have information reflected in the job log. This includes all DDL and DML statements. You can use the job log after executing an SQL program using the DSPJOBLOG CL command, or whilst testing statements in Interactive SQL, from the Display Current Job option of the System Request menu. The messages in the job log can somewhat be used as a trace facility, where you can see the completion message for the statement being performed and get an idea of the time it took to execute. The Job Log can be printed using the signoff CL Command with the *LIST option, that is: `SIGNOFF *LIST`.

Interactive SQL

Interactive SQL is the key tool for identifying the source of any SQL performance problem. Previously we discussed the advantage gained by isolating problem code. You can go one step further in an SQL environment and test most Static SQL statements in Interactive SQL. This will allow you to see the execution characteristics of only your SQL statement. Unless you can simulate the resulting dynamic SQL statement as a static statement, you will have to test dynamic SQL statements in a program.

How to Determine if an Index is Being Used: If you are running a SELECT in Interactive SQL, there is a method for finding out what index is being used. To do this, you need to display the returned rows to the screen. The steps are as follows:

- Enter your SELECT statement

When the results are displayed:

- Press System Request Key (and Enter)
- From the System Request menu: Option 3 - Display current job
- From the Display Job menu: Option 14 - Display open files, if active

You should now see the Display Open Files screen, which is a list of all files opened at the current point in time. It will look something like this:

```
                               Display Open Files

 Job . . :   DSP010002      User . . :   QPGMR          Number . . . :   008122
 Number of open data paths . . . . . . . . . . :      8


                    Member/     Record      File   I/O    ----Open--- Relative
 File       Library Device      Format      Type  Count   Opt Shr-Nbr  Record
 QDUI132    QSYS       DSP010202   USRRCD      DSP      11   IO  NO
 QSQIMAIN   QSQL       DSP010202               DSP       1   IO  NO
 QSQISE     QSQL       DSP010202   BODY        DSP       7   IO  NO
 LBRIDX2    DBITRK  11 LBRIDX2                 LGL       0   I   NO
 LBRSTATS   DBITRK  10 LBRSTATS    FORMAT0001  PHY       2   I   NO          32
 QDUI132    QSYS       DSP010202   USRRCD      DSP       2   IO  NO
 QDDSPOF    QSYS       DSP010202   DETAIL      DSP       8   IO  NO
 QDQUWSRUN  QSYS       DSP010202   F36RPTLIVE  DSP       2   IO  YES  1




 Press Enter to continue.

 F3=Exit   F5=Refresh   F12=Cancel     F16=Job menu
```

In the above example, the user is performing a select on the table LBRSTATS in collection DBITRK. The information displayed **10** shows the file name, library (collection), record format, and an I/O count for the table. In processing this SELECT, the optimizer makes use of an index, LBRIDX2. You can see that the index name is actually displayed on this screen **11**. It is clear from its name, that this is an Index. However another clue is the File Type column. This shows the table as PHY (a physical file), and the index as LGL (a logical file). Another clue is the I/O count column containing an entry of zero for the Index.

This technique applies to all permanent indexes, either created with CREATE INDEX or as a Logical File with attributes similar to an SQL Index.

For best performance, a display as shown above is what we want. We want the optimizer to make use of our index.

Identifying a Temporary Index

For large tables (or slower performing queries) your goal should be to avoid the optimizer having to create its own temporary index. This can be done by creating your own permanent index for the optimizer to use.

Unfortunately, there is no easy way of identifying if a temporary index is created or used. There are two scenarios that we need to consider:

- Creating a new index for the query
- Creating an index from an index.

To determine if either of these take place, there is one method that requires strict attention to the screen whilst the SELECT is being performed. You will notice that on the screen message line (line 24), a program status message is briefly displayed whilst the query is processing. This is not recorded anywhere on the system for you to interrogate later. You may need to execute the query multiple times, depending on the query itself, before you can fully read the message.

Creating a New Index for the SELECT: When the optimizer decides that a **new** index is required, you will see a message on line 24 of the screen as follows:

```
                          Enter SQL Statements

Type SQL statement, press Enter.
===> select * from DBITRK/LBRSTATS
     order by lbruprod, lbritems

                                                                  Bottom
F3=Exit   F4=Prompt   F6=Insert line   F9=Retrieve   F10=Copy line
F12=Cancel            F13=Services     F24=More keys
Query running.  Building access path for file LBRSTATS in DBITRK.
```

In fact, what you will see when this query is processed, are three messages that are displayed in turn:

```
Query running.
Query running.  Building access path for file LBRSTATS in DBITRK.
Query running.  12000 records selected.  Selection complete.
```

Because this table contains 12000 rows, the second message is displayed for a longer period than-for smaller tables, and hence is easier to read.

If we use the method previously described for displaying the name of the index (using the Display Current Job menu), no index name will be displayed in this instance, as the index created is an internal temporary object.

Your approach to optimizing SQL performance should be to reduce the time taken to create this temporary index, or eliminate the need for it altogether by creating a permanent index for later use.

Creating an Index From an Index: In some instances, a query may require data in a particular sequence such that an index that exactly suits the query is not available. However, there may be an index which offers one or more primary keys of the index required. In this case, the optimizer may build a temporary index *over* the permanent index. When this happens, you will see messages like the following:

```
                           Enter SQL Statements

Type SQL statement, press Enter.
===> select * from DBITRK/LBRSTATS
     where lbruprod = 580
     and lbrstati < 'B'
     order by lbrstatc

                                                                     Bottom
F3=Exit   F4=Prompt   F6=Insert line   F9=Retrieve   F10=Copy line
F12=Cancel            F13=Services     F24=More keys
Query running.  Building access path from file LBRIDX6 in DBITRK.
```

What you will notice here is that the message contains "Building access path **from**" instead of "Building access path **for**" as before. Also the permanent index name is named in the message instead of the table name.

In this example, LBRIDX6 includes columns LBRUPROD and LBRITEMS, both in ascending sequence. There is no other index which includes LBRSTATI or LBRSTATC. The ORDER BY clause can contain *any* columns to force the optimizer to build a temporary index from a permanent index.

In fact, what you will see when this query is processed are three messages that are displayed in turn:

```
Query running.
Query running.  Building access path from file LBRIDX6 in DBITRK.
Query running.  209 records selected.  Selection complete.
```

As mentioned before, if we use the method previously described for displaying the name of the index (using the Display Current Job menu), no index name will

be displayed in this instance, as the index created is an internal temporary object, even though we are actually making use of a permanent index.

The conditions that cause the optimizer to use this technique are discussed in detail in "Index from Index" on page 5-35.

Inefficient Indexes

You can now identify when the optimizer is creating its own index and when it is using one that you explicitly create. You need to consider how to identify a user-provided index that is inefficient.

This process is like that of identifying optimizer-created indexes. You must issue a SELECT in Interactive SQL and interpret the messages that are rapidly displayed on the message line of the screen. For a SELECT that uses an inefficient index, you will notice something like the following:

```
                              Enter SQL Statements

 Type SQL statement, press Enter.
 ===> select ITLITEMN, ITLLOCAN, ITLWKCEN, ITLTENTR, ITLNALTW
      from DBITRK/ITLOCATN
      where ITLWKCEN >= 'WK200'
      and ITLTENTR = '1234'
      and ITLNALTW <= 'WK250'

                                                                  Bottom
 F3=Exit   F4=Prompt   F6=Insert line   F9=Retrieve   F10=Copy line
 F12=Cancel            F13=Services     F24=More keys
 Query running. 10 records selected, 100 processed.
```

When the ratio of records selected to records processed is low and the table contains a large number of rows, you will notice that the message slowly increments the numbers on the message line as it processes the table. In this instance, you might find that the optimizer is either not using an index or the index being used is inefficient. You should create indexes such that the number of rows processed is minimized, that is, try not to process all rows in a table.

In this environment, you will not often get a true indication of an inefficient index if your interactive session displays output to the screen. This is because the first screen of data will be displayed as soon as there are enough rows to fill the screen. When SELECT output is displayed at the screen, it is optimized so that the user sees a result as soon as possible, that is, a buffer full of rows.

To get a true indication of index efficiency, you should use F13=Services, and take the option to change the SELECT output device. Set the output device to Printer or Database file (Printer is probably the easiest). Now re-issue the

SELECT statement and watch the message line. It will remain on the screen much longer than before.

Be aware that the optimization techniques employed in Interactive SQL are different between various SELECT output options. Output to the display is processed like embedded SQL which both use the same optimization goal, that is, return some output as soon as possible. Output to a Printer or Database file is optimized based on returning *all* data. (See "Cost Estimation" on page 5-45 for more information on optimization bias). You therefore must ensure that when producing output to the printer that the the SELECT employs the same optimization techniques (that is uses an index or not) as when output is sent to the display.

Work With Jobs Displays

You can use the Work with Active Jobs (WRKACTJOB) or Work with Submitted Jobs (WRKSBMJOB) displays as another way of detecting if the Optimizer builds a temporary index. For a query that creates a temporary index, the "Function" column on either of these displays will contain an entry like the following:

```
                            Work with Active Jobs                    ROCHESTR
                                                          06/21/90  12:55:42
 CPU %:    17.8     Elapsed time:    00:00:00     Active jobs:   33

 Type options, press Enter.
   2=Change    3=Hold   4=End      5=Work with   6=Release   8=Spooled files
   9=Exclude   10=Program stack    11=Locks      13=Disconnect

 Opt  Subsystem/Job  User         Type  CPU %  Function         Status
 __     PCLAN04      QPC          EVK      .0  *  -PASSTHRU      EVTW
 __     RCHAS008     QUSER        EVK      .0  *  -PASSTHRU      EVTW
 __   QCTL           QSYS         SBS      .0                    DEQW
 __     QJSCCPY      QPGMR        BCH      .0  PGM-QSCCPY        DEQW
 __     QOACLNUP     QPGMR        BCH      .0  DLY-22:00:00      DLYW
 __     QPFRCOL      QPGMR        ASJ      .0  PGM-QPMLWAIT      EVTW
 __   QINTER         QSYS         SBS      .0                    DEQW
 __     DSP010202    SQLTEST      INT      .0  IDX-ITLOCATN 12   RUN
 __     DSP010203    QSYSOPR      INT      .0  CMD-WRKACTJOB     RUN
                                                                     More...
 Parameters or command
 ===> ________________________________________________________________________
 F3=Exit      F5=Refresh   F10=Restart statistics   F11=Display elapsed data
 F12=Cancel   F24=More keys
```

In the example above **12**, IDX shows the create index process running. This is also shown when you issue the statement CREATE INDEX. When a temporary index is created, the name that appears with IDX refers to the table upon which the index is built. When performing a CREATE INDEX, this name is that of the created index.

Job Trace

There are a number of trace facilities available on the AS/400 system, including:

- Trace Job (TRCJOB) - a component of OS/400
- Start Job Trace (STRJOBTRC) - a component of Performance Tools
- Trace Internal (TRCINT) - for IBM Service

The recommended trace facility to use is TRCJOB, as it is part of OS/400 and therefore will be available on every AS/400 system. It provides enough information in most scenarios. The TRCJOB command controls traces of program calls and returns that occur in the current job. It can trace module flow, operating system data acquisition, or both.

STRJOBTRC, being a component of the Performance Tools program product, will not always be available on every AS/400 system, as not all systems have Performance Tools. If the Performance Tools are available, then this could be used instead of TRCJOB, as it is essentially the same, but provides some additional information on I/O activity of modules called during the trace period. A trace record is generated for every external (program) call and return, exception, message, and workstation wait in the job.

TRCINT provides too much detail for most users. It is primarily designed for problem analysis. It controls traces of internal events associated with the current job that occur at a level below the machine interface. In addition to OS/400 module names, it also contains names of VMC (Vertical MicroCode) programs which are called from the OS/400 modules. The trace internal report tends to produce 20 times more output for the same user process as the other trace tools. The creation of this trace report takes some time to complete.

The Nature Of Trace

All of these trace facilities involve extra activity to record events that occur during the execution of a program or function. You should be aware that any program executing whilst trace is active will execute **slower** than if it were run without trace. Because of this, you cannot trust the execution times for a program when run in trace mode. Different trace facilities have different impacts on program execution time.

An additional note about TRCJOB and STRJOBTRC, is that even when the trace is ended, there are still extra tasks executing that effect the performance of your job. The only way to be sure the execution elapsed time is accurate, is to signoff and signon again before re-executing your program.

Running the Trace

When you make the decision to run a trace to more closely examine the characteristics of your executing program, you will need to ensure that you don't trace too much activity. Trace tends to produce large quantities of output. You should minimize the output created by starting the trace, running your program, then stopping the trace as soon as possible. The comments made before about program isolation also apply. Try and minimize the amount of code in the program being traced.

Once you have produced the trace output, it is best kept online, rather than printing a copy. You can make use of the searching capabilities of the Display Spooled File function to quickly locate those parts of the trace that are relevant.

Module Names to Look for in Trace

When reading a trace report, it is important to understand what is happening in the flow of your application. The main modules you will see for database access all begin with QQQ.

There are fifteen modules comprising the OS/400 Query component. All fifteen may require invocation for running a query. The order of their invocation is

apparent from Figure 5-9 on page 5-63. First validation processing (QQQVALID, QQQITEMP, QQQVWCMP, and QQQVWFLD). Secondly, optimization processing (QQQOPTIM). Thirdly, implementation processing (QQQIMPLE). Potentially, recovery of SQL Views may be necessary (QDBFIXIT and QQQVWRCY). Finally, module QQQQEXIT is invoked at close time to clean up query created objects.

The Create Access Plan task firstly performs validation processing. QQQVALID, QQQITEMP, QQQVWCMP, and QQQVWFLD could be called by QQQQUERY to check your query definition. Again, recovery of SQL Views (QDBFIXIT and QQQVWRCY) and cleaning up the query environment (QQQQEXIT) are required operations. Only after this, can the optimization processing necessary to run a query occur.

QQQQUERY - Query Main Line: This is the initial module for Create Access Plan and Run Query processing, and is known as the OS/400 Query component. It subsequently invokes the validation, optimization, and implementation query functions, as required. This module also handles the overall coordination of multiple query definitions (for example UNION) and builds the access plan returned on the Create Access Plan and possibly the Run Query requests.

QQQVALID - Query Validation: Validates the query definition templates including the enforcement of concurrency restrictions and authorization constraints. It builds the subset of Query Internal Structures specific to validation processing. This includes almost all Query Internal Structures, except for those built during optimization processing.

QQQVWCMP - Query SQL View Composition: Composes a single query definition template from the two or more query definitions posed by your query and any SQL Views taking part in your query. Each queried View has its own query definition template that must be merged with the user's query of that View.

QQQVWFLD - Query SQL View Field Composition: Called by module QQQVWCMP during query composition for processing of field references. This processing is located in a module separate from QQQVWCMP because it may need to recursively reinvoke itself.

QQQSETUP - Query Set Up Processing: Called by QQQQUERY and QQQVALID, this module creates the internal work spaces used by query processing.

QQQOPTIM - Query Optimization: This is the module knows as the Optimizer. It optimizes access to the result defined by a single query definition template. The optimization processing is primarily concerned with determining what, if any, indexes should be used or built when running the query, and in what order multiple files should be joined.

QQQSQCMP - Query Subquery Join Composition: This module is called if a query contains one or more subqueries. It attempts to transform as much of the subquery into join specifications as is possible. QQQOPTIM will then determine the cost of implementation from the join composite. QQQQUERY will compare the default (ISV) cost to the join cost and the less expensive implementation method will be used.

QQQISVSU - Query Interface Supplied Values (ISV) Set Up Processing: When called by QQQVALID, this module creates ISV spaces, and sets the initial ISV pointer values. These ISV spaces are used by QQQGET to retrieve records

when a subquery is being processed. This relationship is not shown in the following diagram. When QQQISVSU is called by QQQQUERY, this module updates or refreshes these ISV pointer values with the latest values of host variables. ISVs are primarily used for processing subqueries and providing reusable ODPs.

QQQIMPLE* - *Query Implementation: The optimized access to the query result defined by the query definition template is implemented by creating a query MI Cursor object. MI Data Space Index objects may also be created for selection, joining, and ordering requirements. If this module is active for an extended period of time (greater than 0.1 CPU seconds in most cases) then it is likely that it is creating a temporary index.

QQQITEMP* - *Query Temporary Result: Creates temporary copies of files participating in the query, or copies of data produced from intermediate versions of the final query definition. This is most often used in processing a UNION. Temporary files are created in library QTEMP.

QQQACTIV* - *Query Activation of Embedded SQL Statements: This module is called by QQQQUERY to activate the query defined by the implementation process. This function must be done from a separate module so that your authority is propagated to the cursor (not QSYS authority).

QQQQEXIT* - *Query Exit: This is the invocation exit program for module QQQQUERY. It may also be called directly for some errors detected by module QQQQUERY. It cleans up the query environment by closing opened files and destroying any temporary objects.

QDBFIXIT* - *Data Base Recovery Processing: Invoked by module QQQQUERY when recovery is required for a queried SQL View or the open of a View. This module is actually owned by the Data Base component.

QQQVWRCY* - *Query SQL View Recovery: Performs recovery of SQL Views that are in an inconsistent state. Recovery becomes necessary when the structure of the View was being modified when the job or system failed.

QQQGET* - *Query I/O Processing: This module provides an interface to the Database GET modules for those queries that use Interface Supplied Values (ISVs). This module is only invoked when subqueries are present, and acts as a relay between SQL and Database. If QQQGET is called only once after the first call to QSQFETCH, then the OS/400 query component is performing a basic subquery. If it is not in the trace at all, but the program is doing a subquery, then it is likely that the subquery has been converted to a join, and you should see QQQSQCMP in the trace instead. If QQQGET is called repetitively, then the OS/400 Query component is performing a correlated subquery. If your subquery can be rewritten to use basic subquery or join composition, then you will often achieve better performance.

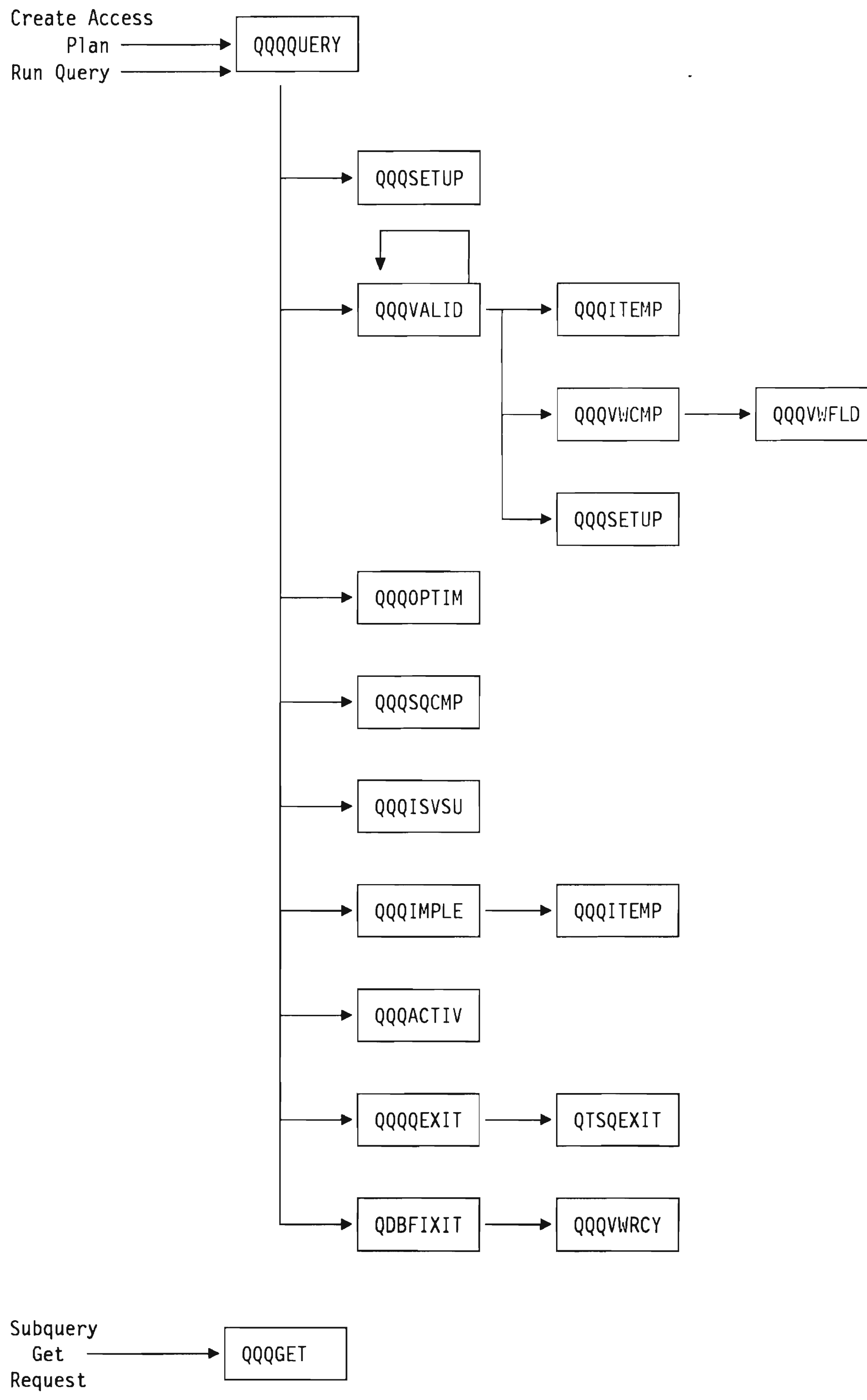


Figure 5-9. OS/400 Query Component Module Flow

Other Significant Modules: In addition to the QQQ modules described above, there are some other modules that you will see in analyzing a trace of an SQL program. Some of the more frequently used modules are:

Module	Description
QDBGETM	DataBase get multiple - uses blocking to retrieve data
QDBGETKY	DataBase get by key - uses File Management
QDBGETSQ	DataBase get sequential
QMH?????	Message handling routines
QSQROUTE	The first called SQL routine - for embedded SQL. When browsing a trace report on the screen, use the search facility to locate this module first.
QSQXCUTE	Interactive SQL execution runtime interpreter module. The first called SQL routine.
QSQAUTH	Handle GRANT/REVOKE statements
QSQBIND	SQL Bind
QSQBLQDT	SQL Query Definition Template Builder
QSQCLOSE	SQL Hard Close Pseudo Closed Cursors
QSQCRTDB	SQL Create Data Base
QSQCRTI	SQL create index
QSQCRTT	Create Table
QSQCRTV	SQL CREATE VIEW
QSQDELET	SQL DELETE
QSQDESC	Describe Runtime Support (Dynamic SQL)
QSQDROP	Handle DROP statement
QSQFETCH	SQL FETCH and Embedded SELECT
QSQINS	SQL INSERT
QSQLABEL	Handle COMMENT & LABEL statements
QSQLCCR	SQL LOCK, CLOSE, COMMIT, and ROLLBACK
QSQOPEN	SQL Open
QSQPREP	Prepare Runtime Support (Dynamic SQL)
QSQPSTAB	Dictionary Services - precompiler/parser
QSQRAPLY	SQL Semantic routines
QSQRCHK	SQL Data Checker
QSQRLEX	SQL Lexical Scan Routines
QSQROUTE	SQL Router
QSQRPARS	SQL Parser Machine
QSQRPTAB	SQL Parse Tables
QSQRTBLS	Scan Tables for the SQL Grammar
QSQRTOKR	SQL Tokenizer Routines
QSQRXLTR	SQL parser translator module

QSQRXTRT Extract field definitions for tables

QSQUPDAT SQL UPDATE

QSQXIT SQL Exit

Data Scope Message SQL7904 should appear only once if your objective is to reuse ODPs. This should occur when your dummy SQL statement (such as DESCRIBE) is executed. If there is more than one occurrence of this scope message, then your ODPs are not reusable. See "Reusability of ODP's" on page 5-38 for more information on reusable ODPs.

Other Trace Events: If you have created an index, and SQL is using that index for processing a query, then you will see the index name in the trace. It will appear with the module QDMCOPEN. The same module will later appear with the table name.

Performance Tools

The AS/400 Performance Tools licensed program product (5728-PT1) contains some tools that you can use to help you to identify performance-related problems with your application. You have seen some references to Performance Tools already in this topic. Some of the facilities that you can use are as follows:

- ANZDBF and ANZDBFKEY CL commands
- STRJOBTRC CL command
- Timing and Paging Statistics Tool (TPST).

Previously discussed was the use of ANZDBF, ANZDBFKEY and STRJOBTRC CL commands. This section mainly refers to use of TPST.

You will find that for most SQL performance tasks, the methods described previously will assist in the majority of cases. You can use TPST for specific performance measurement, rather than tracking down performance bottlenecks.

TPST: TPST is in fact not part of the Performance Tools program product. It is a performance measurement tool, however, and is a PRPQ (5799-DCG). It has multiple uses in any performance environment which deals with performance at a program or module level. It also offers options to store measured performance data into a database file for later extraction into a report.

Here are some points relevant to the use of TPST:

- TPST measures: Synchronous and Asynchronous, Cumulative and Non-Cumulative:

  CPU seconds, Database Reads, Database Writes, non-Database Reads, non-Database Writes

- You can measure cumulative values for these totals also. This means that you can measure the value for the stated program, and all programs and modules it calls. This is especially useful when measuring at a program level.

- If you are looking at specific functions within an SQL program, you can measure only the activity for the module you require. See "Module Names to Look for in Trace" on page 5-60 for module names and the functions they perform.
- TPST is best used in a standalone environment so that module activity of other users is not recorded.
- TPST output can be written to a database file. You can then use the List TPST(LSTTPST) command or your own program to extract measured figures.

Positioning: SQL vs Native File Management

Many application developers who code in high level languages such as RPG and COBOL may be looking at SQL/400 as an alternative for traditional database I/O. This topic gives guidance on common operations performed in applications from a performance viewpoint. The objective of this is to assist you in deciding where SQL can be used to provide better performance or more efficient coding.

Detailed measurements were taken to record the performance of the programs used in specific tests across 28 different environments ranging from batch processing tasks, to random processing and statistical processing, including read, insert, update and delete variations, where applicable. The actual results are too large to be published here. Instead, summary findings are described to give you an **indication** of what you may achieve on your own system.

Environment

The guidelines here are based on testing done by the IBM Australia Field Systems Centre. Comparisons are based on results obtained in these tests only, and will not necessarily apply to every AS/400 system.

The tests were performed in the following environment:

9406-B45

OS/400 Release 3.0

48 MB Memory

3.3 GB DASD

Checksum Off

No User-ASP's

Active Subsystems:

QCTL

QINTER

QBATCH

Tests executed from a workstation under QINTER

Machine pool size 6700K

Base pool size 3850K

JOBD = QDFTJOBD

QINTER pool size 30MB, activity level 18

All devices (screens and printers) varied off except the test display and the console

DASD utilization = 48%

Disclaimer

The information contained in this topic has not been submitted to any formal review and is distributed on an "as-is" basis without warranty either expressed or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends upon the customer's ability to evaluate them and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results may be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The performance data contained in this document was obtained in a controlled environment based on the use of specific data and is presented only to illustrate techniques and procedures to assist readers in the understanding of performance characteristics of SQL/400. The results obtained in other operating environments may vary significantly. Users of this document should verify the applicable data in their specific environment.

Native File Management vs SQL

SQL contains some constructs necessary for its flexibility which adversely can affect its performance in some situations. These include:

- Field Mapping

  Unlike native file management I/O, SQL maps each column in a row individually. File management maps the entire row as one unit. Therefore you will find that there is extra overhead for each column returned. This is especially noticeable when doing FETCH operations, where there is more than one row returned. If a large number of rows are fetched in a loop, then this overhead is multiplied for each row.

- Error Handling

  SQL handles error conditions by using the SQLCA. This is called *return code* error handling, where a result is set if the statement completed in error or successfully. File management I/O uses exception based error processing, where error processing only occurs if a statement does **not** complete successfully. The additional work done by SQL in setting the SQLCA values after the execution of *each* SQL statement again is multiplied when many rows are processed.

- Fetch row at a time

  When SQL performs a FETCH operation, each request calls the QSQFETCH module to return the data to the specified host variables. This is different to file management I/O where a block of records are returned, and multiple READ operations access the same block for the next record. File management performance will be closer to SQL FETCH performance when commitment control is used in this environment

SQL in General

SQL is better suited to set at a time processing, especially in insert, update and delete processing. For processing smaller tables, and smaller proportions of large tables performance of SQL in read (select) tests is adequate, but not as good as non-keyed files where every row is processed. This is because of the points mentioned above regarding field mapping, error processing and FETCH performance.

SQL performs particularly well in special function processing such as calculating statistical information (max, min, avg, sum, count) and LIKE function processing (for an alpha-search environment). A particular point of note about these two scenarios, is the ease with which SQL can be implemented. Alternative environments performing this same function are quite verbose and prone to errors when coding.

SQL should ***not*** be used for single record processing (read, insert, update, or delete) where the ODP is not reused. Without a reusable ODP, SQL requires a full open of the table for every row processed in this scenario, and performs much worse than most other environments.

SQL vs Keyed Files

Keyed Files are the best performing alternative in two areas. Firstly, they provide superior performance when processing single record read, update, and delete operations, when the record must be located by key, that is random processing. No other alternative tested can match logical file performance in this scenario. When processing with an index, SQL performance is second behind keyed file performance. Non-keyed files and OPNQRYF environments were particularly poor in this scenario.

Results for random processing showed that SQL with an index was on average two to four times slower than processing from a keyed logical file when randomly processing 100 rows. If no index was used, then SQL performance was about 60 to 200 times slower than for a keyed logical file. This was in the same order or magnitude as OPNQRYF.

Join Logical Files also perform well compared to SQL which does a dynamic join in a SELECT statement. This is essentially because the access path is created with the CRTLF command. SQL on the other hand creates the access path at execution time (unless an SQL view is used). A join logical file is 2 to 2.3 times faster than SQL in this environment. SQL offers an advantage over join logical files. If the programmer has used the wrong join order in the join logical file, then the application is forced to use that order, whereas SQL will correct for the mistake to optimize performance.

SQL vs Non-keyed Files

Most programmers would consider that keyed physical or logical files are the correct approach for best performance in most application solutions. A significant point of note is that non-keyed files perform well in many environments. In most of the 28 performance tests, the non-keyed environment performed acceptably, and for most of these, it was the best performer, including processing tables of up to 500,000 rows.

In analyzing the details of these tests, the non-keyed environment while having an acceptable elapsed execution time, had higher Total I/O's and CPU seconds in some instances. This implies that in a non-dedicated environment, there would be an adverse effect on other users of the system caused by the execution of programs using non-keyed files.

SQL vs OPNQRYF

In tests that contained both SQL and OPNQRYF environments, results were split into mainly three groups:

- Results where SQL was much faster than OPNQRYF (30 to 75 times faster)
- Results where SQL was up to 10% faster than OPNQRYF
- Results where SQL was significantly slower than OPNQRYF (1.5 to 4 times slower).

SQL is much faster than OPNQRYF when performing dynamic access to a table, when SQL uses an index. If no index is used in these environments, then SQL is much worse.

The use of OPNQRYF is somewhat of a contentious point, as it is relatively difficult to code (the programmer is required not only to code the OPNQRYF statement, but must also code OVRDBF and DLTOVR statements in calls passed via QCMDEXC). SQL on the other hand is much more straightforward, and has the Interactive SQL facility to aid in debugging and testing. The point of contention is the fact that OPNQRYF is a CL command, and is part of the operating system. The SQL Licensed Program Product containing the precompilers and Interactive support is a separate product, and must be purchased before it can be used. SQL runtime support is part of OS/400.

Given the right environment (in about 70% of the tests, SQL performed better than OPNQRYF), SQL is the better choice in terms of execution and coding efficiency. However, programmers will have to be aware of those situations where OPNQRYF surpasses SQL in processing efficiency (despite the fact that it is more difficult to code). OPNQRYF would then be used if performance is a key issue. The situations where OPNQRYF is better are those where the performance advantage gained by the extra optimization techniques of SQL are adversely affected by SQL field mapping and error processing overheads. Areas where you will see OPNQRYF performs better than SQL are:

- "Batch" processing - where a larger percentage of the table is processed, with either complex or simple select conditions
- Join processing - with either complex or simple join conditions.

Static vs Dynamic SQL

In comparing static with dynamic SQL, the overheads expected with dynamic SQL were measurable, with greater total I/O, CPU time, and elapsed time. Dynamic SQL is about 5-10% slower than static SQL in most environments. Dynamic SQL should only be used where extra flexibility is required that is not available by static SQL.

Table of Comparisons

Table 5-4. Chart of Relative Database Techniques Based on Performance Results

Task	Alternatives (best to worst)
Simple Select	non-keyed, OPNQRYF
Complex Select	non-keyed, OPNQRYF
Statistical Select	SQL = OPNQRYF, non-keyed (simple only)
Join Processing	keyed, OPNQRYF
Union	keyed, SQL
Updates - all and sets	non-keyed = SQL, OPNQRYF
Updates - Random	keyed, SQL
Deletes - all	CLRPFM
Deletes - sets	non-keyed = SQL, OPNQRYF
Deletes - Random	keyed, SQL
Static vs Dynamic	Static SQL
Random Reads	keyed, SQL
Insert - Summary File	non-keyed, SQL, OPNQRYF
Insert - Values	non-keyed
Alpha-search	SQL = OPNQRYF

Batch Processing

When processing large files, you will find that the Optimizer will choose Dynamic Row Selection to access the data. This does not always offer the best performance when comparing to a non-keyed file with traditional HLL I/O statements. A Keyed Logical File built over a non-keyed file will often have extremely poor performance as records may not be in the physical sequence of the logical file key. Therefore, when processing from the logical file, excessive paging is done to retrieve required records.

The Importance of Indexes in SQL

Correct index use in processing SQL queries cannot be underestimated. The figures stated above regarding random processing give an indication. If no index is available, you will find that the ODP cannot be reused. Therefore statements executed in a loop will cause multiple opens, and performance will be severely degraded.

SQL Performance Enhancements in Release 3.0

There have been a number of changes to improve functions that support SQL/400 in Release 3.0 of OS/400. These will bring about some performance improvements that will range from small to significant for various applications.

This topic provides a summary of some of these enhancements as a guide to identifying what types of functions will now perform better. The "Improvement" column doesn't represent an average overall improvement for an application, but only for the operations described.

Table 5-5 (Page 1 of 2). Release 3 Performance Enhancements

Item	Description	Improvement
1	**Reusability Restrictions Reduced** In Release 2, columns that were compared to host variables in the WHERE clause had to have an index built over them in order to avoid an open and close upon each execution of the SQL statement. There were other restrictions placed upon the use of host variables in order to avoid opens and closes. Also, each execution of an SQL statement using either a GROUP BY function or a column function (SUM, AVG, and so on) required a full open and close. In Release 3 these restrictions have been removed. Columns without an existing index over them can be compared to a host variable without causing an open and close to be performed for each statement, and host variables may now be used anywhere that is syntactically correct. In addition, SQL statements with GROUP BY functions or column functions will not cause an open and close of the file each time they are executed.	Significant
2	**Allow Reusable ODPs across Invocations** Database cursor is reused and not 'truly' closed until no SQL program is on the invocation stack. This performance improvement was also available under Release 2.0 with PTF 04173 (now superseded by SF04941).	Significant
3	**Subquery performance enhancements** New subquery support offers a more efficient way of what used to require two or more separate queries. Faster subqueries result from using indexes more often and implementing the subquery with a join.	Significant
4	**New Fast Sort allowed for Interactive SQL** For COMMIT(*NONE) SELECTS going to printer or outfile, a new fast sort process will be considered for the ORDER BY for better performance.	Significant
5	**Increased sharing of select/omit access paths** No longer will you have to specify the Select/Omit logical file as the one to query to have the file's access path considered. The OS/400 Query component will consider any Select/Omit logical file over the queried file.	Significant
6	**Reduce number of columns returned in buffer for a read only Declare Cursor** In Release 2, all cursors that were update capable returned *all* underlying columns in the buffer in case you later wished to update a column. Now, only the columns in the SELECT list will be returned resulting in less buffer overhead.	Moderate

Table 5-5 (Page 2 of 2). Release 3 Performance Enhancements

<table>
<tr><th>Item</th><th>Description</th><th>Improvement</th></tr>
<tr><td>7</td><td>Multiple records retrieved after index positioning

Blocking is now used for input only selection, for example:
<pre>DECLARE C1 CURSOR FOR
   SELECT * FROM SOMETABL
   WHERE COL1 = 'A'   (COL1 has an index)</pre>Previously used GETKEY EQUAL followed by GETKEY NEXT EQUAL. This always required multiple single I/O operations. This now uses GET block in place of GETKEY NEXT EQUAL, reducing the number of I/O operations.</td><td>Moderate</td></tr>
<tr><td>8</td><td>Avoid extra record processing for SELECT INTO statements

Where only one row is expected, previously two I/O operations were required to determine if the SQLCODE should be set if more than one row is returned. Two methods are used to reduce this to one I/O operation:
• When blocking is used, the row is checked to see if it is part of a block
• When key retrieval is used, duplicate key feedback is checked</td><td>Moderate</td></tr>
<tr><td>9</td><td>Avoid unnecessary code / streamline code (remove general inefficiencies)</td><td>Moderate</td></tr>
<tr><td>10</td><td>Seize contention reduction

Database is able to lock MI objects at record level instead of at the file to improve concurrency. Previously MI objects were required to be locked at the file level in certain operations. This enhancement allows better concurrent access to objects through reduced seize wait time.</td><td>Moderate</td></tr>
<tr><td>11</td><td>Removed FEOD (Force End of Data) from INSERT with subselect

A different method is used to force insert data to disk without having to force the entire data base object (including object header information).</td><td>Small</td></tr>
<tr><td>12</td><td>Eliminate internal exception on set position

When re-positioning to beginning of file for a reusable ODP, end of file exception (CPF5001) is no longer generated.</td><td>Small</td></tr>
<tr><td>13</td><td>Use less storage when building access paths

When building (or creating) access paths, less impact on other system jobs will result. Previously, up to 80% CPU was used. Now, CPU utilization of create access path is based on a fair proportion of system activity.</td><td>Small</td></tr>
<tr><td>14</td><td>Reduce number of journal entries for SQL update statements

To implement commit control for set-at-a-time operations such as Insert with subselect, Update and Delete, a second level of journaling (transparent to the user) was required.

COMMIT was stopping second level commitment control and then restarting. Stopping and restarting of second level commit no longer occurs, therefore reducing journal entries and associated overheads.</td><td>Small</td></tr>
</table>

6. Distributed Data Management (DDM) Considerations

Distributed Data Management (DDM) allows a user or program on a local system to access remote files via local file processing commands. The fact that the files are really remote is transparent to the application or user. This means that an HLL program can be written to process local and remote files at the same time, with no extra commands necessary in the program to handle the communications between systems to access the remote data.

SQL allows access to files either interactively or from within applications; however SQL does not support DDM file access.[1] This means that HLL programs cannot implement SQL table/file access on remote data, nor can Interactive SQL be used for fast query access to remote files, or for remote file maintenance.[2]

DDM files can be built on SQL tables (that is, a DDM file can reference a remote SQL table), and various CL commands and AS/400 products can use these DDM files. This means that although SQL cannot be used to access DDM files, it can still be used to create all tables on the remote system.

The following is a discussion of the compatibilities and incompatibilities of the SQL and DDM products, and indicates how to circumvent the SQL limitations. No attempt has been made to provide a method of sending SQL statements to a remote system, having them executed and returning the results, like an APPC environment, since this would require considerable extra programming effort at each location. This is not in line with the concept of DDM, where file location is transparent to the programmer, who works as though all files are local to his system. Therefore, emphasis is placed on the best method for replacing an SQL feature with another readily available AS/400 solution.

Interactive Access to Remote Tables

SQL is a very useful tool for fast interactive access to tables and files, for reviewing records, and for quick file maintenance (for example, adding, deleting and updating records). The following tools are also available on the AS/400 system to perform the same functions.

---

1 It is an IBM statement of direction to support distributed relational data access through SQL, so that programmers will not need to know where data resides. SQL/DS already implements "remote unit-of-work" access to remote data, and "distributed unit-of-work" support has also been announced. Since SQL is an SAA product, SQL/400 will also implement these accesses. For further details see IBM Announcement Letter 288-545 "Distributed Relational Data in Systems Application Architecture", dated 881004.

2 Any attempt to access remote files via SQL on a DDM file will fail, with a message indicating that the local file is not a database file. A DDM file is considered to be a device file.

© Copyright IBM Corp. 1990

Product or Command	DDM Support
DFU/400	Supports inserting, updating and deleting records in SQL tables or DDS files.
AS/400 Query	No
SEU	No
SDA	No
DFU/38	Can access remote *keyed* files. Build a logical keyed file on an SQL table if required.
Query/38	Supports queries for DDM files based on SQL tables and DDS files.
OPNQRYF	Supports DDM files (SQL or DDS); all remote files/tables must be on the same system and either an AS/400 or a System/38; if systems are not either two AS/400s or two System/38's then the "group by" clause cannot be used.
AS/400 PC Support	Supports DDM files on both SQL tables and DDS files on remote AS/400 systems and System/38's.

Table 6-1. SQL and System Utilities/Products

From the above table, the following recommendations can be made regarding alternatives to Interactive SQL (queries, updating, deleting, inserting):[3]

- *For Quick File Maintenance*

  DFU/400 provides the file maintenance features that are available through Interactive SQL, such as inserting data into tables, updating and deleting rows and generally viewing particular records, and handles local and DDM files with no limitations. This is a very fast tool which does everything but creation of tables/files. DFU works with one record at a time, whereas SQL allows a set of records to be updated or deleted with one statement.

- *Report Generation*

  AS/400 PC Support uses SQL-like prompts to create a query that transfers data to a personal computer from an AS/400, or vice-versa. Since you can transfer data from the system to a PC, but direct the result to an AS/400 or PC printer instead of a PC file, this is perhaps the easiest method of quickly producing a report on a remote SQL table. Since AS/400 PC Support uses the SQL style and criteria for row and column selection, as well as ordering of rows, this will be an easy task for an SQL user.[4]

  Query/38 can also be used to quickly generate reports based on remote SQL tables, if the product is installed. OPNQRYF can also be used to select and order rows/columns from a remote SQL table, prior to execution of an HLL program, which *does not* use SQL statements.

- *Display-Station Passthrough*

  It is possible to use Interactive SQL on remote tables through the display station passthrough (DSPT) function of OS/400. This enables you to sign on to a remote system, and then work on it as though you are a local user. Therefore, once you are signed on to the remote system, you can use all

[3] Further details on other considerations, and restrictions regarding the type of remote system (for example, non-AS/400 and non-System/38) can be found in Chapter 2 of the *DDM User's Guide*.

[4] See discussion of AS/400 PC Support in "AS/400 Query Tools" on page 1-1 for more details.

SQL statements to work on the remote tables. Since SQL provides the functions of DFU *and* Query in one product this can provide a one-stop solution for remote table activity.

CL Commands on a Remote SQL Table via DDM

A DDM file can be created to reference an SQL table on a remote system, or a remote logical file built on an SQL table on the same remote system. A logical file cannot be built locally on a remote SQL table. CL commands (for example, CPYF, ALCOBJ, DLCOBJ and DSPFFD) can be used on these DDM files to reference the remote table.

Programming Considerations

DDM files should only be used for occasional random remote file access, but not for processing entire remote files, since performance may not be satisfactory. Therefore, if a whole file must be processed, other options should be considered.

Read-Only Remote File/Table Access

If it is necessary to process the majority or all records of a remote file along with some local files, but for read-only access, then the best solution is to copy the data onto the local system prior to processing. This can be accomplished by using a CPYF CRTFILE(*YES) command on a local DDM file pointing to the remote table, and then deleting the created file at the end of processing. Since no updating is taking place on that file, there is no need to lock, or work on, the actual data on the remote system, nor send results back.

If several files are involved on a read-only basis from each system, then consideration can be given as to which system should actually run the program, that is, to which system files should be copied considering how many records are involved.

Remote File/Table Updating Required

In cases where local files and remote files must be processed and updated together, the following options are available:

- *SQL Access*

  In order to use SQL processing, the remote files must be copied to the local system for processing, and then copied back to the remote system at the end of processing. To ensure that users cannot access the file at the remote location in the meantime, the table should be locked to all users using an ALCOBJ (*EXCL) command. The CPYF command can then be used to copy the data from a DDM file based on the table to a local file, and then after processing, from the local file back to the DDM file. The table can then be deallocated DLCOBJ (*EXCL).

  Locking a remote table may be suitable for after-hours batch processing, but less so for daytime processing when other users may require access to a table. In situations where the locking of a table for the length of time necessary for it to be copied to another system, processed, and returned makes this option an unsatisfactory solution, HLL processing of the files should be considered.

- *HLL Processing*

  Local and remote files/tables can be processed identically using HLL statements and DDM files over the remote files/tables. No SQL statements can be used to access the remote files. All HLL file declarations, read, write and update statements can be used with no indication required as to the location of any files within the programs. This method requires no changes from the regular HLL processing that users have used up until now.

HLL processing of files is likely to be the most appropriate method for handling remote file update. It does not require the locking of remote tables, nor any special programming statements. However, if many accesses to remote files must be made, performance may be improved considerably by copying the data to the local system, whether or not SQL processing is used.

Limitations and Recommendations

- *Commitment Control*
  - Commitment control cannot be implemented on a DDM file, whether the file is built on an SQL table or a DDS-defined file.
  - DDM files cannot be based on remote files containing more than one record format, or join-logical files.
- *Migration Recommendation*

  Users moving from the System/38 and System/36 to the AS/400 system may have implemented DDM file processing already, but will not have used SQL. In such cases, they should keep their applications as is. They will not lose any function.

7. SQL Commitment Control

Commitment control applies to programs that execute SQL data manipulation language statements. There are three levels of commitment control under which SQL/400 can run. These are *NONE, *CHG and *ALL. These values are specified by the COMMIT keyword parameter on the STRTSQL command when starting Interactive SQL, and on the CRTSQLxxx command used to compile RPG, COBOL, C/400, FORTRAN and PL/I programs. These levels are a little different from the isolation levels supported by SQL/DS and DB2 and will be compared later in this section.

1. *NONE specifies that commitment control is not being used. COMMIT and ROLLBACK statements are not allowed in the application program. If SQL data definition language statements are run by the application (or in the Interactive SQL session), *NONE must be specified. Each SQL statement is finalized as it is completed, and any changes to the database are immediately executed with no option to rollback. There is no "unit of recovery" concept with *NONE.
2. *CHG specifies that all rows that were updated, inserted or deleted since the last unit of recovery are locked until the transaction is committed or rolled back. A COMMIT statement must be issued to make the changes to the database permanent and to release the locks. Rows that were read for update, but were not updated, will only be locked while the cursor is positioned on the row.
3. *ALL specifies that all rows that were read, updated, inserted or deleted since the last unit of recovery are locked until the transaction is committed or rolled back, or the job ends. A COMMIT statement must be issued to make the changes to the database permanent. Even if no changes were made, all the rows that were read will have locks on them and will require a commit (or rollback) to release them.

   An application program that was compiled under *ALL, or an Interactive SQL statement running under *ALL, which reads an entire table or tables and ends without doing a commit or rollback would effectively have that whole table locked until signoff or until a commit or rollback was performed.

Default Values

It is very important to note that the default for the COMMIT keyword used in RPG, COBOL, C/400, FORTRAN and PL/I compilations is *CHG, which means that these programs are compiled for, and will automatically run under commitment control. You must therefore code COMMIT or ROLLBACK statements in your program. If you do not commit, your locks will *not* be released when the application program ends, with performance implications for other users who might be waiting for rows that you have locked. Also, when your interactive session or batch job ends, the database changes will be removed by an implicit rollback.

© Copyright IBM Corp. 1990

Interactive Implications under Commitment Control

If your program ends without performing a commit or rollback, you can then enter a COMMIT command from the CL command line and the transaction would be ended and your changes would become permanent in the database. However, if you do not either commit in the program or commit from the command line and then you sign off, an implicit rollback will be done and all your changes since the last commit will be lost.

Batch Implications under Commitment Control

Batch programs must have either a COMMIT statement in the program or a COMMIT command in the CL program that calls the application program if the changes are to be made to the database. If no commit is performed then at the end of the batch job stream a rollback will automatically be done and the changes will be removed. A COMMIT statement with HOLD is useful in batch programs in order to provide greater concurrency to other users by releasing locks as the program progresses through the tables.

Cursor Stability and *CHG

SQL/400 does not support isolation-level cursor stability (a level of commitment control used in SQL/DS and DB2), but *CHG provides a similar effect but with the following difference. A user operating under level *CHG, can read rows that have been updated by other concurrent users but not yet committed, whereas a DB2 or SQL/DS user with isolation-level cursor stability will not see updates to rows that have been made by another concurrent user and not yet committed.

Neither *CHG nor cursor stability allow another concurrent user to update rows that have been updated by your current unit of recovery.

Repeatable Read and *ALL

SQL/400 does not support isolation-level repeatable read (a level of commitment control used in SQL/DS, DB2 and OS/2 Extended Edition). Specifying *ALL on the commitment control parameter is similar to isolation-level repeatable read on DB2 or SQL/DS, but not exactly the same.

Both *ALL and repeatable read keep locks on rows (or pages in DB2) that have been read, updated, inserted and deleted until the next commit, rollback or end of job. Repeatable read also guarantees that the set of records that you have selected, cannot be changed, either by an insert, update or delete by any other application during your unit of recovery, and that if you re-issue the same SELECT statement within the same unit of recovery, you will see the same result set. This is achieved by access path locking and adjacent key locking on SQL/DS and DB2. SQL/400 locks the individual rows within a set, but cannot prevent the *set* from being added to by an insertion by another user. SQL/400 will allow another application to insert rows into the set that you have selected, with the result that if you re-execute the SELECT statement you could see new records included in the result.

If User A performs the following selection under *ALL or *CHG:

```
SELECT PARTNO, DESCRIPTION FROM INVENTORY
WHERE PARTNO BETWEEN 207 AND 231
```

Result set
207 GEAR
209 BOLT
221 BOLT
222 BOLT
231 NUT

User B can insert a new row into this set between two existing part numbers:

```
INSERT INTO INVENTORY VALUES (210,'GEAR',25)
```

If user A re-executes the same selection the result set would be different:

Result set
207 GEAR
209 BOLT
210 GEAR
221 BOLT
222 BOLT
231 NUT

The only way to prevent another user from accessing the set you have selected is to exclusively lock the table for the duration of your job, with the ALCOBJ command and lock level *EXCL or SQL LOCK TABLE statement.

Row Locking Under the Various Commitment Control Options

Different levels of concurrency (that is users sharing the same data) can be achieved through the use of the different commitment control parameters (*NONE *CHG and *ALL). Table 7-1 on page 7-4 shows two users accessing the same SQL table and the type of accesses each is allowed. User A either reads or updates/deletes a set of rows, or inserts some rows into the table. User B also tries to read, update/delete or insert into the same set of rows before User A issues an explicit COMMIT statement.

It is important to distinguish between a table and a set within the table. While whole tables may be locked at different levels (for example, AS/400 system level lock "exclusive" (*EXCL), or "shared for update" (*SHRUP)), commitment control locking works on *sets* of records within the table.

Table 7-1. Lock Levels and Activities Permitted

USER A		USER B			
LOCK LEVEL	**ACTION**	**LOCK LEVEL**	**ACTION**		
			Select	**Update, Delete**	**Insert**
***ALL**	Select row(s) from table	*ALL	U	U	M
		*CHG	Y	U	M
		*NONE	Y	U	Y
	Delete/update row(s) in table	*ALL	M[2]	M[5]	M[4]
		*CHG	Y[1]	M[5]	M[4]
		*NONE	Y[1]	M[5]	Y
	Insert into table	*ALL	M[3]	M[3]	Y
		*CHG	Y[1]	M[3]	Y
		*NONE	Y[1]	M[3]	Y
***CHG**	Select row(s) from table	*ALL	Y	Y	Y
		*CHG	Y	Y	Y
		*NONE	Y	Y	Y
	Delete/update row(s) in table	*ALL	M[2]	M[5]	M[4]
		*CHG	Y[1]	M[5]	M[4]
		*NONE	Y[1]	M[5]	Y
	Insert into table	*ALL	M[3]	M[3]	Y
		*CHG	Y[1]	M[3]	Y
		*NONE	Y[1]	M[3]	Y

Table 7-1. Lock Levels and Activities Permitted. Activities allowed for User B on the same set of records, given User A's lock level and actions. Y = Yes, allowed; M = May result in negative return code - see notes; U = Unlikely, probable conflict.

Examine this example where user A could use the following command:

```
UPDATE INVENTORY
SET ONHAND = 100
WHERE DESCRIPTION = 'NUT'
```

at the same time as user B was using this command:

```
UPDATE INVENTORY
SET ONHAND = 200
WHERE DESCRIPTION = 'BOLT'
```

and at the same time, user C could be using these commands:

[1] User B will see uncommitted changes and insertions made by User A. If User A then did a rollback, User B may still see the original changes unless he re-executes the select.

[2] This is only a problem (depending upon timing and WAITRCD value) in the case of a delete, if user B reads a row first, then user A tries to delete it.

[3] Would only prevent the records inserted by User A from being available to User B if they met the SELECT statement criteria.

[4] Would only prevent the records inserted by User B from being available to User A if they met the SELECT statement criteria.

[5] Would only occur in case of delete if both users tried to delete the same row at the same time, which is unlikely.

```
INSERT INTO INVENTORY VALUES (251,'NUT',25)
INSERT INTO INVENTORY VALUES (252,'BOLT',50)
```

As can be seen from the table, *ALL provides less concurrency than *CHG. It is generally recommended that application programs use the default value *CHG.

If the cursor is considered update capable (see section "Notes Regarding the FOR UPDATE OF Clause" on page 4-6) rows are locked as they are read. Under all levels of commitment control including *NONE, if the cursor is considered update capable, the row to which the cursor is currently pointing is locked against all other updates from the *same* program, except for an UPDATE WHERE CURRENT OF CURSOR. If the program is executing under *NONE the rows are always released after the next fetch. If the program is executing under *CHG and and a row is not updated, its lock is released as the next row is fetched. If the program is executing under *ALL, the rows that have been read are kept locked against another job.

If the cursor is considered read only (see section defining "Notes Regarding the FOR UPDATE OF Clause" on page 4-6) no locks are put on the rows as they are read to prevent updates from the same program. However, if the program is executing under commitment control level *ALL the rows will be locked against updates from another job. If the program is executing under *CHG or *NONE no locks will kept locked against updates from another job.

Another example is provided here to explain Table 7-1 on page 7-4. Assume that both User A and User B are issuing the following DELETE statement at the same time:

```
DELETE FROM INVENTORY WHERE DESCRIPTION = 'BOLT'
```

This is unlikely to return a negative SQL return code due to a conflict. For any given row that is a bolt, either User A's DELETE statement will delete the row before User B gets to it or vice versa. Even if they both attempt to delete the same row at exactly the same time, only one will get the lock on the row and delete it, then the other user will not find the row anymore so he will just go on to see if he can find any more rows that have a description of 'BOLT'. There is a slim possibility that a negative SQLCODE could be returned, but this would only happen if one user locked the row, then timesliced out and did not get a timeslice again until after another user timed out waiting for a lock on the same row. This would only occur because the first user secured a lock on the row, but before it could be deleted, he would be timesliced out.

Examine another example using update. Assume both users are running exactly the same application as follows:

```
...
UPDATE inventory SET ONHAND = 200 WHERE DESCRIPTION = 'BOLT'
...
COMMIT
```

Again, if you look at the Table 7-1, you see an M which indicates that User B may get a negative SQLCODE. Timing becomes a major factor in determining what will happen. Assume the sequence is as follows:

```
1. User A locks row 10
2.                        User B attempts to lock row 10 and
                          since User A has a lock on it he waits
3. User A completes the
   update of row 10
4. User A issues the
   COMMIT
5.                        User B now gets the lock on row
                          10, and completes his update
6.                        User B now issues the COMMIT
```

Note that no error occurred. User B waited for a period of time, but because User A completed his transaction before the record lock timeout occurred to User B, every step completed satisfactorily. The record wait timeout default on the system is 60 seconds so most of the time no timeout occurs. The user may change this timeout value to another value for each file.

COMMIT and ROLLBACK with HOLD Option

In SQL/400, the COMMIT and ROLLBACK statements have an optional HOLD parameter. If COMMIT is specified without the HOLD parameter, all changes made during the current unit of recovery are made permanent in the database, all open cursors are closed, all table and row locks obtained by this unit of recovery are released and all prepared statements are discarded.

If COMMIT HOLD is specified, the changes to the database are made permanent and all row locks are discarded but the cursors are not closed, nor are table locks released, nor are prepared statements discarded. The cursor remains pointing to the current row.

If ROLLBACK is specified *without* the HOLD parameter, all open cursors are closed, all the changes that were done in this unit of recovery are rolled back, all table and row locks are released and all prepared statements are discarded.

If ROLLBACK HOLD is specified, the changes made to the database during this unit of recovery are rolled back and row locks are discarded but the cursors are not closed, nor are table locks discarded, nor are prepared statements discarded. The cursor is moved back to point to the same row that it was pointing to when the rolled back unit of recovery was started.

Note that the HOLD versions of these commands are not available for the CL or HLL versions of this command.

Here is a sample RPG program showing the use of COMMIT HOLD and ROLLBACK HOLD. This program is used after stocktaking to interactively update the quantity on hand for each part in the inventory file. The parts that are to be updated are selected in batches by description and are displayed on the screen five at a time. After the quantities are altered on the screen, the database is updated with the five new values and committed with HOLD so that the five locks are released and the cursor points to the next row.

This program would be compiled with commitment control level *ALL:

```
Screen 1      Stocktake Update Quantity on Hand

    Enter description  ____________________



 F3=Exit
```

```
Screen 2      Part Number        Quantity on Hand

                 221                  25
                 222                  15
                 300                 125
                 324                 300
                 328                 195
```

```
H
FPROG5   CF  E                       WORKSTN
```

Arrays for storing the parts as they are read five at a time and for displaying on the screen:

```
E                   ARP        5  3 0        PART NUMBER ARRAY
E                   ARH        5  4 0        ON HAND QTY ARRAY
```

Screen 1 is displayed asking for the description of the group of parts that is to be selected for update. For example, BOLTS or NUTS is keyed and this value is moved into host variable :DESCR.

```
C                    EXFMTSCREEN1
C           *INKC    DOWNE'1'
C                    Z-ADD0       ARH
C                    Z-ADD0       ARP
```

All the parts that match the description keyed on screen 1 are selected:

```
C/EXEC SQL
C+ DECLARE C1 CURSOR FOR
C+ SELECT PARTNO, DESCR, ONHAND
C+ FROM INVENTORY
C+ WHERE DESCR = :DESCR
C+ ORDER BY PARTNO
C/END-EXEC
```

You must *not* include a FOR UPDATE OF clause here because the updating of the table will not be done using UPDATE....WHERE CURRENT OF, which updates each row after it has been read. Instead the program is going to read five rows before displaying them and then they will all be updated together.

```
C/EXEC SQL
C+ OPEN C1
C/END-EXEC
```

Five rows are read from the selected set and stored in the screen arrays. At the end of the fetch loop, the cursor will be pointing to the fifth row unless the end of the cursor has been reached first, in which case the program will go to END1.

Because there is no FOR UPDATE OF clause, this is considered to be a read only cursor and will allow an UPDATE WHERE = host-variable in the same program.

If the program is compiled with *CHG, the locks that are acquired as each row is fetched are immediately discarded and thus it will allow the rows to be updated by another user (or another cursor within the same job) while they are displayed. If the program is compiled with *ALL the read locks acquired at each fetch will be retained and will not allow another user to update the displayed rows before the changes are committed.[6]

```
C           READ      TAG

C           1         DO   5          X       10
C/EXEC SQL WHENEVER NOT FOUND GO TO END1
C/END-EXEC
C/EXEC SQL WHENEVER SQLERROR GOTO END2
C/EXEC SQL
C+ FETCH C1 INTO :PARTNO, :ONHAND
C/END-EXEC
C                     MOVEAPARTNO    ARP,X
C                     MOVEAONHAND    ARH,X
C                     END
```

If the end of the cursor has been reached SQLCOD will be 100 and indicator 99 is set on.

```
C           END1      TAG
C           SQLCOD    COMP 100                     99
```

The five rows are moved to the screen and screen 2 is displayed so that the on-hand quantity for each part can be updated.

```
C                     EXFMTSCREEN2
```

If any SQL error, for example, ROW IN USE (SQLCODE -913) or NO RECORD FOUND (SQLCODE 100) is encountered during the update, the program will go to END2 where a rollback is done.

```
C/EXEC SQL  WHENEVER NOT FOUND GOTO END2
C/END-EXEC
```

The updated fields are read from the screen and the program loops around five times updating the previous four rows and the current row of the INVENTORY table. This type of update does not affect the cursor position, nor does it use the cursor to point to the rows to be updated. Instead the rows are located by a direct read to the row using the part number as the key.

[6] Or another cursor within the same job. If *CHG the other cursor may read and update; if *ALL the other cursor may read and update all rows except the row currently in use by the first cursor and the cursor is update capable. This would be the case if there were an UPDATE or DELETE WHERE CURRENT OF in the program, or a FOR UPDATE OF clause or a dynamic statement in the program.

```
C           1         DO   5        X       10
C                     MOVE ARP,X    PARTNO
C                     MOVE ARH,X    ONHAND
C           PARTNO    IFNE 0
C/EXEC SQL
C+ UPDATE INVENTORY
C+ SET ONHAND = :ONHAND
C+ WHERE PARTNO = :PARTNO
C/END-EXEC
C                     END
C                     END
```

If no SQLERROR occurred on the update then the changes are committed with the HOLD option so that the cursor remains open and pointing to the next row to be processed, and the previous five locks are released.

```
C/EXEC SQL
C+ COMMIT HOLD
C/END-EXEC
```

If there are more rows to be processed the program goes back to the point where the next row is fetched from the cursor.

```
C  N99                GOTO READ
C                     SETOF                    99
```

Otherwise the cursor is closed and the first screen is displayed again so that another set of inventory items can be processed.

```
C/EXEC SQL
C+ CLOSE C1
C/END-EXEC
C                     EXFMTSCREEN1
C                     END
```

End of main "do loop" of program when F3 is pressed.

```
C                     SETON                    LR
C                     GOTO END
```

Changes made during the last unit of recovery will be rolled back if any error occurred, and the program will fetch the rows again and display them on the screen.

```
C           END2      TAG
C/EXEC SQL
C+ ROLLBACK HOLD
C/END-EXEC
C                     GOTO READ
C           END       TAG
```

If the program had been displaying only one row at a time on the screen, the SELECT statement would have been coded with a FOR UPDATE OF clause. An update lock would have been taken at fetch time and the UPDATE WHERE CURRENT OF statement would have been used. No other user would be able to update the row while it was displayed on the screen. In this case the program should do a COMMIT HOLD periodically after a certain number of updates in order to release the locks.

Rollback Considerations

Many users up until now have never implemented journaling or commitment control for execution of HLL programs, even without SQL table access. Since some applications may implement SQL and HLL file access in the same program, it is necessary to consider all the implications of this implicitly invoked recovery feature. This section summarizes SQL and system commitment control facilities and highlights how they should be used together. It also explains some of the restrictions of SQL commitment control.

HLL Use of Commitment Control

This section discusses the HLL versus SQL accessing of files and tables and the implications of each on commitment control. Since the system provides two types of commitment control, it is often not clear as to which one should be used, or whether it is necessary to invoke both. Operating with commitment control is the default with SQL. SQL tables can be accessed from within HLL programs, either by regular file processing commands (for example, reads and writes) or through SQL statements. If the program uses SQL statements, SQL commitment control will be implemented as indicated at program creation (that is, as specified on the CRTSQLxxx command with *ALL, *CHG or *NONE).

For a non-SQL processing program, commitment control will be implemented according to the STRCMTCTL command issued before program execution. In both cases, in order to actually implement COMMIT *ALL or *CHG, COMMIT and ROLLBACK commands must be used either in, or after, the HLL program. There are three types of COMMIT and ROLLBACK statements: SQL COMMIT and ROLLBACK, from either the Interactive SQL command line, or embedded in an SQL HLL program; HLL commit and rollback (for example, RPG COMIT and ROLBK, or COBOL COMMIT and ROLLBACK), entered from within the HLL program; CL COMMIT and ROLLBACK, entered wherever a CL command can be entered (for example, the Command Entry screen, or a CL program). There is no difference between each of the three types; they all commit or rollback *all* pending row changes. The SQL COMMIT with HOLD statement also has some other considerations as discussed in "COMMIT and ROLLBACK with HOLD Option" on page 7-6.

Non-SQL Table Processing within an HLL Program

COBOL programs must have an I-O-CONTROL paragraph in the Environment Division, naming the tables upon which commitment control is required. RPG programs must have F-continuation specifications with KCOMIT in columns 53 to 58 for every file which will use commitment control. A COMMIT/ROLLBACK statement must be either in the program itself, or following the CALL statement in the calling program; otherwise the changes will only be temporary.

Non-SQL processing programs do not specify commitment control on the CRTXXXPGM command, but must specify a commitment control level on the STRCMTCTL command.[7] Commitment control must be started using the STRCMTCTL command, prior to program execution; otherwise the program will not execute if it contains commitment control statements. It can be entered interactively, or through a calling CL program. All files on which commitment

---

[7] The rules governing *locks* acquired on tables in these cases are consistent with the table in "SQL Commitment Control" on page 7-1.

control is to be used must already be journaled to the same journal at this time. At the end of the program, commitment control is still active, until the ENDCMTCTL command is executed. This would cause a following STRCMTCTL command to fail. If a STRCMTCTL command is issued once commitment control is active, or an ENDCMTCTL command issued when commitment control has already been terminated, an error will be received and a CL program will fail if this has not been taken into account (MONMSG).

SQL Table Processing within an HLL Program

If only SQL access to a table is required in a program, then in RPG no F- or I-specifications are required for the table, and in COBOL no FD statements are necessary for table definition. Also, no commitment control statements of the type I-O-CONTROL (COBOL) are necessary to name the files on which commitment control is required. SQL will automatically invoke the commitment control function when the first table access is made, provided the program was compiled with commitment control level *CHG or *ALL. In this case there are two options for implementing the actual commit or rollback:

- A COMMIT or ROLLBACK statement can be issued within the program, to finalize all actions immediately.
- A COMMIT or ROLLBACK statement can be issued after program execution, either interactively, or through a calling CL program.

Whether the COMMIT/ROLLBACK statement is in the program or following it, commitment control is still active after termination of the program until explicitly stopped using the ENDCMTCTL command. It should be noted that since SQL commitment control is automatically invoked as soon as an SQL statement on a table is received, a STRCMTCTL command is not necessary. However, it will not cause an error if used prior to SQL table access. Since the STRCMTCTL command is implicitly invoked and the ENDCMTCTL command is not, the user must be careful not to leave records locked unintentionally, or to cause a following STRCMTCTL command to fail, by omitting the ENDCMTCTL command.

HLL or SQL Table Processing Summary

To summarize, for programs only implementing SQL table accesses, the STRCMTCTL command is not required. But in both cases the COMMIT/ROLLBACK command must be implemented either in the HLL program or following it, and before an ENDCMTCTL command. If an ENDCMTCTL command is processed and changes are pending COMMIT, they will be rolled back.

The following CL program would implement commitment control on tables accessed through HLL programs with or without SQL:

```
PGM
STRCMTCTL LCKLVL(*ALL/*CHG/*NONE)*
CALL program-name
COMMIT/ROLLBACK (if not included in program)
ENDCMTCTL
ENDPGM
```

* Optional when table access is through SQL. A MONMSG CL statement could be used to ensure that if commitment control were *already* active, the program would not end abnormally.

Note: Journaling must be active before the program is invoked.

SQL and Non-SQL Table Access in the Same HLL Program

It is possible to access an SQL table using SQL statements *and* HLL statements (for example, reads and writes) in the same program. In such cases, it is very important to consider the SQL commitment control implications, since SQL commitment control is invoked implicitly while HLL commitment control must be explicitly programmed (that is, through the I-O-CONTROL statement (COBOL) or the KCOMIT statement (RPG)).

Consider the case of an HLL program which accesses an SQL table through HLL file processing commands *and* SQL statements, and then ends abnormally. Will all the changes made before the ending be rolled back, or committed? The answer lies in the level of commitment control specified on the CRTSQLxxx command, and whether the program explicitly invokes the system commitment control function.

1. *CRTSQLxxx *NONE and no HLL commitment control*

   All changes made to the table will be kept.

2. *CRTSQLxxx *CHG or *ALL and no HLL commitment control*

   All changes made to the table through SQL statements will be rolled back, while all HLL file changes will be permanent.

3. *CRTSQLxxx *CHG or *ALL and HLL commitment control*

   All changes made to the table will be rolled back, regardless of how they were made unless COMMIT was specified immediately after the program ends.

4. *CRTSQLxxx *NONE and HLL commitment control*

   All changes made to the table through HLL processing will be rolled back unless COMMIT is specified immediately after the program ends, while all SQL changes will be permanent.

Options 2 and 4 are not normally desirable, since they leave the table in a corrupt state, with some changes having been rolled back, and others not. Options 1 and 3 are both satisfactory; but it is important to remember to specify *NONE on the CRTSQLxxx command if no commitment control is desired. Just omitting commitment control from the HLL program will not provide for no commitment control, and Option 2 will be the result.

Therefore, it is recommended that if SQL and HLL table accesses are to be used concurrently in a program, HLL commitment control should be invoked explicitly in the program through the I-O-CONTROL statement (COBOL) or KCOMIT statement (RPG). This ensures that *all* changes made to a table are committed or rolled back together.

Commitment Control on Tables/Journals in Different Collections

In order to implement the three commitment control modes it offers, SQL automatically creates a journal and journal receiver in each collection. All activity on tables in each collection is by default automatically recorded here. SQL statements from within an HLL program cause journal entries to be recorded, and if the program is created with commitment control *CHG or *ALL,

the entries can be used to rollback or commit the changes during or after program execution.

If you are using tables from different collections in the same HLL program, SQL commitment control requires that the tables be journaled into the same journal. However, with the journaling as set up by SQL, journal entries for each table are written to the QSQJRN journal in its own collection. Imagine an accounts receivable application, which on an occasional basis has to access a general ledger table in a different collection, and which uses SQL processing. By default, the application will be created with commitment control *CHG, and the tables will be journaled to different collections. The application will now fail because it does not meet the SQL requirement of having all tables accessed in a program being journaled to the same collection.

This requirement is the same as that of the system commitment control function, which also requires that all files under commitment control within a job be journaled to the same journal. Therefore, it is not possible to use the SQL automatic commitment control modes *CHG and *ALL in jobs which use SQL statements but run on tables with journals in different collections, without first making some changes to the journaling as set up by SQL. You must either remove the commitment control feature by ending the journaling, or continue to use it by moving from the default SQL journal to a user-created journal.

Ending Journaling

To end the SQL journaling, the ENDJRNPF command must be executed. This will end journaling of the tables named in the FILE parameter.

```
ENDJRNPF FILE(table1 table2...) JRN(QSQJRN) LIB(collection)
```

SQL Journaling in Different Collections

Sometimes it is necessary to keep tables and journals for the same job in different collections, such as in the previous example where an accounts receivable application accesses a general ledger table stored in a different collection. In such cases, steps must be taken prior to execution of the application, to detach the tables from their SQL journals, and attach them to another user-created journal. First you must find or create a journal and journal receiver in a separate library. Journaling must then be ended (ENDJRNPF as indicated in the previous section) on all affected tables, and restarted using the STRJRNPF command.

```
STRJRNPF FILE(table1 table2...) JRN(journal) LIB(collection)
```

Once the STRJRNPF command has been issued, any accesses to the tables are recorded. The SQL commitment control modes *ALL, *CHG and *NONE are now available as usual, along with the automatic commit and rollback features.

To summarize, if a program accesses two tables, for example, table1 in collection1 and table2 in collection2, and they are being journaled to different collections (as is the SQL default) the following CL program could be used to "rearrange" the journaling prior to, and after, execution of the program. Note the use of object allocation to ensure integrity.

```
ENDJRNAP (table1)
ALCOBJ OBJ((table1 *FILE *EXCL))
ENDJRNPF FILE(table1) JRN(QSQJRN) LIB(collection1)
STRJRNPF FILE(table1) JRN(QSQJRN) LIB(collection2)
STRJRNAP (table1)
DLCOBJ OBJ(table1)
CALL program
ENDJRNAP (table1)
ALCOBJ OBJ((table1 *FILE *EXCL))
ENDJRNPF FILE(table1) JRN(QSQJRN) LIB(collection2)
STRJRNPF FILE(table1) JRN(QSQJRN) LIB(collection1)
STRJRNAP (table1)
DLCOBJ OBJ(table1)
```

"Read Only" Access of Tables in Different Collections

If the program is created with commitment control *CHG or *ALL, the first table access will be permitted, but all subsequent accesses to other tables will not be accepted. A negative SQL code will be returned indicating that the cursor for that table cannot be opened due to invalid journaling, since all tables/files under commitment control in the same job must be journaled to the same journal. As usual, the program will continue to execute, unless a check is made of the SQLCODE.

Commitment Control Considerations for New Tables

Since SQL only automatically journals tables created by the CREATE TABLE statement, there are other journaling considerations when using the CPYF, CRTDUPOBJ, RSTOBJ and MOVOBJ commands to create tables. See the section on tables and files in "SQL Implementation Techniques" on page 3-32 for further details.

8. SQL Security

SQL provides an application interface to AS/400 security for granting or revoking access to SQL tables and views, and controls the creation of indexes. The SQL security on the AS/400 system is a function that is mainly available for compatibility with other SQL products, as the AS/400 system has built-in system security facilities which provide a more extensive function. On the AS/400 system, the SQL GRANT and REVOKE statements use the GRTOBJAUT and RVKOBJAUT system control language commands to actually effect the change in security for an object. SQL GRANT and REVOKE only provide a subset of the function available when using GRTOBJAUT and RVKOBJAUT. The underlying changes in security for an object made with SQL GRANT and REVOKE may be seen by the DSPOBJAUT or EDTOBJAUT commands.

As the GRANT and REVOKE statements are designed only to be used in an SQL environment, only security for tables and views in a collection are able to be updated. Objects in a non-collection library (and objects which are not tables, views or physical files in a collection) cannot be secured via these statements.

Note that SQL system catalog tables are not able to be secured via SQL GRANT and REVOKE. The system control language commands must be used for this.

In the literature discussing SQL naming conventions, "Authorization ID" is a term commonly used. This is most relevant when discussing security, as this refers to the user profile of the person to whom authority is given (or from whom authority is revoked).

In standard AS/400 logical file design, a logical file may be created containing a projection of the fields in a physical file. With this, a logical file may be created with a subset of fields from the original physical file. Privileges may be granted to the physical and logical file such that the user only has access to this subset of fields (via the logical file) and no other access to the related physical file. In order to grant a user authority to the subset of fields, object operational rights must *not* be granted to the physical file and *must* be granted to the logical file.

The SQL/400 GRANT and REVOKE statements do not return data. As such, they can be easily executed dynamically in any programming language on the AS/400 system that supports SQL.

Default Security Levels

Table 8-1 on page 8-2 contains a summary of the various SQL objects, and their default security values after creation by a user. In all cases public authority is *EXCLUDE and authorization list is *NONE. These should be considered when granting or revoking other authorities.

© Copyright IBM Corp. 1990

Table 8-1. Default Security Values For SQL Created Objects

Object	Object Authority for owner
Collection	*ALL
Journal	*ALL
Journal Receiver	*ALL
Dictionary Objects (LFs)	*OBJOPR, *OBJMGT, *OBJEXIST
Dictionary Objects (PFs)	*ALL
Catalog LFs and Views	*OBJOPR, *OBJMGT, *OBJEXIST
Data Dictionary	*ALL
User Tables	*ALL
User Views	*OBJOPR, *OBJMGT, *OBJEXIST
User Indexes	*OBJOPR, *OBJMGT, *OBJEXIST

In the above table, note that the object operational authority is granted for certain objects without any corresponding data rights for the object. Note that if the user is a member of a group profile, then the owner of the object optionally may be the group profile, and all other authorities specified above are given to the group profile.

Changing Authorities

In this section, we will take as an example a user who did not create the original collection and objects contained therein. Hence, on a level 30 secured system, such a user would have no default authority to access any of the objects in the collection. This is a likely situation in a production environment. Without considering adopted authority, the following points outline items that should be set into place in establishing the production environment.

Authority to the Collection

The user must have *USE authority (*OBJOPR authority with *READ rights) for the library containing the tables, views and indexes. In addition to the following points, this will provide access to all functions required except for the creation of indexes (see "Authority for Creating Objects in a Collection" on page 8-6), tables and views. Note that if the user is a member of a group profile, the group profile can have the required *USE authority. Alternately, the default *PUBLIC authority can be changed to *USE for the library (as long as the user does not have *EXCLUDE rights granted to him), or an authorization list be created for the library and associated users.

SELECT, INSERT, UPDATE and DELETE

Granting specific authority for each of these functions for a user will give the user object operational rights and the individual data rights to a table, of *READ, *ADD, *UPD and *DLT, respectively. For example, if a user required authority to perform a SELECT statement on a table, the command:

```
GRANT SELECT ON TABLE table-name TO user
```

would change the object authorities for the table for the user from:

```
----Object-----  ----------Data----------
Opr  Mgt  Exist  Read  Add  Update  Delete
              -  none  -
```

to

```
----Object-----  ----------Data----------
Opr  Mgt  Exist  Read  Add  Update  Delete
 X                X
```

Then adding authority to update the table with the statement:

```
GRANT UPDATE ON TABLE table-name TO user
```

would change the authorities for the user to:

```
----Object-----  ----------Data----------
Opr  Mgt  Exist  Read  Add  Update  Delete
 X                X           X
```

and similarly for the other data rights and associated functions.

Views

Data rights for select, insert, update and delete of a view are stored with the table and not the view. If a user is given the data right to read from a table then he will automatically have the same authorities for all of the defined views if the views have object operational rights for that user. (Similarly, if *all privileges* are revoked for a view, then *all privileges* are revoked for a base table also).

For example, assume you have a table which has authorities granted for user USER for performing selection and update; and also a view which the user is not authorized to (does not have object operational rights) at this stage.

```
        ----Object-----  ----------Data----------
        Opr  Mgt  Exist  Read  Add  Update  Delete
Table:   X                X           X
View :                -  none  -
```

Then if you grant authority for the user to select, update, and delete from that view:

```
GRANT SELECT, UPDATE, DELETE ON view-name TO user
```

the authorities are now changed to

```
       ----Object-----  ----------Data-----------
       Opr  Mgt  Exist  Read  Add  Update  Delete
Table:  X               X          X       X
View :  X
```

At first, this may be difficult to follow. Data rights for a view are assigned by providing only the object operational right. Individual data rights are changed on the base table itself.

If you now choose to revoke SELECT statement authority for the view only:

```
REVOKE SELECT ON view-name FROM user
```

the authorities are now changed to

```
       ----Object-----  ----------Data-----------
       Opr  Mgt  Exist  Read  Add  Update  Delete
Table:  X                          X       X
View :  X
```

The implication of this is that the user can now no longer perform a SELECT statement on either the table or view, when all you wanted was to revoke the authority for selection from the view.

The key message from this point is that *authorities for a table and its defined views cannot conflict.*

Granting/Revoking ALL Authority

Granting all privileges (or ALL) for a table will provide the following system authorities:

```
       ----Object-----  ----------Data-----------
       Opr  Mgt  Exist  Read  Add  Update  Delete
        X    X           X     X    X       X
```

In this sense, the user has authority to perform SELECT, INSERT, UPDATE, DELETE, and CREATE INDEX statements.

Note: the authorities are correct at the table level for the creation of indexes. See "Authority for Creating Objects in a Collection" on page 8-6 for the other two authorities required before this can be done.

This is different from giving a user *ALL authority for an object (say a table) as a system security function. The system control language command:

```
GRTOBJAUT library/table-name *FILE user *ALL
```

would give all of the above authorities as well as object existence rights.

Granting all rights for a view (assuming there is no authority to access the table and the view and assuming the view is SELECT, INSERT, UPDATE and DELETE capable) will provide the following system authorities:

```
        ----Object-----  ----------Data-----------
        Opr  Mgt  Exist  Read  Add  Update  Delete
Table:                    X     X     X       X
View :   X
```

Views that are read-only would only grant *READ to the table. Views that do not allow inserts would not grant *ADD rights to the table.[1]

Note that indexes cannot be created over a view.

Delegation of Authority: Another point of possible confusion occurs with revoking ALL authorities. Consider three users, QSECOFR, ADMIN, and USER. Suppose that originally, QSECOFR has all system authorities. He gives authority to ADMIN to administer the security for other SQL users and ADMIN always grants authority with SQL GRANT statements. ADMIN then grants authority to USER to access certain tables and views, again using the SQL GRANT statement.

Now, say ADMIN has changed jobs and no longer needs to use the SQL databases, so QSECOFR revokes all privileges from ADMIN. Be aware that the privileges assigned to USER by ADMIN will still be in existence. Revoking authority from ADMIN will *not* cascade and revoke authorities that ADMIN has assigned. This is a function of other SQL products but is not supported on the AS/400 system due to the granularity of the security functions.

Authority for Selecting on System Tables

As briefly mentioned before, GRANT and REVOKE statements cannot be used for changing authority for a user to perform SELECT statements on system catalog tables such as SYSCOLUMNS, SYSTABLES, SYSINDEXES, and SYSVIEWS.

If an end user (or a programmer otherwise authorized like an end user) requires extra authority to perform these functions, then this will need to be done outside SQL with the GRTOBJAUT and RVKOBJAUT system control language commands. However, this is a little more complex than it at first seems. Not only does the user need object operational access to the logical file named SYSCOLUMNS (for example), authority will also have to be granted to the underlying physical files that reside in the collection library that are part of the catalog facility.[2] Each of these files will require at least *READ authority.

As this is so complex, you should consider securing these objects via an authorization list for each collection. Any additional users can be simply added to the authorization list rather than changing the authority for each object.

---

[1] To insert into a view, at least all of the columns that are in the original table, but not in the view must be defined as NOT NULL WITH DEFAULT, so that data not inserted will default to a value based on the data type (blanks for character columns and zeros for numeric columns).

[2] These files can be seen when displaying the objects in the library. There are two groups. Some are identified by QIDCTxnn, where the xnn will vary for each of the different objects, x is a character, and nn two digits. There are 17 of these. Two other files are QSQCOLUMNS and QSQTABLES. Authority for all 19 files will need to be changed.

Authority for Creating Objects in a Collection

In granting authority for a user to create an index, table or view, some objects need to have privileges over and above those mentioned previously. The following items should be changed, so that the user has at least:

- *OBJOPR and *ADD authority to the *DTADCT object. This has the same name as the library.
- *OBJOPR and *ADD authority on the collection library.

  Note: this only needs to be *USE for all other operations.

Authority for Creating an Index

To create an index, a user must have object operational and object management rights to the table upon which the index will be created.

This statement would then allow a user to create an index on the table:

```
GRANT INDEX ON table-name TO user
```

This would change the authority for the table only, by adding the *OBJMGT authority. For example, a user which has previously been granted SELECT statement privileges would have the following authorities:

```
----Object-----  ----------Data-----------
Opr  Mgt  Exist  Read  Add  Update  Delete
 X                X
```

Now, adding the authority to create indexes, it would look like this:

```
----Object-----  ----------Data-----------
Opr  Mgt  Exist  Read  Add  Update  Delete
 X    X           X
```

Security Recommendations

In order to simplify the granting and revoking of authorities to the many SQL objects discussed in the preceding section, the use of authorization lists is recommended. A user can be given access to many objects quickly by his addition to an authorization list.

The number of authorization lists required will depend upon the way in which security will be implemented on the system. Assuming a level 30 secured system, there are two cases: using SQL GRANT and REVOKE for access to tables and views; or using the system control language commands for changing the authority to the tables and views. Each of the cases is discussed below.

Using SQL GRANT and REVOKE

SQL GRANT and REVOKE may be used in an environment where an application has been migrated from another system which supports SQL (for instance DB2). This can also be used where security requirements are very dynamic and an end user program is required to perform the authorization functions easily.

In either of these cases, only a small number of authorization lists would be required to allow quick changes to the objects which cannot be authorized with GRANT and REVOKE. These would include an authorization for:

- Each collection library, where users are given *USE authority:

  One for each library, or

  One for groups of libraries, or

  One for all libraries.

- Each of the 19 system catalog objects as well as the catalog tables: SYSCOLUMNS, SYSINDEXES, SYSKEYS, SYSTABLES, SYSVIEWDEP, SYSVIEWS for each collection. Users should be given *USE authority.

- Collection libraries where tables, views and indexes might be created or dropped. Users should be given *CHANGE authority. This may be used in place of the first authorization list for libraries (or in conjunction with it for selected users only). Also, this list could be combined with authorization for the data dictionary (*DTADCT) object (which has the same name as the library). This requires a minimum authority of *USE, but if combined with the other functions needed for object creation and deletion, this could automatically be *CHANGE due to the structure of the authorization list mechanism.

Using System Security for All Authorities

This would be most likely in the majority of sites where security is relatively static or intersystem compatibility of SQL security is not as important. In this instance, all of the above listed authorization lists would be required in addition to further authorization lists to replace in part the SQL GRANT and REVOKE statements. These extra authorization lists may include authorizing all tables, views and indexes at the same level, one list for each application.

You may consider separate authorization lists for different complexities of user, or department of user. These users would need to be members of group profiles where only the group is assigned to the list. Note that an object may be authorized by only one authorization list.

Authorization lists should not be used in any further detail than this as maintenance of the list would tend to be the same or more effort than granting or revoking individual object authorities.

Interactive SQL and Security

Care should be taken in granting authority to use SQL in the interactive environment for reasons of performance, security or data integrity. Interactive SQL is designed as a programmer tool. If an end-user is authorized to use Interactive SQL, then he is able to execute any SQL DML statement against a table to which he is authorized. For instance, in a normal application environment, an end-user will be given authority to read, insert, update and

delete records in the order master file in an on-line transaction processing environment. However, the on-line order entry program, *limits* the extent to which the user perform these read, insert, update, and delete operations through the controlling logic of the order entry program itself. Using Interactive SQL, there is no controlling logic, and the user is free to perform any of these functions that he wishes.

Interactive SQL will allow a user to enter a complex select statement which may tie up large quantities of system resource. The user may do this inadvertently or intentionally, but the end result is degraded response times for other users of the system.

Use the Edit Object Authority (EDTOBJAUT) command (or appropriate GRTOBJAUT and RVKOBJAUT command parameters) to change the default authority for the STRSQL command.

Program Adoption of Authority

In many application environments, using program adoption of authority is the simplest and most straightforward method for the control of access to data. In this environment, production applications could be set up once, and any new users given access to the initial program they require. All other rights would automatically be available provided the environment had been set up for this. This is a suitable approach in implementing application security with or without SQL.

Another aspect which is worth considering is in a large development environment where different levels of programmers require certain levels of access to the resources. In this case, then the authorization list approach offers the most flexibility.

Commitment Control

Be aware that no SQL data definition statements can be executed when commitment control is active (*CHG or *ALL). Two of these statements are SQL GRANT and REVOKE. It is important that COMMIT(*NONE) is specified for an interactive session, or COMMIT(*NONE) is specified for program compilation when SQL GRANT or REVOKE statements are to be executed. When performing SQL precompilation, the COMMIT parameter defaults to *CHG.

9. Interactive SQL

SQL consists of three main parts:

1. SQL parser, prompter (and SEU syntax checker) and run-time support
2. SQL precompilers
3. SQL interactive interface.

Parser, prompter and run-time support are included in the base OS/400. The precompilers and interactive interface are included in the SQL/400 product. This section will discuss the major aspects of Interactive SQL. Detailed information may be obtained from chapter 10 of the *Programming: Structured Query Language Programmer's Guide (SC21-9609)*.

Starting Interactive SQL

Interactive SQL can be started by a menu option or by the Start SQL Session command (STRSQL). There are two parameters of STRSQL which influence the SQL session.

- The first is the NAMING parameter, which decides whether the system or SQL naming convention will be used in the session. The SQL naming convention (*SQL) uses the "collection.table" concept. The system naming convention (*SYS) uses the "library/file" concept. This is the default. This cannot be modified from within the session.
- The REFRESH parameter tells the system when to refresh the data shown; either, only when forward paging, or always: *ALWAYS is the default. This can be modified from within the session.

Interactive SQL Session

After starting Interactive SQL, the statement entry screen is presented. Since the command entry screen was changed in release 3.0, STRSQL no longer resembles the command screen. Rather it resembles the command entry screen as it was prior to release 3.0. Most of the techniques for working with the release 2.0 command entry screen also apply to the SQL statement entry screen, for example, retrieve (F9), editing statements, scrolling through the session log, and so on. The SQL statements which can be entered in an Interactive SQL session are listed in Chapter 10 of the *Programming: Structured Query Language Programmer's Guide (SC21-9609)*. These statements can be processed in three ways:

- Check for correct syntax only
- Syntax check, and perform a validity check on used objects
- Syntax check, validity check and run the statement

Statements can be entered directly if the syntax, and the clauses are known. Otherwise, the statements can be prompted for, by using F4, and then keying in the appropriate information required.

© Copyright IBM Corp. 1990

All statements entered during a session can be saved into a source file, where they may be further edited for inclusion into a program. Statements may be saved at any time during the session by using F13, and choosing the appropriate option.

However, be aware that these saved statements **cannot** be recalled from the source file, to be used in an Interactive session.

On leaving the Interactive session, if Option 1 is taken, then the session work will be saved. If STRSQL is used again, then this saved session can be restored, and work can be continued in the session, in interactive mode.

HELP Support

Contextual HELP is available throughout the Interactive SQL session. The HELP text displayed depends on the screen content and cursor position. More information can be obtained about an error message by positioning the cursor on the message and pressing the HELP key. Prompting is available for SQL statements (use F4), but to obtain more background information (about Interactive SQL and SQL in application programs) use the following procedure:

- Press the HELP key on any screen of Interactive SQL
- Press F2 for extended HELP if required.
- Press F11 for search index
- Press F5 for all topics.

Session Services

The session service function can be called by pressing F13 on the statement entry screen. The following functions are available:

- Change commitment control
- Change statement processing control
- Change SELECT statement output device
- Change list of libraries
- Change list type (system- or SQL-created objects)
- Print current session
- Remove all entries from current session
- Save session in source file
- Change Data Refresh Options.

The device which will receive the results of a SELECT statement can be a display, printer or database file. The file created by this function can then be used as a normal database file, and can be input to the SQL session, if required. This can be very useful for creating and manipulating test data.

Prompting and List Functions

Prompting can be used to help enter data into the SQL statements (F4). The type of SQL statement to be used can be chosen in this way. When this has been done, prompting can be continued to indicate which information is required by SQL in order to process the statement.

Use of the Function keys on the statement entry screen will also present information to help fill in what is required. Library list (F16) allows you to list the libraries/SQL collections from which SQL can access objects. File list (F17) shows you all the files (tables) that the current SQL session has access to. Field list (F18) presents all the fields (columns) of the selected files. One or more entries from the file list and the field list may be selected to be copied to the statement entry screen at the current cursor position.

Prompting Within SEU

When editing a program that contains embedded SQL statements, you may prompt for the SQL statements. For a *new* SQL statement, you need to create the *SQL statement environment* before prompting. To do this, use the SEU editor to insert the following (for an RPG/400 program):

```
C/EXEC SQL
C+
C/END-EXEC
```

You can then press the F4 key or type P over the sequence number to prompt for the statement. If you are prompting for an SQL statement that has already been entered, simply press the F4 key or type P over the sequence number against any line of the statement.

Where to Use Interactive SQL and SQL Prompting

Interactive SQL is very powerful and if not properly controlled it can easily lead to unwanted data manipulation. It is not aimed directly at the End User, although statement prompting is available. Even so, Interactive SQL is a very useful tool for programmers and database administrators. The following lists its recommended uses:

- Data base administrators may use Interactive SQL to maintain access privileges to collections, tables or views.
- Data base administrators can use Interactive SQL to select information from system catalog tables.
- Programmers may use Interactive SQL to create and maintain test data.
- The main purpose of Interactive SQL is to test SQL statements before including them into an HLL program.
- Prompt in SEU for proper syntax of SQL statements. Full list support is also available.

10. SQL Standards

SQL/400 complies with the following proposed standards. Deviations and omissions are documented in the following paragraphs.

- *ISO Standard Database Language SQL* (ISO 9075-1987)
- *ISO Standard Database Language SQL* (ISO 9075-1989)

  This is a replacement standard for ISO 9075-1987. This standard contains all the features of the previous standard plus the integrity enhancement features.
- *ANS Database Language SQL* (ANSI X3.135-1986)
- *ANS Database Language SQL* (ANSI X3.135-1-1989)

  This is a replacement standard for ANSI X3.135-1986. This standard contains all the features of the previous standard plus the integrity enhancement features.
- *ANS Database Language SQL* (ANSI X3.168-1989)

  This includes the embedded languages from the appendixes of the previous standards and defines C and Ada support.
- *ISO and ANS Working Draft Database Language SQL2*

  This is a draft of the next version of the ANS and ISO standards. It is now out for public review.
- *Federal Information Standards SQL* (FIPS 127)
- *Federal Information Standards SQL* (FIPS 127.1)

  This is a replacement of the previous FIPS standard that includes the ANS X3.135-1-1989 and ANS X3.168-1989 standards and sets minimum limits for various items.
- *SAA Common Programming Interface Database Reference* (SC26-4348-1).

ISO 9075-1989 and ANS X3.135-1-1989

SQL/400 claims SQL-DML conformance to level 1 of the ANS/ISO SQL standard. The following facilities substantiate that claim:

- Direct processing of SQL data manipulation language statements
- Embedded SQL COBOL
- Embedded SQL PL/1
- Embedded SQL C
- Embedded SQL FORTRAN.

SQL/400 differences from Level 1

1. The set of reserved words specified in the standard are not all reserved.

   This is not required for compliance except in FIPS. FIPS requires flagging support that would flag all the reserved words in the standard.

© Copyright IBM Corp. 1990

2. PUBLIC

   PUBLIC is supported, but if private rights have been granted to a user, the PUBLIC rights do not apply to that user.

SQL/400 omissions of Level 1

1. DISTINCT (that is AVG(DISTINCT column-name)) is not supported for COUNT, AVG, SUM, MIN, or MAX functions.
2. Ada style comments (for example -- comment) are not supported.
3. UPDATE privileges on a column are not supported.
4. A module processor is not provided.

   This is not required for compliance since SQL/400 supports embedded SQL.
5. Indicator variables of a type other than small integer.

   The standards allow the implementation to choose its data type for indicator variables as long as it is an exact numeric type with zero scale. Since the data types supported are restricted, this means that in:

   - COBOL, DISPLAY SIGN LEADING SEPARATE must be used
   - FORTRAN, INTEGER must be used
   - PL/I, DECIMAL must be used

   Note that SQL2 (this is the next version of the standards) is relaxing this so that in COBOL and PL/I, SQL/400 will comply without any change. The same is not yet true in FORTRAN, but should be before the standard is published.
6. View composition restrictions.

   - The select list must not include column functions if R is derived from a view whose outer subselect includes DISTINCT. Furthermore, if R is derived from a view whose outer subselect includes DISTINCT, the select list must identify all columns of the view (possibly by SELECT *) and must not include DISTINCT or expressions
   - The select list must not include column functions if R is derived from a view whose outer subselect includes GROUP BY or HAVING clauses.
   - If more than one table or view name is specified in a FROM clause, a view whose outer subselect includes DISTINCT must not be identified.
   - If more than one table or view name is specified in a FROM clause, a view whose outer subselect includes GROUP BY or HAVING must not be identified.

SQL/400 differences from Level 2

1. None.

SQL/400 omissions of Level 2

1. A SCHEMA processor is not provided.

   This support is not required for compliance to the ANS or ISO standards. It is required for FIPS.
2. Escape character in the LIKE predicate is not supported.

3. UNIQUE support on a column definition is not supported.
4. NULL and full null value support is not provided.
5. USER is not allowed in the subselect of a CREATE VIEW statement.
6. The optional INDICATOR specification for host variables is not supported.
7. The WITH CHECK OPTION is not supported.
8. The WITH GRANT OPTION is not supported.
9. Repeatable read is not supported.

   The standard in 4.16 requires that "all read operations are reproducible within a transaction". This can not quite be achieved by the COMMIT(*ALL) option. Reproducible implies that the result of a SELECT statement must be the same (except for changes the transaction itself makes) every time it is executed in the transaction. COMMIT(*ALL) prevents any other jobs from deleting or updating rows that have been read by a transaction, but it does not prevent inserts or updates to other rows from occurring that could now show up if the SELECT statement was executed again. For example, User A executes a SELECT statement with a WHERE col1 = 1. User B inserts a row where col1 = 1. If User A executes the same SELECT statement in the same transaction again, he will now get a result that contains the original row and the new inserted row. In order to comply with the reproducible requirement, User B would have to be prevented from inserting his record until User A issued a commit or rollback.

   This omission should not affect compliance. Note that the standard never mentions the term "repeatable read". SQL/400 can meet the "reproducibility" requirement by having the user request that all the tables in the SELECT statement be locked *EXCLRD or *SHRNUP (depending on whether the cursor is read only or not).

ANS X3.135-1-1989 Integrity Enhancement

SQL/400 differences

1. None.

SQL/400 omissions

1. CHECK clause is not supported.
2. User specified defaults are not supported.
3. Referential integrity is not supported.
4. NOT NULL can be repeated and not order dependent.

   For example, CREATE TABLE x.x (col1 CHAR(10) NOT NULL NOT NULL) is allowed in ANS.

ANS X3.168-1989 Embedded SQL

SQL/400 differences

1. Integer is COMP instead of COMP-4.

   This should not affect compliance. Note that it is also inconsistent with the ANS COBOL standard. The ANS COBOL standard says that COMP is implementation-defined (on OS/400, COMP is packed decimal and COMP-4 is integer). Note that in SQL2, this has been fixed, and BINARY is used for integer instead of COMP. Since SQL/400 already supports BINARY, no change is necessary.

SQL/400 omissions

1. COBOL DISPLAY SIGN LEADING SEPARATE is not supported.
2. COBOL host variable names starting with a digit are not supported.

   This is really a restriction in the COBOL/400 compiler.
3. C host variables with const and volatile attributes are not supported.
4. Embedded SQL in Pascal is not supported.

   This should not affect compliance. Note that only one language is necessary to comply. SQL/400 supports both COBOL, C, FORTRAN and PL/I.
5. Embedded SQL in Ada is not supported.

   This should not affect compliance. Note that only one language is necessary to comply. SQL/400 supports both COBOL, C, FORTRAN and PL/I.

FIPS 127.1 Compliance

FIPS SQL is based on ANSI X3.135-1-1989 and on ANSI X3.168-1989 so all above omissions and differences apply. FIPS requires full SQL conformance to level 2 DDL and DML and requires support of either the Module Language or Embedded SQL interface to one or more FIPS programming languages (PL/I is not a FIPS Programming Language). The integrity option is not required. In addition, the following omission applies:

- Flagging of language levels is not supported.

FIPS 127.1 specifies a default set of minimum limits that a product must meet unless the procurement specifies different limits. The following default limits can not be satisfied by SQL/400:

- Identifiers must support up to 18 characters.

  SQL/400 supports 18 character identifiers for cursors and statements, but not for columns, tables, views, index names, and correlation names.

SAA Common Programming Interface Database Reference

The *SAA CPI Database Reference (SC26-4348-1)* defines the standard SQL language for IBM. The following lists the major omissions SQL/400 has with SAA. For detailed information on how SQL/400 conforms to this definition, see the *SAA CPI Database Reference* . Differences and omissions are printed in green ink in the reference.

The major omissions are:

1. Null value support
2. Variable length field support

   Variable length host variables are supported, but SQL/400 does not support creating a table with a variable length column.
3. Date/Time Support
4. Commit/Rollback of Data Definition Statements

   You can not commit or rollback data definition statements such as CREATE TABLE. If a data definition statement is embedded in a program or issued interactively, COMMIT(*NONE) must have been specified.
5. DISTINCT in a column function

   DISTINCT in the SELECT clause is supported (that is, SELECT DISTINCT x). DISTINCT is not supported in a column function (for example, MAX(DISTINCT x) is not supported).
6. ALTER TABLE
7. GRAPHIC and VARGRAPHIC
8. COMP-2 in COBOL

   Note that this is actually a restriction of COBOL/400.
9. Lower case characters are not allowed in delimited column names.

11. SQL/400 Portability

In this section you will be considering various aspects concerning the portability of SQL/400 between the AS/400 system and the other SAA environments that support the SQL language, namely DB2 under MVS*, SQL/DS under VM* and OS/2 EE Database Manager. SAA is IBM's announced set of standards for consistency between these four environments. For complete details refer to *SAA Common Programming Interface Database Reference*. However, SAA SQL is not yet fully implemented in all the environments.

Data Definition Language portability and Data Manipulation Language portability are examined separately below.

Data Definition Language

Detailed below are the differences in SQL data definition language functions between SQL/400 and SAA.

SAA Size Limits

SAA has specified maximum sizes for certain named items which may be less than those allowed in specific environments. In order to be portable, applications should not exceed the SAA maximum sizes.

The major differences are highlighted below. The full list of limits appears in the *SAA CPI Database Reference*.

	SAA	SQL/400
Longest AUTHID	8	10
Row length (bytes)	4005	32766
Number of rows per table	16777215	16777215
Most columns in a table	255	8000
Most columns in a view	140	8000
Largest decimal value	15 digits	31 digits

Table 11-1. Size Differences in Commonly Used SQL Items

SAA Functions Not Implemented in SQL/400

Certain SAA SQL functions are not implemented in release 3.0 of SQL/400. They are:

1. TABLE CREATE functions:
    - NULL Values - you can only specify NOT NULL or NOT NULL WITH DEFAULT
    - VARCHAR
    - GRAPHIC and VARGRAPHIC
    - DATE, TIME and TIMESTAMP
    - Delimited column names do not allow lower case characters.
2. ALTER TABLE

© Copyright IBM Corp. 1990

3. The set of reserved words specified in the standard are not all reserved in SQL/400. New words may be added to the SQL/400 list as future releases add new functions to the product.

Data definition language statements cannot be run under commitment control. In other words if your application was compiled with *CHG or *ALL or has COMMIT and ROLLBACK statements in it, it cannot also contain CREATE, COMMENT ON, DROP, GRANT, LABEL ON, or REVOKE statements.

System Catalogs

The names of the SQL/400 catalogs are:

- SYSCOLUMNS
- SYSINDEXES
- SYSKEYS
- SYSTABLES
- SYSVIEWDEP
- SYSVIEWS.

This is similar to a subset of the DB2 implementation but other implementations of SQL have different and additional catalogs.

SQL/400 Data Definition Extensions

SQL/400 supports an additional data type of NUMERIC on the CREATE TABLE statement.

SQL/400 supports collections through the CREATE COLLECTION and DROP COLLECTION statements. Tables, views, and indexes must be created in collections. Any tables or views referenced in SQL data definition language statements must exist in a collection.

Authorization Control Differences

There are several items to be considered:

1. The *SYS option on the CRTSQLxxx makes programs run under OS/400 security *USER instead of SQL security with the *OWNER attribute (as well as enabling system naming convention instead of SQL naming convention). If portability is required, this should be avoided.
2. View privileges are different. Data rights (INSERT, UPDATE, DELETE and SELECT statement privileges) are stored with the table, not the view. If a user loses his data rights to the table, he also loses his rights to the view.
3. SQL/400 does not remember dependent privilege descriptors. (For example, if User A grants User B a privilege, and then User A loses the privilege, User B does *not* lose the privilege automatically on SQL/400.)
4. Public privileges are supported but if private rights are granted to a user, the public rights do not apply.
5. Update privileges on a column are not supported. A user who has update privileges on a particular table cannot be excluded from updating individual columns of that table.

Other Considerations

Other authorization considerations that have to be made include:

1. Do not rely on the use of library lists as this is not supported in the other environments.
2. Consider the establishment of a database creation program to use DDL data streams that may be ported from other systems. Such a program could also be used to re-execute SQL data streams stored in source physical files during Interactive SQL sessions.

Data Migration

Data can be moved from one system to another via sequential files. Each system (DB2, SQL/DS, SQL/400 and OS/2 Database Manager) has its own utility program with which to unload its database files into sequential files on tape. Once it is in sequential format, it can be transported to the other system and loaded into the predefined database files. To unload data from DB2, use the DB2 utilities or DXT (Data Extract) program. Use the Data Bases Service Utility (DBSU) to unload from SQL/DS. From the system, use the copy file (CPYF) or copy-to-diskette/copy-to-tape (CPYTODKT/CPYTOTAP) commands to copy the files to tape or diskette, and CPYF or CPYFRMTAP/CPYFRMDKT to unload the data.

Data Conversion Considerations

The collating sequence is different between ASCII and EBCDIC, therefore you will get different results from an ORDER BY clause run on the PS/2* and the same ORDER BY clause run on the AS/400 system.

The internal formats for numeric data differ in the various environments. The following areas could cause difficulties when migrating data between databases in different environments:

- Delimited ASCII leading zero
- Decimal errors handled differently
- Floating point arithmetic has different format in different environments.
- NULL value data cannot be stored in an SQL/400 table.

Summary

If you wish to migrate table definitions and data, you need to keep things simple. Follow the SAA guidelines for name and item sizes, and use data types CHAR, INTEGER, SMALLINT and DECIMAL (odd precision) and NOT NULLs.

Data Manipulation Language

See the section "SQL Standards" on page 10-1 for details of SQL/400 extensions and limitations that should be avoided if portability is important.

Locking Rules

The automatic row locking rules are different in SQL/400 from the other products. For example, DB2 locks by TABLESPACE and PAGE with escalation to TABLESPACE if ANY is specified. SQL/DS locks by DBSPACE, PAGE or ROW with escalation to DBSPACE if too many rows are locked. SQL/400 locks by ROW up to a maximum of 32768 rows.

Isolation Levels

There are three levels of commitment control under which SQL/400 can run. They are *NONE, *CHG and *ALL. These levels are a little different from the isolation levels CURSOR STABILITY and REPEATABLE READ supported by SQL/DS and DB2. See the section "SQL Commitment Control" on page 7-1 for further details.

COBOL/400 and SQL Portability

Although COBOL and SQL are SAA-standard products, COBOL/400 with SQL does not yet completely match SAA COBOL or the other COBOL/SQL products. Batch programs are already quite portable, but interactive programs with data presentation (AS/400 workstation files) are less so. Each environment has its own mapping methods, and these cannot be transported to other systems. However, in the future, the situation will become easier, with full implementation of the Presentation Manager and Dialog Manager in all the SAA environments. The following are some of the points to remember if you want to make your applications portable among other SAA-standard systems with different architectures.

Data Type Equivalence

Only the following data types are supported in SQL/400. Their COBOL/400 equivalent, with restrictions are indicated:

SQL Data Type	COBOL/400 Equivalent
CHAR	01 field PIC X(n)
NUMERIC	01 field PIC S9(n)[2]
INTEGER	01 field PIC S9(9) COMP-4
SMALLINT	01 field PIC S9(4) COMP-4
DECIMAL	01 field PIC S9(n)V9(d) COMP-3[3]
FLOAT	01 filler PIC 9(10) COMP-2[1]

Table 11-2. Data Type Equivalences

[1] COBOL/400 does not support COMP-2.

[2] This is not supported in SAA SQL.

Using COPY-DDS or SQL INCLUDE

COBOL/400 SQL allows file/table definitions to be copied into programs in file declaration statements, or working storage sections. The statement COPY-DDS-ALL-FORMATS of filename or tablename will retrieve the field descriptions, and create them into a host data structure within the program. However, this feature is not available in other COBOL SQL products.

The SQL INCLUDE statement retrieves SQL statements from separate source files, so that file descriptions, for example, could be stored separately from programs and copied in. However, whereas the COPY-DDS statement retrieves the original file description and requires no extra work, the SQL INCLUDE statement requires code to be physically placed into these separate source files.

COBOL using SQL/DS or DB2 accepts the SQL INCLUDE statement. Therefore, if portability is important to you, COPY-DDS statements should be replaced by SQL INCLUDE statements, and separate source modules with file/table descriptions created. Other COBOL products may also require table *definition* in addition to the creation of a host data structure as previously mentioned. For example, COBOL using SQL/DS, DB2 or OS/2 EE also requires an SQL DECLARE statement to describe each table and view that the program accesses.

GOBACK Statement

The GOBACK statement is an optional COBOL/400 statement which performs the same functions as the COBOL/400 STOP RUN and EXIT commands. However, in COBOL using SQL/DS and DB2, the GOBACK statement *must* be coded to end a program. Therefore, for all applications likely to be ported to other systems, the GOBACK statement should be used.

SQL Continuation Characters

With COBOL/400 SQL statements, when continuing a string constant from one line to the next, the first non-blank character on the next line must be a "string delimiter". This can be an apostrophe (') or quotation marks ("). For COBOL using SQL/DS, the character must be an apostrophe ('). If the delimiter identifier is continued from one line to the next, the first non-blank character on the next line must be the SQL escape character. This SQL escape character is either a quotation mark (") or an apostrophe (') as specified on the COBOL/400 precompiler option. In COBOL using SQL/DS this must be a quotation mark (").

For further information on SAA SQL standards see "SQL Standards" on page 10-1 and also the *SAA CPI Database Reference*. For further information on SAA COBOL standards see the *SAA CPI COBOL Reference*.

[3] For COBOL/400 n+d must be maximum 18 but with the S/370, n + d must be maximum 15, and odd.

12. SQL/400 and Relational Theory

SQL/400 was introduced into OS/400 at Release 1 Modification Level 2. Since then, changes have been made and some new functions have been added to the SQL product on the AS/400 system. Currently, there is only partial compliance with the full SAA definition of SQL. Some of the advanced functions are not yet implemented.

This chapter discusses the traditional relational theory and attempts to contrast SQL/400 with the theory as originally defined by Dr E.F. Codd, which has been refined by other relational database exponents.

Codd's Relational Rules

Codd defined a set of rules by which a relational database product could be evaluated. These rules were first introduced in the magazine *ComputerWorld*, in the October 14, 1985 issue. The rules outlined function that was determined to be important in a relational database management system. The rules are briefly described below.

Rule 0 - Foundation Rule: For any system that is advertised as a relational database management system, that system must be able to manage databases entirely through its relational capabilities. If a system does not pass rule 0 it is not worth rating.

This rule does not disallow any non-relational capabilities to exist. It does mean that the system must support insert, update, and delete relational (multiple-records-at-a-time processing) operations and must provide support for rule 1 and at least partial support for rule 2.

Rule 1 - Information Rule: All information in the database including the catalogs should be viewed in a tabular form.

Rule 2 - Guaranteed Access Rule: All data can be accessed by its values, not its physical address or physical placement in the row or table. A primary key is required.

Rule 3 - Systematic Treatment of Nulls: A null value is an unknown value and therefore should not participate in any built-in functions such as sums, averages and counts. This also implies support for testing for NULL (IS NULL predicate).

Rule 4 - Active On-line Catalog: There must be a set of descriptions of the data in the system which is active all the time.

Rule 5 - Comprehensive Data Sublanguage: A relational database may support several interfaces and languages, but one language with well defined syntax must support the following functions: data definition, view definition, data manipulation, integrity constraints, authorization, and transaction boundaries.

© Copyright IBM Corp. 1990

Rule 6 - ***View Updating Rule:*** SQL views should be able to be updated with full transparency. This poses the implication of updating a join view, which is likely to include some complex ambiguities.

Rule 7 - ***High-level Insert, Update, and Delete:*** Data Manipulation functions should operate on sets. This is relatively straightforward for the select operator, but less often considered for UPDATE and DELETE, and INSERT.

Rule 8 - ***Physical Data Independence:*** There should be no visible association between the logical data and the physical construct. Changes in physical characteristics or even implementations such as indexes should not be visible to either an application program or an interactive terminal user. Without this independence new technology will result in either changes to the application program or failure to benefit from the new function.

Rule 9 - ***Logical Independence:*** There should be no impact on existing applications when changes are made to logical views rather than physical tables.

Rule 10 - ***Integrity Independence:*** Primary and foreign keys should conform to the rules of referential integrity. Without this capability, additions of new dependent relationship types may result in reprogramming applications that create or delete participants in this relationship.

Rule 11 - ***Distribution Independence:*** The data must be capable of distribution transparent to any user. Functions like join and union must be able to be performed across tables which are physically located on different systems. This also implies that a view containing a join can be created over two physically separate tables. Codd's rule implies that a data base management system (DBMS) does not necessarily have to have the function to support distributed database. However, the DBMS must be designed whereby future distributed database support can be added without any effect on the data itself.

Rule 12 - ***Non-Subversion:*** If access to database data is allowed through a non-relational interface, the integrity of the database must not be able to be subverted. For example, a UNIQUE index over a table prevents unique values from existing in the table. This constraint must be enforced even thorough non-relational interfaces.

Now that the relational rules have been defined, it is worthwhile to compare how some of the database products conform. The products compared in the following charts are:

- IBM Database 2, Version 2 Release 2
- IBM SQL/DS, Version 3 Release 1
- IBM SQL/400, Release 3 Modification level 0
- IBM OS/2 Database Manager, Release 1.2

Rule	DB2	SQL/DS	SQL/400	OS/2	SAA Standard
1	Yes	Yes	Yes	Yes	Yes
2	Yes	Yes	Partial	Yes	Partial
3	Partial	Partial	No	Partial	Partial
4	Yes	Yes	Yes	Yes	Yes
5	Yes	Yes	Yes	Yes.	Yes
6[1]	Partial	Partial	Partial	Partial	Partial
7	Yes	Yes	Yes	Yes	Yes
8	Yes	Yes	Yes	Yes	Yes
9	Partial	Partial	Partial	Partial	Partial
10	Yes	Yes	No	Yes	No
11	Yes	Yes	Yes	Yes	Yes[2]
12	Yes	Yes	Yes	Yes	Yes

Table 12-1. Comparison of Codd's Relational Rules to IBM SQL products

Although DB2 and SQL/DS do not comply with all of Codd's rules, you can see there are many facilities that are implemented on these systems. Both score nine yeses and three partials. AS/400 SQL scores seven yeses and three partials. SAA standards achieve seven yeses and four partials. DB2 and SQL/DS comply with almost all of the relational database requirements when using Codd's rules as a measure.

The AS/400 system is a full participant in the SAA guidelines. We can therefore reasonably expect that the AS/400 database and SQL/400 will be developed in line with these documented standards over a period of time.

Referential Integrity

Referential integrity is the second of the two relational database integrity rules that are defined in the relational model.[3] Referential integrity introduces the database concepts of *primary* and *foreign keys*. A primary key is a column which uniquely identifies a row. A foreign key is a column in one table that is used to reference a row in an associated table. It does this by matching the foreign key with the primary key of the associated table.

For instance, in a personnel database, the employee table may contain a department number column. This department number is a foreign key, and can be used to associate the employee with the department name held in the department table. For example:

[1] Update capable views is a debated point. IBM versions of SQL are able to update views under certain circumstances. See "Views" on page 3-7 for more information on updating views.

[2] An SAA announcement was made on 4th October 1988 to include the full support of Distributed Relational Data. The details of this announcement are available in IBM Announcement Letter 288-545 entitled: Distributed Relation Data In Systems Application Architecture. A subsequent SAA announcement was made on 26th June 1990 on the availability of the architectures for Distributed Database. The details of this announcement are available in IBM Announcement Letter 290-363.

[3] The first rule is *entity integrity* and states that a primary key in a base table cannot have null values. The AS/400 system implementation of SQL does not support nulls, thus partially complying with this rule, but does not provide primary key syntax.

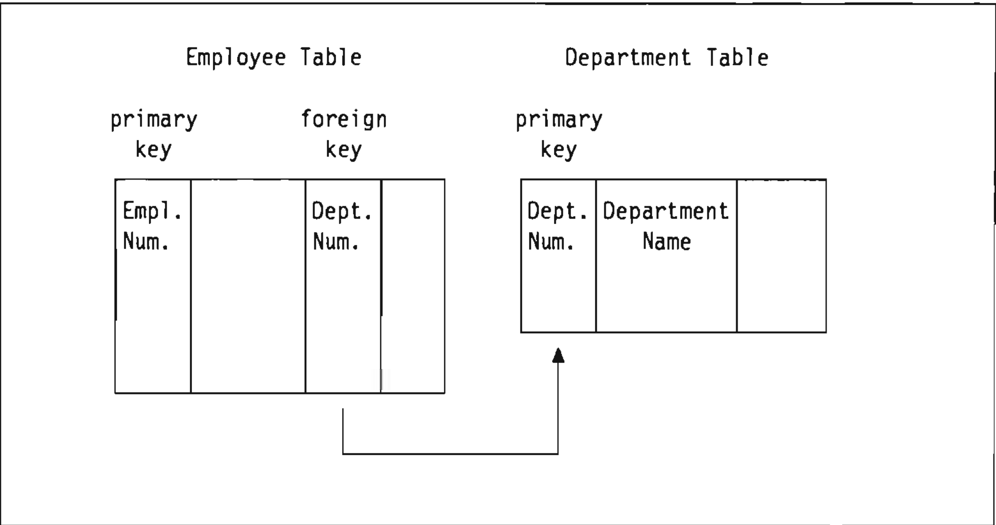


Figure 12-1. Example of a Foreign Key. A Personnel Data Base

Referential integrity therefore is concerned with ensuring that the foreign key in the first table (department number in the employee table) matches a primary key in the related table (department number in department table); or that the foreign key (Dept Num in Employee) contains a null value (hence there is no relationship for that row). In the example above, the department table is known as the parent table and the employee table is known as the dependent table.

This is a relatively straightforward concept in theory, but is often difficult to enforce. Consider, in the example above, when an update is required in both tables. Assume that the department number must change, and all the employees that work in that department must have their department number changed in the employee table. The problem that arises is that normally only one table can be updated in a single SQL statement. We can make the update to the employee table in one SQL statement,[4] being:

```
UPDATE PERSONNEL/EMPLOYEE
SET DEPTNO = 344
WHERE DEPTNO = 333
```

However, as soon as this update is performed, you have violated the referential integrity rule, as the department table still knows that department as 344.

The next SQL statement would bring that into line:

```
UPDATE PERSONNEL/DEPARTMENT
SET DEPTNO = 344
WHERE DEPTNO = 333
```

and the tables would now conform to the referential integrity rule. But in this process, for a short period of time, the rule was violated.

[4] Although you only use one SQL statement, this is actually implemented in the call to the SQL routine as an update performed many times. We would notice this more if there were a failure while performing this one SQL statement. Some rows would be updated, and others not.

The rule is likely to be violated only when UPDATE, INSERT, or DELETE operations are made. As SELECT only retrieves data, there is no need to concern ourselves with referential integrity when reading from a table.

The use of COMMIT and ROLLBACK statements can help overcome the breach of referential integrity. We can start a new transaction before any update is made. We then perform the updates, and if they complete successfully, you COMMIT the changes. If there is a failure during the process, you can ROLLBACK and retry, thus ensuring that the rule is breached only for a very short time.

AS/400 and Referential Integrity

On the AS/400 system there are no facilities to support referential integrity. From the discussion of "Codd's Relational Rules" on page 12-1, you can also see that this is not yet a documented component of SAA.

Appendix A. Code Example For Use of SQL WHENEVER in RPG

```
F*  File declaration for QPRINT
F*
FQPRINT  O   F     132           PRINTER
I*
I* Structure for report 1.
I*
IRPT1      E DSTEMPRACT
I              STARTDATE                         STARDT
I              ENDDATE                           ENDDT
I              EMPTIME                           EMPTIM
I*
I* Structure for report 2.
I*
IRPT2        DS
I                                           1   6 PRJNUM
I                                           7  42 PNAME
I                                        B  43  440EMPCNT
I                                        P  45  4920LDTIM
I                                        P  50  542NEWTIM
I*
I            DS
I                                           1   3 PROJID
I                                        P   4   62PERCNT
C*
C                     Z-ADD.06       PERCNT
C                     MOVEL'MA%'     PROJID
C*
C* Update the selected projects by the new percentage. If
C* errors occur during the update, rollback the changes.
C*
C/EXEC SQL WHENEVER SQLERROR GOTO UPDERR
C/END-EXEC
C*
C/EXEC SQL
C+ UPDATE USER1/TEMPRACT
C+    SET EMPTIME = EMPTIME * (1+:PERCNT)
C+    WHERE PROJNO LIKE :PROJID
C/END-EXEC
C*
C* Commit changes.
C*
C/EXEC SQL COMMIT
C/END-EXEC
C*
C/EXEC SQL WHENEVER SQLERROR GO TO RPTERR
C/END-EXEC
C*
C* Report the updated statistics for each employee assigned to
C* selected projects.
C* Write out the header for report 1.
C*
C                     EXCPTRECA
```

© Copyright IBM Corp. 1990

```
C/EXEC SQL declare c1 cursor for
C+    select * from user1/tempract
C+    where tempract.projno like :projid
C+    order by empno
C/END-EXEC
C*
C/EXEC SQL
C+  OPEN C1
C/END-EXEC
C*
C* Fetch and write the rows to QPRINT.
C*
C/EXEC SQL WHENEVER NOT FOUND GO TO DONE1
C/END-EXEC
C           SQLCOD    DOUNE0
C/EXEC SQL
C+   FETCH C1 INTO :RPT1
C/END-EXEC
C                     EXCPTRECB
C                     END
C           DONE1     TAG
C/EXEC SQL
C+  CLOSE C1
C/END-EXEC
C*
C* For each project selected, generate a report containing the
C* project number, project name, the old total of employee hours,
C* and the new total of employee hours.
C*
C* Write out the header for report 2.
C*
C                     EXCPTRECC
C/EXEC SQL
C+  DECLARE C2 CURSOR FOR
C+    SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),
C+       SUM(EMPTIME/(1.0+:PERCNT)),SUM(EMPTIME)
C+    FROM USER1/TEMPRACT, USER1/TPROJ
C+    WHERE TEMPRACT.PROJNO = TPROJ.PROJNO
C+    GROUP BY TEMPRACT.PROJNO, PRNAME
C+    HAVING TEMPRACT.PROJNO LIKE :PROJID
C+    ORDER BY 1
C/END-EXEC
C*
C/EXEC SQL OPEN C2
C/END-EXEC
C*
C* Fetch and write the rows to QPRINT.
C*
C/EXEC SQL WHENEVER NOT FOUND GO TO DONE2
C/END-EXEC
C           SQLCOD    DOUNE0
C/EXEC SQL
C+   FETCH C2 INTO :RPT2
C/END-EXEC
```

```
C                    EXCPTRECD
C                    END
C           DONE2    TAG
C/EXEC SQL CLOSE C2
C/END-EXEC
C                    GOTO FINISH
C*
C* Error occurred while updating table.  Inform user and rollback
C* changes.
C*
C           UPDERR   TAG
C                    EXCPTRECE
C/EXEC SQL WHENEVER SQLERROR CONTINUE
C/END-EXEC
C*
C/EXEC SQL
C+   ROLLBACK
C/END-EXEC
C                    GOTO FINISH
C*
C* Error occurred while generating reports.  Inform user and exit.
C*
C           RPTERR   TAG
C                    EXCPTRECF
C*
C* All done.
C*
C           FINISH   TAG
C                    SETON                       LR
OQPRINT  E 0201          RECA
O                                  21 'UPDATED EMPLOYEE PROJ'
O                                  37 'ECT ACCOUNT DATA'
O        E 01            RECA
O                                   8 'EMPLOYEE'
O                                  17 'PROJECT'
O                                  26 'ACCOUNT'
O                                  36 'EMPLOYEE'
O        E 02            RECA
O                                   7 'NUMBER'
O                                  16 'NUMBER'
O                                  25 'NUMBER'
O                                  34 'HOURS'
O        E 01            RECB
O                        EMPNO      7
O                        PROJNO    16
O                        ACTNO L   26
O                        EMPTIML   37
O        E 22            RECC
O                                  42 'ACCUMULATED STATISTIC'
O                                  54 'S BY PROJECT'
```

```
0       E 01            RECC
0                                  7 'PROJECT'
0                                 56 'NUMBER OF'
0                                 66 'PREVIOUS'
0                                 76 'CURRENT'
0       E 02            RECC
0                                  6 'NUMBER'
0                                 21 'PROJECT NAME'
0                                 56 'EMPLOYEES'
0                                 64 'HOURS'
0                                 75 'HOURS'
0       E 01            RECD
0                       PRJNUM     6
0                       PNAME     45
0                       EMPCNTL   55
0                       OLDTIML   67
0                       NEWTIML   77
0       E 01            RECE
0                                 28 '*** ERROR Occurred while'
0                                 52 ' updating table. SQLCODE'
0                                 53 '='
0                       SQLCODL   62
0       E 01            RECF
0                                 28 '*** ERROR Occurred while'
0                                 52 ' generating reports. SQL'
0                                 57 'CODE='
0                       SQLCODL   67
```

Appendix B. Code Example For Use of SQL WHENEVER in COBOL

```
IDENTIFICATION DIVISION.

PROGRAM-ID.  CBLEX.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-AS400.
OBJECT-COMPUTER. IBM-AS400.
INPUT-OUTPUT SECTION.

FILE-CONTROL.
    SELECT PRINTFILE ASSIGN TO PRINTER-QPRINT
       ORGANIZATION IS SEQUENTIAL.

DATA DIVISION.

FILE SECTION.

FD  PRINTFILE
    BLOCK CONTAINS 1 RECORDS
    LABEL RECORDS ARE OMITTED.
01  PRINT-RECORD PIC X(132).

WORKING-STORAGE SECTION.
77  PROJID PIC X(3).
77  PERCENTAGE PIC S999V99 COMP-3.

***************************************************************
*  Structure for report 1.                                    *
***************************************************************

01  RPT1.
    COPY DDS-TEMPRACT OF USER1-TEMPRACT.

***************************************************************
*  Structure for report 2.                                    *
***************************************************************

01  RPT2.
    15  PROJNO PIC X(6).
    15  PROJECT-NAME PIC X(36).
    15  EMPLOYEE-COUNT PIC S9(4) COMP-4.
    15  OLD-TOTAL-TIME PIC S9(6)V99 COMP-3.
    15  NEW-TOTAL-TIME PIC S9(6)V99 COMP-3.

    EXEC SQL
         INCLUDE SQLCA
    END-EXEC.
77  CODE-EDIT PIC ---99.
```

© Copyright IBM Corp. 1990

```
*****************************************************************
*  Headers for reports.                                         *
*****************************************************************
 01  RPT1-HEADERS.
     05  RPT1-HEADER1 PIC X(132)
               VALUE "UPDATED EMPLOYEE PROJECT ACCOUNT DATA".
     05  RPT1-HEADER2.
         10  FILLER PIC X(10) VALUE "EMPLOYEE".
         10  FILLER PIC X(9) VALUE "PROJECT".
         10  FILLER PIC X(9) VALUE "ACCOUNT".
         10  FILLER PIC X(104) VALUE "EMPLOYEE".
     05  RPT1-HEADER3.
         10  FILLER PIC X(10) VALUE " NUMBER".
         10  FILLER PIC X(9) VALUE "NUMBER".
         10  FILLER PIC X(9) VALUE "NUMBER".
         10  FILLER PIC X(104) VALUE " HOURS".
 01  RPT2-HEADERS.
     05  RPT2-HEADER1.
         10  FILLER PIC X(21) VALUE SPACES.
         10  FILLER PIC X(111)
                 VALUE "ACCUMULATED STATISTICS BY PROJECT".
     05  RPT2-HEADER2.
         10  FILLER PIC X(9) VALUE "PROJECT".
         10  FILLER PIC X(38) VALUE SPACES.
         10  FILLER PIC X(11) VALUE "NUMBER OF".
         10  FILLER PIC X(10) VALUE "PREVIOUS".
         10  FILLER PIC X(65) VALUE " CURRENT".
     05  RPT2-HEADER3.
         10  FILLER PIC X(9) VALUE "NUMBER".
         10  FILLER PIC X(38) VALUE "PROJECT NAME".
         10  FILLER PIC X(11) VALUE "EMPLOYEES".
         10  FILLER PIC X(10) VALUE " HOURS".
         10  FILLER PIC X(65) VALUE "  HOURS".
 01  RPT1-DATA.
     05  FILLER PIC X VALUE SPACE.
     05  EMPNO PIC X(6).
     05  FILLER PIC XXX VALUE SPACES.
     05  PROJNO PIC X(6).
     05  FILLER PIC X(4) VALUE SPACES.
     05  ACTNO PIC ZZZ99.
     05  FILLER PIC X(3) VALUE SPACES.
     05  EMPTIME PIC ZZZZ9.99.
     05  FILLER PIC X(96) VALUE SPACES.
 01  RPT2-DATA.
     05  PROJNO PIC X(6).
     05  FILLER PIC XXX VALUE SPACES.
     05  PROJECT-NAME PIC X(36).
     05  FILLER PIC X(4) VALUE SPACES.
     05  EMPLOYEE-COUNT PIC ZZZ9.
     05  FILLER PIC X(5) VALUE SPACES.
     05  OLD-TOTAL-TIME PIC ZZZZ9.99.
     05  FILLER PIC XX VALUE SPACES.
     05  NEW-TOTAL-TIME PIC ZZZZ9.99.
     05  FILLER PIC X(56) VALUE SPACES.
```

```
 PROCEDURE DIVISION.
 A000-MAIN.
     MOVE 0.06 TO PERCENTAGE.
     MOVE "MA%" TO PROJID.
     OPEN OUTPUT PRINTFILE.

***************************************************************
* Update the selected projects by the new percentage. If an   *
* error occurs during the update, ROLLBACK the changes,       *
***************************************************************

     EXEC SQL
          WHENEVER SQLERROR GO TO E010-UPDATE-ERROR
     END-EXEC.
     EXEC SQL
          UPDATE USER1/TEMPRACT
            SET EMPTIME = EMPTIME * (1+:PERCENTAGE)
            WHERE PROJNO LIKE :PROJID
     END-EXEC.

***************************************************************
*  Commit changes.                                            *
***************************************************************

     EXEC SQL
          COMMIT
     END-EXEC.

     EXEC SQL
          WHENEVER SQLERROR GO TO E020-REPORT-ERROR
     END-EXEC.

***************************************************************
*  Report the updated statistics for each employee assigned to*
*  the selected projects.                                     *
***************************************************************

***************************************************************
*  Write out the header for Report 1.                         *
***************************************************************

     write print-record from rpt1-header1
           before advancing 2 lines.
     write print-record from rpt1-header2
           before advancing 1 line.
     write print-record from rpt1-header3
           before advancing 2 lines.
     exec sql
          declare c1 cursor for
            select *
            from user1/tempract
            where tempract.projno like :projid
            order by empno
     end-exec.
```

```
     EXEC SQL
          OPEN C1
     END-EXEC.

     PERFORM B000-GENERATE-REPORT1
         THRU B010-GENERATE-REPORT1-EXIT
         UNTIL SQLCODE NOT EQUAL TO ZERO.

 A100-DONE1.
     EXEC SQL
          CLOSE C1
     END-EXEC.

*************************************************************
*  For each project selected, generate a report containing the*
*  project number, project name, the old total of employee    *
*  hours, and the new total of employee hours for each        *
*  project.                                                   *
*************************************************************
*  Write out the header for Report 2.                         *
*************************************************************

     MOVE SPACES TO PRINT-RECORD.
     WRITE PRINT-RECORD BEFORE ADVANCING 2 LINES.
     WRITE PRINT-RECORD FROM RPT2-HEADER1
           BEFORE ADVANCING 2 LINES.
     WRITE PRINT-RECORD FROM RPT2-HEADER2
           BEFORE ADVANCING 1 LINE.
     WRITE PRINT-RECORD FROM RPT2-HEADER3
           BEFORE ADVANCING 2 LINES.

     EXEC SQL
          DECLARE C2 CURSOR FOR
            SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),
                   SUM(EMPTIME/(1.0+:PERCENTAGE)),
                   SUM(EMPTIME)
            FROM USER1/TEMPRACT, USER1/TPROJ
            WHERE TEMPRACT.PROJNO=TPROJ.PROJNO
            GROUP BY TEMPRACT.PROJNO, PRNAME
            HAVING TEMPRACT.PROJNO LIKE :PROJID
            ORDER BY 1
     END-EXEC.
     EXEC SQL
          OPEN C2
     END-EXEC.

     PERFORM C000-GENERATE-REPORT2
          THRU C010-GENERATE-REPORT2-EXIT
          UNTIL SQLCODE NOT EQUAL TO ZERO.

 A200-DONE2.
     EXEC SQL
          CLOSE C2
     END-EXEC.
```

```
****************************************************************
*  All done.                                                   *
****************************************************************

 A900-MAIN-EXIT.
     CLOSE PRINTFILE.
     STOP RUN.

****************************************************************
*  Fetch and write the rows to PRINTFILE.                      *
****************************************************************

 B000-GENERATE-REPORT1.
     EXEC SQL
          WHENEVER NOT FOUND GO TO A100-DONE1
     END-EXEC.
     EXEC SQL
          FETCH C1 INTO :TEMPRACT
     END-EXEC.
     MOVE CORRESPONDING TEMPRACT TO RPT1-DATA.
     WRITE PRINT-RECORD FROM RPT1-DATA
           BEFORE ADVANCING 1 LINE.

 B010-GENERATE-REPORT1-EXIT.
     EXIT.

****************************************************************
*  Fetch and write the rows to PRINTFILE.                      *
****************************************************************

 C000-GENERATE-REPORT2.
     EXEC SQL
          WHENEVER NOT FOUND GO TO A200-DONE2
     END-EXEC.
     EXEC SQL
          FETCH C2 INTO :RPT2
     END-EXEC.
     MOVE CORRESPONDING RPT2 TO RPT2-DATA.
     WRITE PRINT-RECORD FROM RPT2-DATA
           BEFORE ADVANCING 1 LINE.

 C010-GENERATE-REPORT2-EXIT.
     EXIT.

****************************************************************
*  Error occurred while updating table.  Inform user and       *
*  rollback changes.                                           *
****************************************************************

 E010-UPDATE-ERROR.
     EXEC SQL
          WHENEVER SQLERROR CONTINUE
     END-EXEC.
     MOVE SQLCODE TO CODE-EDIT.
```

```
     STRING "**** ERROR Occurred while updating table.  SQLCODE="
           CODE-EDIT DELIMITED BY SIZE INTO PRINT-RECORD.
     WRITE PRINT-RECORD.
     EXEC SQL
          ROLLBACK
     END-EXEC.
     STOP RUN.

***************************************************************
*  Error occurred while generating reports.  Inform user and  *
*  exit.                                                      *
***************************************************************

 E020-REPORT-ERROR.
     MOVE SQLCODE TO CODE-EDIT.
     STRING "**** ERROR Occurred while generating reports. SQLCODE
-           "=" CODE-EDIT DELIMITED BY SIZE INTO PRINT-RECORD.
     WRITE PRINT-RECORD.
     STOP RUN.
```

Appendix C. Code Example For Use of SQL WHENEVER in PL/I

```
PLIEX: PROC;

  DCL PROJID CHAR(3);
  DCL PERCENTAGE FIXED DECIMAL(5,2);

  /* File declaration for sysprint */
  DCL SYSPRINT FILE EXTERNAL OUTPUT STREAM PRINT;

  /* Structure for report 1 */
  DCL 1 RPT1,
        %INCLUDE TEMPRACT(TEMPRACT,RECORD);

  /* Structure for report 2 */
  DCL 1 RPT2,
        15 PROJNO CHAR(6),
        15 PROJECT_NAME CHAR(36),
        15 EMPLOYEE_COUNT FIXED BIN(15),
        15 OLD_TOTAL_TIME FIXED DECIMAL(8,2),
        15 NEW_TOTAL_TIME FIXED DECIMAL(8,2);

  EXEC SQL INCLUDE SQLCA;

  PERCENTAGE = 0.06;
  PROJID = 'MA%';
  OPEN FILE(SYSPRINT);

  /* Update the selected projects by the new percentage. If an error */
  /* occurs during the update, ROLLBACK the changes.                 */
  EXEC SQL WHENEVER SQLERROR GO TO UPDATE_ERROR;
  EXEC SQL
       UPDATE USER1/TEMPRACT
          SET EMPTIME = EMPTIME * (1+:PERCENTAGE)
          WHERE PROJNO LIKE :PROJID;

  /* Commit changes */
  EXEC SQL
       COMMIT;
  EXEC SQL WHENEVER SQLERROR GO TO REPORT_ERROR;

  /* Report the updated statistics for each employee assigned to the */
  /* selected projects.                                              */

  /* Write out the header for Report 1 */
  put file(sysprint) edit('UPDATED EMPLOYEE PROJECT ACCOUNT DATA')
                         (col(1),a);
  put file(sysprint)
      edit('EMPLOYEE','PROJECT','ACCOUNT','EMPLOYEE')
          (skip(2),col(1),a,col(11),a,col(20),a,col(29),a);
  put file(sysprint)
      edit('NUMBER','NUMBER','NUMBER','HOURS')
          (skip,col(2),a,col(11),a,col(20),a,col(30),a,skip);
```

© Copyright IBM Corp. 1990

```
  exec sql
       declare c1 cursor for
         select *
         from user1/tempract
         where tempract.projno like :projid
         order by empno;
  EXEC SQL
       OPEN C1;

  /* Fetch and write the rows to SYSPRINT */
  EXEC SQL WHENEVER NOT FOUND GO TO DONE1;

  DO UNTIL (SQLCODE ¬= 0);
    EXEC SQL
         FETCH C1 INTO :RPT1;
    PUT FILE(SYSPRINT)
        EDIT(RPT1.EMPNO,RPT1.PROJNO,RPT1.ACTNO,RPT1.EMPTIME)
            (SKIP,COL(2),A,COL(11),A,COL(21),F(5),COL(29),F(8,2));
  END;

DONE1:
  EXEC SQL
       CLOSE C1;

  /* For each project selected, generate a report containing the     */
  /* project number, project name, the old total of employee hours,  */
  /* and the new total of employee hours for each project.           */

  /* Write out the header for Report 2 */
  PUT FILE(SYSPRINT) EDIT('ACCUMULATED STATISTICS BY PROJECT')
                          (SKIP(3),COL(22),A);
  PUT FILE(SYSPRINT)
      EDIT('PROJECT','NUMBER OF','PREVIOUS','CURRENT')
          (SKIP(2),COL(1),A,COL(48),A,COL(59),A,COL(70),A);
  PUT FILE(SYSPRINT)
      EDIT('NUMBER','PROJECT NAME','EMPLOYEES','HOURS','HOURS')
          (SKIP,COL(1),A,COL(10),A,COL(48),A,COL(60),A,COL(71),
           A,SKIP);

  EXEC SQL
       DECLARE C2 CURSOR FOR
         SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),
            SUM(EMPTIME/(1.0+:PERCENTAGE)),SUM(EMPTIME)
         FROM USER1/TEMPRACT, USER1/TPROJ
         WHERE TEMPRACT.PROJNO=TPROJ.PROJNO
         GROUP BY TEMPRACT.PROJNO, PRNAME
         HAVING TEMPRACT.PROJNO LIKE :PROJID
         ORDER BY 1;
  EXEC SQL
       OPEN C2;

  /* Fetch and write the rows to SYSPRINT */
  EXEC SQL WHENEVER NOT FOUND GO TO DONE2;
```

```
  DO UNTIL (SQLCODE ¬= 0);
    EXEC SQL
         FETCH C2 INTO :RPT2;
    PUT FILE(SYSPRINT)
        EDIT(RPT2.PROJNO,RPT2.PROJECT_NAME,EMPLOYEE_COUNT,
             OLD_TOTAL_TIME,NEW_TOTAL_TIME)
            (SKIP,COL(1),A,COL(10),A,COL(50),F(4),COL(59),F(8,2),
            COL(69),F(8,2));
  END;

DONE2:
  EXEC SQL
       CLOSE C2;
  GO TO FINISHED;

  /* Error occurred while updating table.  Inform user and rollback */
  /* changes.                                                        */
UPDATE_ERROR:
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  PUT FILE(SYSPRINT) EDIT('*** ERROR Occurred while updating table.'||
   '  SQLCODE=',SQLCODE)(A,F(5));
  EXEC SQL
       ROLLBACK;
  GO TO FINISHED;

  /* Error occurred while generating reports.  Inform user and exit. */
REPORT_ERROR:
  PUT FILE(SYSPRINT) EDIT('*** ERROR Occurred while generating '||
   'reports.  SQLCODE=',SQLCODE)(A,F(5));
   GO TO FINISHED;

  /* All done  */
FINISHED:
  CLOSE FILE(SYSPRINT);
  RETURN;

END PLIEX;
```

Appendix D. Code Example For Use of SQL in C/400

```
 1  #include "string.h"
 2  #include "stdlib.h"
 3  #include "stdio.h"
 4
 5  main()
 6  {
 7     char projid??(3??);
 8     double percentage;
 9
10     /* File declaration for qprint */
11     FILE *qprint;
12
13     /* Structure for report 1 */
14     struct {
15             char empno??(7??);
16             char projno??(7??);
17             short actno;
18             char startdate??(7??);
19             char enddate??(7??);
20             float emptime;
21             } rpt1;
22
23     /* Structure for report 2 */
24     struct {
25             char projno??(7??);
26             char project_name??(37??);
27             short employee_count;
28             double old_total_time,new_total_time;
29            } rpt2;
30
31     EXEC SQL INCLUDE SQLCA;
32
33     percentage = 0.06;
34     strcpy(projid,"MA%");
35     qprint=fopen("QPRINT","w");
36
37     /* Update the selected projects by the new percentage. If an error */
38     /* occurs during the update, ROLLBACK the changes.                 */
39     EXEC SQL WHENEVER SQLERROR GO TO update_error;
40     EXEC SQL
41          UPDATE USER1/TEMPRACT
42             SET EMPTIME = EMPTIME * (1+:percentage)
43             WHERE PROJNO LIKE :projid;

44
45     /* Commit changes */
46     EXEC SQL
47          COMMIT;
48     EXEC SQL WHENEVER SQLERROR GO TO report_error;
49
50     /* Report the updated statistics for each employee assigned to the */
51     /* selected projects.                                               */
52
53     /* Write out the header for Report 1 */
54     fprintf(qprint,    "UPDATED EMPLOYEE PROJECT ACCOUNT DATA");
```

© Copyright IBM Corp. 1990

```
 55     fprintf(qprint,"\n\nEMPLOYEE  PROJECT  ACCOUNT  EMPLOYEE");
 56     fprintf(qprint,  "\n NUMBER   NUMBER   NUMBER    HOURS\n");
 57
 58     exec sql
 59          declare c1 cursor for
 60            select *
 61            from user1/tempract
 62            where tempract.projno like :projid
 63            order by empno;
 64     EXEC SQL
 65          OPEN C1;
 66
 67     /* Fetch and write the rows to QPRINT */
 68     EXEC SQL WHENEVER NOT FOUND GO TO done1;
 69
 70     do {
 71       EXEC SQL
 72            FETCH C1 INTO :rpt1;
 73       fprintf(qprint,"\n %6s   %6s   %6d   %8.2f",
 74               rpt1.empno,rpt1.projno,rpt1.actno,rpt1.emptime);
 75       }
 76     while (SQLCODE==0);
 77
 78   done1:
 79     EXEC SQL
 80          CLOSE C1;
 81

 82     /* For each project selected, generate a report containing the     */
 83     /* project number, project name, the old total of employee hours,  */
 84     /* and the new total of employee hours for each project.           */
 85
 86     /* Write out the header for Report 2 */
 87     fprintf(qprint,"\n\n\n                     ACCUMULATED STATISTICS\
 88   BY PROJECT");
 89     fprintf(qprint,  "\n\nPROJECT                                \
 90      NUMBER OF  PREVIOUS   CURRENT");
 91     fprintf(qprint,    "\nNUMBER   PROJECT NAME                  \
 92      EMPLOYEES   HOURS      HOURS\n");
 93
 94     EXEC SQL
 95          DECLARE C2 CURSOR FOR
 96            SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),
 97               SUM(EMPTIME/(1.0+:percentage)),SUM(EMPTIME)
 98            FROM USER1/TEMPRACT, USER1/TPROJ
 99            WHERE TEMPRACT.PROJNO=TPROJ.PROJNO
100            GROUP BY TEMPRACT.PROJNO, PRNAME
101            HAVING TEMPRACT.PROJNO LIKE :projid
102            ORDER BY 1;
103     EXEC SQL
104          OPEN C2;
105
106     /* Fetch and write the rows to QPRINT */
107     EXEC SQL WHENEVER NOT FOUND GO TO done2;
108
109     do {
110       EXEC SQL
111            FETCH C2 INTO :rpt2;
112       fprintf(qprint,"\n%6s   %36s  %6d     %8.2f  %8.2f",
113               rpt2.projno,rpt2.project_name,rpt2.employee_count,
```

```
114              rpt2.old_total_time,rpt2.new_total_time);
115     }
116     while (SQLCODE==0);
117
118   done2:
119     EXEC SQL
120          CLOSE C2;
121     goto finished;
122
123     /* Error occured while updating table.  Inform user and rollback   */
124     /* changes.                                                        */
125   update_error:
126     EXEC SQL WHENEVER SQLERROR CONTINUE;
127     fprintf(qprint,"*** ERROR Occurred while updating table.  SQLCODE="
128             "%5d\n",SQLCODE);
129     EXEC SQL
130          ROLLBACK;
131     goto finished;
132
133     /* Error occured while generating reports.  Inform user and exit.  */
134   report_error:
135     fprintf(qprint,"*** ERROR Occurred while generating reports.  "
136             "SQLCODE=%5d\n",SQLCODE);
137     goto finished;
138
139     /* All done  */
140   finished:
141     fclose(qprint);
142     exit(0);
143
144 }
```

Appendix E. Code Example For Use of SQL in FORTRAN/400

```
 1        PROGRAM FTNEX
 2
 3        CHARACTER*3 PROJID
 4        REAL*4      PERCENTAGE
 5
 6  ****************************************************************
 7  *  Variables for report 1.                                     *
 8  ****************************************************************
 9
10        CHARACTER*6 EMPNO, PROJNO1, ENDDATE, STARTDATE
11        INTEGER*2   ACTNO
12        REAL*8      EMPTIME
13
14  ****************************************************************
15  *  Variables for report 2.                                     *
16  ****************************************************************
17
18        CHARACTER   PROJNO2*6, PROJECT_NAME*36
19        INTEGER*4   EMPLOYEE_COUNT
20        REAL*8      OLD_TOTAL_TIME, NEW_TOTAL_TIME
21
22        EXEC SQL
23       -     INCLUDE SQLCA
24
25
26        PERCENTAGE = 0.06
27        PROJID = 'MA%'
28        OPEN(7,FILE='QPRINT')
29
30  ****************************************************************
31  * Update the selected projects by the new percentage. If an    *
32  * error occurs during the update, ROLLBACK the changes.        *
33  ****************************************************************
34
35        EXEC SQL
36       -     WHENEVER SQLERROR GO TO 99900
37        EXEC SQL
38       -     UPDATE USER1/TEMPRACT
39       -        SET EMPTIME = EMPTIME * (1+:PERCENTAGE)
40       -        WHERE PROJNO LIKE :PROJID
41
42  ****************************************************************
43  *  Commit changes.                                             *
44  ****************************************************************
45
46        EXEC SQL
47       -     COMMIT
48
49        EXEC SQL
50       -     WHENEVER SQLERROR GO TO 99910
51
```

© Copyright IBM Corp. 1990

```
 52   ***************************************************************
 53   *  Report the updated statistics for each employee assigned to*
 54   *  the selected projects.                                      *
 55   ***************************************************************
 56
 57   ***************************************************************
 58   *  Write out the header for Report 1.                          *
 59   ***************************************************************
 60
 61         WRITE(7,10000)
 62         WRITE(7,10010)
 63         WRITE(7,10020)
 64         exec sql
 65        -     declare c1 cursor for
 66        -       select *
 67        -         from user1/temprract
 68        -         where tempract.projno like :projid
 69        -         order by empno
 70         EXEC SQL
 71        -     OPEN C1
 72
 73   ***************************************************************
 74   *  Fetch and write the rows to QPRINT.                         *
 75   ***************************************************************
 76
 77         EXEC SQL
 78        -     WHENEVER NOT FOUND GO TO 110
 79   100   CONTINUE
 80         EXEC SQL
 81        -     FETCH C1 INTO :EMPNO, :PROJNO1, :ACTNO,
 82        -                    :STARTDATE, :ENDDATE, :EMPTIME
 83         WRITE(7,10040) EMPNO,PROJNO1,ACTNO,EMPTIME
 84         GO TO 100
 85   110   CONTINUE
 86         EXEC SQL
 87        -     CLOSE C1
 88
 89   ***************************************************************
 90   *  For each project selected, generate a report containing the*
 91   *  project number, project name, the old total of employee    *
 92   *  hours, and the new total of employee hours for each        *
 93   *  project.                                                    *
 94   ***************************************************************
 95

 96
 97   ***************************************************************
 98   *  Write out the header for Report 2.                          *
 99   ***************************************************************
100         WRITE(7,10050)
101         WRITE(7,10060)
102         WRITE(7,10070)
103
104         EXEC SQL
105        -     DECLARE C2 CURSOR FOR
106        -       SELECT TEMPRACT.PROJNO, PRNAME, COUNT(*),
107        -              SUM(EMPTIME/(1.0+:PERCENTAGE)),SUM(EMPTIME)
108        -         FROM USER1/TEMPRACT, USER1/TPROJ
109        -         WHERE TEMPRACT.PROJNO=TPROJ.PROJNO
110        -         GROUP BY TEMPRACT.PROJNO, PRNAME
```

```
111         -         HAVING TEMPRACT.PROJNO LIKE :PROJID
112         -         ORDER BY 1
113          EXEC SQL
114         -     OPEN C2
115
116   ***************************************************************
117   *  Fetch and write the rows to QPRINT.                        *
118   ***************************************************************
119
120          EXEC SQL
121         -     WHENEVER NOT FOUND GO TO 210
122   200    EXEC SQL
123         -FETCH C2 INTO :PROJNO2, :PROJECT_NAME,
124         -:EMPLOYEE_COUNT,:OLD_TOTAL_TIME,:NEW_TOTAL_TIME
125          WRITE(7,10080)PROJNO2,PROJECT_NAME,EMPLOYEE_COUNT,
126         -        OLD_TOTAL_TIME,NEW_TOTAL_TIME
127          GO TO 200
128   210    CONTINUE
129          EXEC SQL
130         -     CLOSE C2
131
132   ***************************************************************
133   *  All done.                                                  *
134   ***************************************************************
135
136          CLOSE(7)
137          STOP
138
139   99900 CONTINUE
140   ***************************************************************
141   *  Error occured while updating table.  Inform user and       *
142   *  rollback changes.                                          *
143   ***************************************************************
144
145          EXEC SQL
146         -     WHENEVER SQLERROR CONTINUE
147          WRITE(7,10090) SQLCODE
148          EXEC SQL
149         -     ROLLBACK
150          STOP
151
152   ***************************************************************
153   *  Error occured while generating reports.  Inform user and   *
154   *  exit.                                                      *
155   ***************************************************************
156
157   99910 CONTINUE
158          WRITE(7,10100) SQLCODE
159          STOP
160   10000 FORMAT('UPDATED EMPLOYEE PROJECT ACCOUNT DATA')
161   10010 FORMAT(//'EMPLOYEE',T11,'PROJECT',T20,'ACCOUNT',T29,'EMPLOYEE')
162   10020 FORMAT(/' NUMBER',T11,'NUMBER',T20,'NUMBER',T30,'HOURS'/)
163   10040 FORMAT(T2,A,T11,A,T21,I5,T29,F8.2)
164   10050 FORMAT(//T22,'ACCUMULATED STATISTICS BY PROJECT')
165   10060 FORMAT(/'PROJECT',T48,'NUMBER OF',T59,'PREVIOUS',T70,'CURRENT')
166   10070 FORMAT('NUMBER',T10,'PROJECT NAME',T48,'EMPLOYEES',T60,'HOURS',
167         -       T71,'HOURS')
168   10080 FORMAT(A,T10,A,T50,I4,T59,F8.2,T69,F8.2)
169   10090 FORMAT('*** ERROR Occurred while updating table.  SQLCODE=',
```

```
170          -        I6)
171   10100 FORMAT('*** ERROR Occurred while generating reports.  SQLCODE=',
172          -        I5)
173           END
```

Appendix F. Sample Tables Used in Examples

Inventory Table

```
CREATE TABLE INVENTORY
(PARTNO SMALLINT NOT NULL,
DESCR   CHAR(24) NOT NULL,
ONHAND  INTEGER  NOT NULL)
```

PARTNO	DESCR	ONHAND
207	GEAR	600
209	CAM	50
221	BOLT	650
222	BOLT	1250
231	NUT	700
232	NUT	1100
241	WASHER	6000
285	WHEEL	350
295	BELT	85

Table F-1. Sample Inventory Table

© Copyright IBM Corp. 1990

Supplier Table

```
CREATE TABLE SUPPLIER
(SUPPNO SMALLINT NOT NULL,
NAME    CHAR(15) NOT NULL,
ADDRESS CHAR(35) NOT NULL)
```

SUPPNO	NAME	ADDRESS
51	DEFECTO PARTS	6 BUM ST, BROKEN HAND, WY
52	VESUVIUS INC	512 ANCIENT BLVD, POMPEII
53	ATLANTIS CO	8 OCEAN AVE, WASHINGTON DC
54	TITANIC PARTS	32 LARGE ST, BIGTOWN TX
57	EAGLE HARDWARE	64 TRANQUILITY PLACE, MN
61	SKY PARTS	128 ORBIT BLVD, SYDNEY AUSTRALIA
64	KNIGHT LTD	256 ARTHUR COURT, CAMELOT ENGLAND

Table F-2. Sample Supplier Table

Quotations Table

```
CREATE TABLE QUOTATIONS
(SUPPNO SMALLINT NOT NULL,
PARTNO  SMALLINT NOT NULL,
PRICE   DECIMAL  NOT NULL,
DELTIME SMALLINT NOT NULL,
ONORDER INTEGER  NOT NULL)
```

SUPPNO	PARTNO	PRICE	DELTIME	ONORDER
51	221	.30	10	50
51	231	.10	10	0
53	222	.25	15	200
53	232	.10	15	0
53	241	.08	15	0
54	209	18.00	21	0
54	221	.10	30	150
54	231	.04	30	200
54	241	.02	30	200
57	285	21.00	15	0
57	295	8.50	21	24
61	221	.20	21	200
61	222	.20	21	200
61	241	.05	21	0
64	207	29.00	14	20
64	209	19.50	7	7

Table F-3. Sample Quotations Table

Employee Table

```
CREATE TABLE TEMPL
(EMPNO CHAR(6) NOT NULL WITH DEFAULT,
FIRSTNME CHAR(12) NOT NULL WITH DEFAULT,
MIDINIT CHAR(1)  NOT NULL WITH DEFAULT,
LASTNAME CHAR(15) NOT NULL WITH DEFAULT,
DEPTNO CHAR(3) NOT NULL WITH DEFAULT,
PHONENO CHAR(4)  NOT NULL WITH DEFAULT,
HIREDATE CHAR(6) NOT NULL WITH DEFAULT,
JOBCODE DECIMAL(3) NOT NULL WITH DEFAULT,
EDUCLVL SMALLINT NOT NULL WITH DEFAULT,
SEX CHAR(1) NOT NULL WITH DEFAULT,
BRTHDATE CHAR(6) NOT NULL WITH DEFAULT,
SALARY DECIMAL(8,2)  NOT NULL WITH DEFAULT)
```

EMP-NO	FIRSTNME	MID-INIT	LASTNAME	DEPT-NO	PHONE-NO	HIRE-DATE	JOB-CODE	EDUC-LVL	SEX	BRTH-DATE	SAL-ARY
000010	Christine	I	Haas	A00	3978	750101	66	18	F	330814	52750
000020	Michael	L	Thompson	B01	3476	731010	61	18	M	480202	41250
000030	Sally	A	Kwan	C01	4738	750405	60	20	F	410511	38250
000050	John	B	Geyer	E01	6789	690817	58	16	M	450915	40175
000060	Irving	F	Stern	D11	6423	730914	55	16	M	450707	32250
000070	Eva	D	Pulaski	D21	7831	800930	56	16	F	530526	36170
000090	Eileen	W	Henderson	E11	5498	700815	55	16	F	410515	29750
000100	Theodore	Q	Spenser	E21	0972	800619	54	14	M	561218	26150
000110	Vicenzo	G	Lucchesi	A00	3490	680516	58	19	M	491105	46500
000120	Sean		O'Connell	A00	2167	731205	58	14	M	421018	29250
000130	Delores	M	Quintana	C01	4578	710728	55	16	F	350915	23800
000140	Heather	A	Nicholls	C01	1793	761215	55	18	F	460119	28420
000150	Bruce		Adamson	D11	4510	720212	55	16	M	470517	25280
000160	Elizabeth	R	Pianka	D11	3782	771011	54	17	F	550412	22250
000170	Masatoshi	J	Yoshimura	D11	2890	780915	54	16	M	510105	24680
000180	Marilyn	S	Scoutten	D11	1682	730707	53	17	F	490221	21340
000190	James	H	Walker	D11	2986	740726	53	16	M	520625	20450
000200	David		Brown	D11	4501	760303	55	16	M	410529	27740
000210	William	T	Jones	D11	0942	790411	52	17	M	530223	18270
000220	Jennifer	K	Lutz	D11	0672	780829	55	18	F	480319	29840
000230	James	J	Jefferson	D21	2094	761121	53	14	M	350530	22180
000240	Salvatore	M	Marino	D21	3780	791205	55	17	M	540331	28760
000250	Daniel	S	Smith	D21	0961	791030	52	15	M	391112	19180
000260	Sybil	P	Johnson	D21	8953	750911	52	16	F	361005	17250
000270	Maria	L	Perez	D21	9001	800930	55	15	F	530526	27380
000280	Ethel	R	Schneider	E11	8997	770324	54	17	F	360328	26250

Employee Project Account Table

```
CREATE TABLE TEMPRACT
(EMPLNO CHAR(6) NOT NULL WITH DEFAULT,
PROJNO CHAR(6) NOT NULL WITH. DEFAULT,
ACTNO SMALLINT NOT NULL WITH DEFAULT,
STARTDATE CHAR(6) NOT NULL WITH DEFAULT,
ENDDATE CHAR(6) NOT NULL WITH DEFAULT,
EMPTIME DECIMAL(5,2) NOT NULL WITH DEFAULT)
```

EMPLNO	PROJNO	ACTNO	STARTDATE	ENDDATE	EMPTIME
000160	MA2100	20	860501	860829	500
000170	MA2100	20	860901	861231	500
000180	MA2100	20	870105	870430	650
000060	MA2100	10	870101	881101	500
000110	MA2100	20	880101	880301	400
000220	MA2112	50	871001	880615	900
000170	MA2112	70	870601	880102	100
000190	MA2112	70	880201	880601	100
000180	MA2113	70	870401	871215	400
000210	MA2113	80	870401	871215	500
000230	MA2113	70	870401	871215	300
000010	AD3100	10	880101	880701	500
000070	AD3110	10	880101	880201	100
000230	AD3111	60	880101	880315	100
000240	AD3111	70	880215	880915	500
000250	AD3112	60	880101	880201	100
000270	AD3113	60	880301	880401	100
000260	AD3113	70	880615	880701	80

Table F-4. Sample Employee Project Account Table

Department Table

```
CREATE TABLE TDEPT
(DEPTNO CHAR(3) NOT NULL WITH DEFAULT,
DEPTNAME CHAR(36) NOT NULL WITH DEFAULT,
MGRNO CHAR(6) NOT NULL WITH DEFAULT,
ADMRDEPT CHAR(3) NOT NULL WITH DEFAULT)
```

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
A00	Computer Service Div.	000010	
B01	Planning	000020	A00
C01	Information Center	000030	A00
D01	Development Center		A00
E01	Support Services	000050	A00
D11	Manufacturing Systems	000060	D01
D21	Administration Systems	000070	D01
E11	Operations	000090	E01
E21	Software Support	000100	E01

Table F-5. Sample Department Table

Project Table

```
CREATE TABLE TPROJ
(PROJNO CHAR(6) NOT NULL WITH DEFAULT,
PRNAME CHAR(36) NOT NULL WITH DEFAULT,
DEPTNO CHAR(3) NOT NULL WITH DEFAULT,
DEPTMGR CHAR(6) NOT NULL WITH DEFAULT,
PRSTAFF DECIMAL(5,2) NOT NULL WITH DEFAULT,
PRSTDATE CHAR(6) NOT NULL WITH DEFAULT,
PRENDATE CHAR(6) NOT NULL WITH DEFAULT,
MAJPROJ CHAR(6) NOT NULL WITH DEFAULT)
```

PROJNO	PRNAME	DEPTNO	DEPTMGR	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
AD3100	ADMINISTRATION SERVICES	D01	000010	6.5	860101	830201	
AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	871001	880215	AD3100
AD3111	PAYROLL PROGRAMMING	D21	000230	2	880101	880401	AD3110
AD3112	PERSONNEL PROGRAMMING	D21	000250	1	870320	870601	AD3110
AD3113	ACCOUNT PROGRAMMING	D21	000270	2	870901	880315	AD3110
MA2100	MFG AUTOMATION	D11	000060	12	860324		
MA2110	MFG PROGRAMMING	E21	000100	3	870928	880219	MA2100
MA2112	ROBOT DESIGN	E01	000050	3	860106	881111	MA2110
MA2113	PROD CONTROL PROG	D11	000060	3			MA2100

Index

A

access path 5-31
access plan 5-2, 5-47
ALCOBJ 6-3
ALIAS 3-38
ALL authority 8-4
always-active HLL 5-15
analyzing performance problems 5-52
ANZDBF and ANZDBKEY 5-53
apostrophe, in COBOL and C programs 3-19
application design 5-15
application development with SQL 3-32
arrival sequence 5-31
associated space, of program object 5-47
AS/400 PC Support 6-2
AS/400 Query 1-2, 2-5, 6-2
authorization
  control differences 11-2
  ID 3-1, 8-1
  in dynamic SQL 4-10
  list 8-5, 8-6

B

binding 3-20, 5-2, 5-3
  AS/400 Query 5-3
  dynamic SQL programs 5-2
  OPNQRYF 5-3
  OS/400 Query Management 5-3
  PC Support file Transfer 5-3
  physical and logical files 5-3
  summary 5-4
building access path message, when performing SELECT 3-6

C

cartesian product 3-10, 5-49
catalog 3-3
  LFs and views 8-2
  maintenance, performance implications 3-4
  tables 8-5, 8-7
change System/36 Environment (CHGS36) 3-46
changing file/table structure 3-36
CLRPFM 5-14
COBOL GOBACK statement 11-5
COBOL programming 3-37
Codd, Dr E.F. 12-1
collection 8-2
column selection 5-35
COMIT (RPG) 7-10
COMMENT 2-6
commit 3-28, 7-10
COMMIT and ROLLBACK with HOLD 7-6
COMMIT parameter, precompile option 3-18
commitment control 3-44, 8-8
  after CPYF/MOVOBJ 3-35
  and batch applications 7-2
  and interactive applications 7-1
  and row locking 7-3
  DDM files 6-4
  in different collections 7-12, 7-13
  maximum row lock 4-2
concurrency 7-3
  optimizing 5-29
COPY DD 3-39
COPY DDS 3-35, 3-37, 3-42
copying data 3-33
correlated subquery 3-15
correlation names 3-11
cost estimation 5-45
CPYF 3-34
  commitment control 7-14
  *MAP (changing table/file structure) 3-36
CREATE 2-6
CREATE TABLE 3-33
creating a temporary table, in a program 4-3
CRTDUPOBJ 3-34
CRTSQLxxx commitment control levels 7-10
CSP implications 5-16
cursor stability and *CHG 7-2
cursors
  definition of 4-1
  processing with 4-4
  processing without 4-1
C/400 debug 3-18

D

data conversion considerations 11-3
data definition language - portability 11-1
data description specifications 2-1
data dictionary 8-2
data manipulation language and portability 11-4
data migration 11-3
data space 5-32
  scan 5-32
database design 5-9
database, name change 3-3
data/text merge option (AS/400 Query) 1-3
DDL 2-6
DDM files 1-3, 3-44, 6-1
  PC Support 1-5
DDS 2-1
  files 1-2, 1-3, 3-33, 3-45
debug
  C/400 3-18

© Copyright IBM Corp. 1990

debug, output in job log 5-54
decision-making rules, of optimizer 5-48
declarative SQL statements 3-43
declare cursor 3-43
declare statement 3-43
default filter factors 5-47
default values 7-1
definition
  access path 5-31
  access plan 5-2
  arrival sequence 5-31
  binding 5-2
  column selection 5-35
  cursor 5-5
  data space 5-32
  default filter factors 5-47
  file management row selection 5-36
  index from index 5-35
  key row positioning 5-33
  key Row selection 5-33
  keyed sequence 5-31
  late binding 5-3
  left-most key 5-32
  miniplan 5-47
  open data path (ODP) 5-2
  optimizer 5-7
  OS/400 query component 5-5
  primary key 5-32
  prototype ODP 5-3
  query 5-5
  query definition template (QDT) 5-7
  rebinding 5-3
  "haven't spent too much time" condition 5-48
deleting 4-3, 4-7
  using CLRPFM 5-14
design (performance)
  application 5-15
  database 5-9
  guidelines 5-8
  program 5-17
device file (DDM) 6-1
DFU 2-2, 2-5
  DFU/38 6-2
  DFU/400 6-2
  to insert into views 3-7
dictionary objects (LF's) 8-2
dictionary objects (PFs) 8-2
display data base relations (DSPDBR) 5-54
display-station passthrough 6-2
distributed unit-of-work 6-1
DROP 2-6
DSPFFD 3-4
DSPMSGD CL command 3-28
DSPOBJAUT 8-1
dynamic
  non-SELECT statements 4-10
  row selection 5-32
  SELECT statements 4-7
dynamic *(continued)*
  SQL 4-7
  SQL performance 4-13
  SQL, GRANT and REVOKE 8-1

E

EDTOBJAUT 8-1
ending journaling 7-13
entity integrity 12-3
equi-join 3-8
error handling 3-44
error message text 3-27
EXECUTE IMMEDIATE 4-10
externally described files 3-32

F

FEOD (Force End of Data) - INSERT with subselect 5-72
field definitions, in IDDU 2-5
file
  conversion (System/36) 3-45
  definitions, in IDDU 2-4
  size, when creating index 3-5
file management row selection 5-36
filter factors, default 5-47
FOR UPDATE OF clause 4-6
foreign key 12-3

G

GRANT 8-1
greater-than, less-than and non-equi joins 3-10
GRTOBJAUT 8-1

H

high-level language programs 3-37

I

I-O-CONTROL (COBOL) 7-10
IDDU 2-4
  files 1-2, 1-3, 3-33, 3-45
implementation cost 5-45
INCLUDE 3-41, 3-43
  file, precompiler option 3-19
indexed file organization 3-29
indexes 3-5, 5-31
  correct usage 5-52
  creating 3-35, 5-11
    from another index 5-35
  how to determine if they are being used 5-55
  identifying a new index created for SELECT 5-56
  identifying an index created from an index 5-57
  identifying temporary index usage 5-56, 5-59
  implicit sharing 5-12
  inefficient, how to identify 5-58

indexes *(continued)*
  optimizing usage 5-18
indicator variables 3-28
inner join 3-12
inserting 4-3
  data into views 3-7
Interactive Data Definition Utility 2-4
interactive implications under commitment control 7-2
interactive SQL 8-7, 9-1
  for debugging performance problems 5-54
  new sort 5-71
invocations 5-15
isolation levels 11-4

J

job log 5-54
job trace 5-59
  STRJOBTRC 5-60
  TRCINT 5-60
  TRCJOB 5-59
join 2-3, 3-8
  and index creation 3-5
  fields, matching attributes 5-13
  files 3-35
  logical files in DDS 3-35
  on OPNQRYF 1-4
  optimization 5-48
  ordering algorithm 5-49
  update capable, partial 5-20
  using PC Support 1-5
join optimization 5-23
  forcing temporary result file 5-25
  index over secondary file 5-24
  join from smallest to largest 5-25
  many record selection conditions 5-24
  predicates on WHERE clause 5-23
  primary join file has fewest number of selected rows 5-24
  redundant WHERE predicates 5-24
journal 8-2
  entries for SQL update statements 5-72
  receiver 8-2
journaling 7-10, 7-12
  to different journals in same application 7-13
  with copy/move commands 3-34
  with different collections 3-35

K

KCOMIT (RPG) 7-10
key range estimates 5-47
key row positioning 5-33
key row selection 5-33
keyed sequence 5-31

L

LABEL 2-6
late binding 5-3
left-most key 5-32
LIKE comparison operator, and indexes 3-7
literals, use of in COBOL and C programs 3-19
locking rules 11-4
logical file 2-1, 8-1

M

mass insertions 4-3
Menu-Based Natural Language Query (MBNLQ) 1-6
message text, for SQL messages 3-28
module names to look for in trace 5-60
  QDBFIXIT 5-62
  QDBGETKY 5-64
  QDBGETM 5-64
  QDBGETSQ 5-64
  QMH????? 5-64
  QQQACTIV 5-62
  QQQGET 5-62
  QQQIMPLE 5-62
  QQQISVSU 5-40, 5-61
  QQQITEMP 5-62
  QQQOPTIM 5-61
  QQQQEXIT 5-62
  QQQQUERY 5-61
  QQQSETUP 5-61
  QQQSQCMP 5-61
  QQQVALID 5-61
  QQQVMRCY 5-62
  QQQVWCMP 5-61
  QQQVWFLD 5-61
  QSQAUTH 5-64
  QSQBIND 5-64
  QSQBLQDT 5-64
  QSQCLOSE 5-64
  QSQCRTDB 5-64
  QSQCRTI 5-64
  QSQCRTT 5-64
  QSQCRTV 5-64
  QSQDELET 5-64
  QSQDESC 5-64
  QSQDROP 5-64
  QSQFETCH 5-64
  QSQINS 5-64
  QSQLABEL 5-64
  QSQLCCR 5-64
  QSQOPEN 5-64
  QSQPREP 5-64
  QSQPSTAB 5-64
  QSQRAPLY 5-64
  QSQRCHK 5-64
  QSQRLEX 5-64
  QSQROUTE 5-64
  QSQRPARS 5-64
  QSQRPTAB 5-64

module names to look for in trace (continued)
QSQRTBLS 5-64
QSQRTOKR 5-64
QSQRXLTR 5-64
QSQRXTRT 5-65
QSQUPDAT 5-65
QSQXCUTE 5-64
QSQXIT 5-65
moving data 3-33
MOVOBJ 3-34
commitment control 7-14
multiple access plans 5-4
multiple format logical files 1-3
multiple systems with SQL/400 3-44

N

naming conventions 3-1, 3-2
naming convention, SQL 3-19
nested SELECT 3-13
non-reusable ODP 5-39
non-SQL table access 3-42
normalization 3-8, 5-9
nulls 3-7
NOT NULL 3-36

O

ODP
office 5-6
open data path
definition 5-2
generalized 5-41
non-reusable 5-39
reusability 5-38
reusable 5-39
reusable ODP
across invocations 5-15, 5-71
OPNQRYF 1-4, 6-2
optimizer 5-7, 5-44, 5-48
catalog use of 3-4
decision-making rules 5-48
optimizer weightings 5-46
ALLIO 5-46
FIRSTIO 5-46
MINWAIT 5-46
optimizing
concurrency 5-29
CPU usage 5-17
index usage 5-18
I/O with blocking 5-29
ORDER BY 3-38
with UNION 3-17
OS/400 query component 5-5, 5-44
outer join 3-12
OVRDBF 3-37

P

page faults 5-32
parameter markers 4-11
PC Support (AS/400) 1-5
performance
arithmetic expressions 5-27
ASP 5-14
BETWEEN clause on keys 5-23
checklist 5-8
cursor operation 5-19
database file management 5-14
dynamic SQL 5-17
journal management 5-14
LIKE predicate 3-32, 5-23
native file management vs SQL 5-67
numeric conversion 5-26
SQL in general 5-68
SQL versus HLL I/O 3-32
SQL vs keyed files 5-68
SQL vs non-keyed files 5-68
SQL vs OPNQRYF 5-69
static vs dynamic SQL 5-69
string truncation 5-26
updating via cursor operation 5-19
user ASP 5-14
performance enhancements in release 3.0 5-71
performance tools 5-6, 5-65
physical file 8-1
physical files 2-1
PL/I program, WHENEVER clause 3-25
portability 11-1
pre-fetching 5-33
precompile
listing 3-18
optimization 5-44
precompilers 9-1
precompiling 5-2, 5-44
primary key 5-32, 12-3
program adoption of authority 8-8
program design 5-17
project 2-3
prompting of SQL statement in SEU 9-3
prototype ODP 5-3

Q

QSQJRN 3-35
Query 1-3
query definition template (QDT) 5-7, 5-44, 5-47
Query/38 6-2
licensed program product (5728-DB1) 1-3
tables 1-3
quotes, in COBOL and C programs 3-19
Q&A database 5-6

R

random access of tables 3-43
re-optimization 5-47
read only declare cursor, columns in 5-71
READ PRIOR 3-42
read prior in SQL 3-29
read-only remote file/table access 6-3
rebinding 5-3
recommendation
  DDL methodology 2-9
  for use of Interactive SQL 9-3
  naming convention 3-2
  security implementation 8-6
record format definitions, in IDDU 2-5
record order 3-42
records retrieved after index positioning 5-72
redundant data 5-9
referential integrity 12-3
relational operators 2-3
relative file organization 3-29
remote file access 6-1
remote file/table updating 6-3
remote unit-of-work 6-1
renaming fields, RPG 3-40
repeatable read and *ALL 7-2
retrieval 4-4
retrieving 4-1
reusability 5-38
  of ODP's 5-38
  restrictions 5-71
reusable ODP 5-39
  across invocations 5-15, 5-71
REVOKE 8-1
ROLBK (RPG) 7-10
rollback 3-28, 7-10
row locking and commitment control 7-3
row selection methods
  column selection 5-35
  dynamic row selection 5-32
  file management row selection 5-36
  index from index 5-35
  key row positioning 5-33
  key row selection 5-33
rows
  length 5-11
  number 5-11
  number inserted, updated or deleted 3-27
RPG Host Variable Definition 3-39
RPG programming 3-38
RSTOBJ 3-34
RTVMSG CL command 3-28
RUNQRY 1-2
RVKOBJAUT 8-1

S

SAA
  and SQL 3-43

SAA *(continued)*
  functions not implemented in SQL/400 11-1
  size limits 11-1
screen design aid, use with SQL tables 3-43
SDA 2-2, 6-2
  with SQL tables 3-43
second level message 3-28
seize contention 5-72
SELECT 2-3, 3-35
  SELECT statements, with varying-list selection 4-10
  statements, with fixed-list selection 4-7
SELECT INTO processing 5-72
sequence 2-3
sequential file organization 3-29
set position exception 5-72
SEU 6-2
  prompting of SQL statement 9-3
sharing select/omit access paths 5-71
SQL
  data definition language 2-6
  INCLUDE 3-41, 11-5
  naming convention 3-1
SQLCA 3-27
SQLCA, use of in a program 3-20
SQL/400 1-1
  data definition extensions 11-2
standards 10-1
statement entry screen 9-1
static and dynamic SQL 4-1
static SQL 4-1
storage when building access paths 5-72
STRCMTCTL 7-10
STRJRNPF 3-35
STRQRY 1-2
STRSQL 9-1
structure - changing tables or files 3-36
subquery 3-13
  correlated 3-15
  join implementation 5-50
  non-join implementation 5-51
  optimization 5-50
subquery performance 5-71
subroutines in WHENEVER processing 3-22
system catalogs 11-2
system naming convention 3-2
System/36
  Environment 3-46
  file conversion 3-45
  file library (QS36F) 3-46
System/38
  Query 1-3
  Utilities (5728-DB1) 1-3

T

table size 5-11
tables (System/38) 1-3

test data files 3-37
timing and paging statistics tool (TPST) 5-65
transferring data (PC Support) 1-5
truncation 3-29

U

underline character (_) 3-38
union 2-3, 3-15
update capable join, partial 5-20
updating 4-2
  with ORDER BY on SELECT 4-4
user ASP 5-14
user tables 8-2
user views 8-2

V

validation 5-47
view authority 8-3
views 3-7

W

WHENEVER 3-21, 3-43, 3-45
  with continue 3-23
wordiness 3-30
work with commands, identifying index creation 5-59
workstation files 3-43
WRKQRY 1-2

READER'S COMMENTS

Title: *SQL/400: A Guide for Implementation*
GG24-3321-01
International Technical Support Center, Rochester

You may use this form to communicate your comments about this publication, its organization or subject matter with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Comments:

Reply requested Name : ______________________________

Yes / No Job Title : ______________________________

Address : ______________________________

____

Cut or Fold Along Line

Reader's Comment Form

Fold and tape — Please Do Not Staple — Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 40 ARMONK, N Y

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Technical Support Center
Department 977, Building 663-3
Highway 52 and NW 37th Street
Rochester, Minnesota 55901 U.S.A.

Fold and tape — Please Do Not Staple — Fold and tape

READER'S COMMENTS

Title: *SQL/400: A Guide for Implementation*
GG24-3321-01
International Technical Support Center, Rochester

You may use this form to communicate your comments about this publication, its organization or subject matter with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Comments:

Reply requested Name : ______________________

Yes / No Job Title : ______________________

Address : ______________________

____

Cut or Fold Along Line

Reader's Comment Form

Fold and tape | **Please Do Not Staple** | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Technical Support Center
Department 977, Building 663-3
Highway 52 and NW 37th Street
Rochester, Minnesota 55901 U.S.A.

Fold and tape | **Please Do Not Staple** | Fold and tape

GG24-3321-01

Structured Query Language/400: A Guide for Implementation
OS/400 Release 3.0

GG24-3321-01

Printed in the U.S.A.

IBM